OLLSCOIL TEICNEOLAÍOCHTA
BHAILE ÁTHA CLIATH

TU DUBLIN

TECHNOLOGICAL
UNIVERSITY DUBLIN

Systems Software Report CA2

DT282-4
BSc in Computer Science (International)

Jennifer Nolan
C16517636

School of Computer Science
TU Dublin – City Campus

3rd May 2020

Table of Contents

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

Have you included a video demo as part of the assignment: Yes
Link to Video:
https://drive.google.com/file/d/1aDRFIrbBCZzVnBWYH6OKdUPKPjiJCGi5/view?usp=sharing

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:

Jennifer Nolan

Jennifer Nolan

3rd May 2020

