

Systems Software Report CA2

DT228

BSc in Computer Science

**Gary Kelly**

**C16380531**

School of Computer Science

TU Dublin – City Campus

**03/05/2020**

**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 |
| F2 | Server | Yes |
| F3 | Multithreaded connections | Yes |
| F4 | File Transfer | Yes |
| F5 | Transfer Authentication using Real and Effective ID’s | Yes |
| F6 | Synchronisation (Mutex Locks) | Yes |
|  | Video -> in compressed folder | Yes |

# *Feature 1 - Client Program*

The client socket program communicates with the server on port 8081 using the loopback address 127.0.0.1 and the IPV4 protocol. The client communicates with the server by sending three streams to the server and receiving an acknowledgement from the server for each stream. The program begins by connecting to the socket server and asking the user to enter a filename. This filename can be the relative or absolute path to the file being sent. After entering the filename, the user is then shown a menu of possible destinations to write the file to. This menu is implemented using a do while loop and a switch statement. After accessing this information, the client sends the user the files destination(destination path + filename), files content and the uid of the client. The uid is converted to a string sent to the server and converted back to an int on the server. At the end of the client program the client receives a final server message indicating as to the success / failure of the file transfers and closes the client connection. The below image shows a sample scenario of the above process:

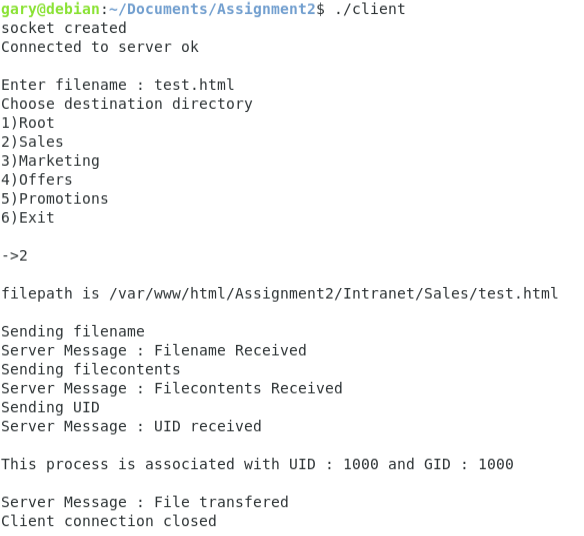


Figure 1 Client socket program output

# *Feature 2 – Server Program*

The server program begins by creating the socket, initializing its variables to communicate with the client on port 8081 using the loopback address 127.0.0.1 and the IPV4 protocol, binding these variables with the socket and listening for a client connection. A while loop is used to continually accept connections and create new threads (threading will be discussed in next feature). The file transfer connection is called on creation of each thread and handles the connection of each client. This function subsequently calls the receiveAcknowledge function each time it’s receiving information from the client. This function receives the information sent by the client stores it in the relevant location and writes an acknowledgement back to the client to confirm it has received the data. After receiving all data needed for the file transfer the server program changes permissions to the client’s permissions and attempts to write the contents of the file to its desired location. The client is notified whether this was successful / unsuccessful, the permissions of the program are reset to root and the thread is then closed. This process is shown below:

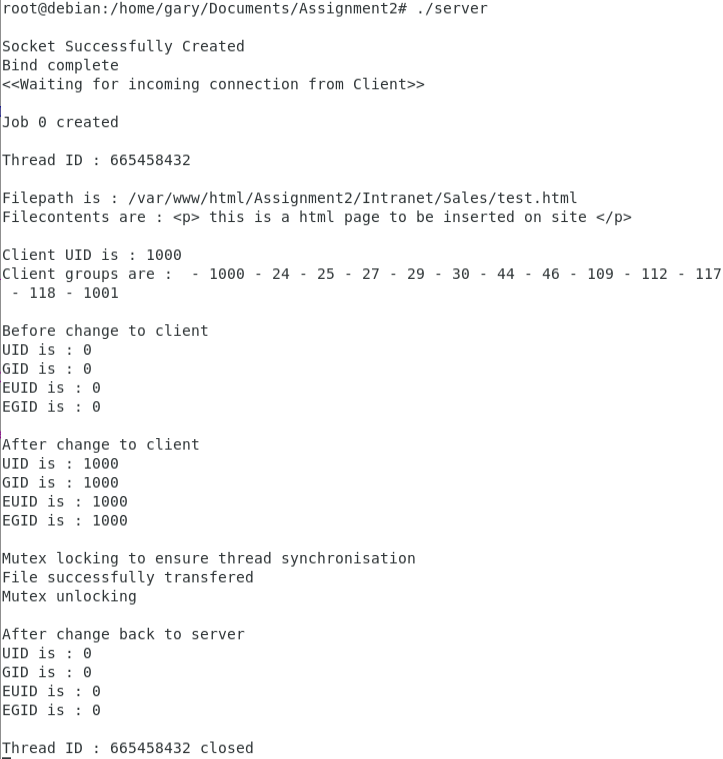


Figure 2 Server socket program output

# *Feature 3 - Multithreaded connections*

As discussed in the server feature the server offers the ability to serve multiple clients concurrently. This is achieved through the use of threading. A while loop is used to continuously accept connections. The new connection is allocated memory and created using pthread\_create. This command creates a new flow of control which executes the function for handling the client connection. At the start of this function the thread id is output to the console to confirm each client is on a different thread. One issue with multithreading is multiple threads attempting to access a shared resource which will be discussed in feature 6.

# *Feature 4 - File Transfer*

As discussed in feature 1, the client gives the files destination(destination path + filename) and files content to the server in order to be stored. These values are sent to the server using the send() command which sends a buffer to the server program using the connected socket. After each buffer is sent the client uses the recv() command to receive a message from the server whether this process was successful / unsuccessful. In addition to the file’s destination and files content the uid of the client is also sent in order to validate that the user has the relevant permission to write to the file destination. This process will be discussed further in the following section.

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

As discussed in features 1 and 4, the clients uid is sent to the server in order to validate that the user has the relevant permission to write to the file destination. The server receives this uid and uses the getpwuid() command to search the user database for an entry with a matching uid. After this the server program uses the getgrouplist() function to get the groups that the client is linked with. These groups are stored in an array using the gid\_t struct. A set of system calls are used to change the thread from root permissions to the client’s permissions using the clients uid and associated groups array. After changing to the client’s permissions, the program attempts to write the file contents to its destination before reverting back to the root’s permissions.

In addition to the program functionality, terminal commands were used in order to ensure each user only had permissions to folders depending on the groups they were in:

* The **usermod -a -G groupName username** command was used to add the relevant group permissions to each user.
* The **chgrp groupName /groupPath/groupName** command was used to assign each folder with its own group.
* The **chmod g+rwx /groupPath/groupName** command was used to alter the group permissions and allow the group to read, write and execute inside the folder.

# *Feature 6 - Synchronisation (Mutex Locks)*

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

As discussed in feature 3, synchronisation was achieved for the concurrent access to shared resources through the use of a mutex lock. A mutex is a lock that ensures all other concurrent threads cannot use the shared resource until it has been released. This was implemented by first initializing a global pthread\_mutex\_t variable in the server program, then locking that mutex before writing to the destination file inside the thread and then unlocking the mutex after finishing the write operation.

# *Conclusion*

In summary I’m very happy with my implementation of this project. All of the CTO’s requirements have been met, the programs are very simple to use and the output to the console is very clear. All error reporting is done through the use of perror commands that each give a clear indication as to where and why the program failed. I believe the implementation is similar to what would be used in an industry setting for a similar task, albeit extremely simplified. The code is as modular as possible with functions used in both the client and server code to handle sending and receiving messages.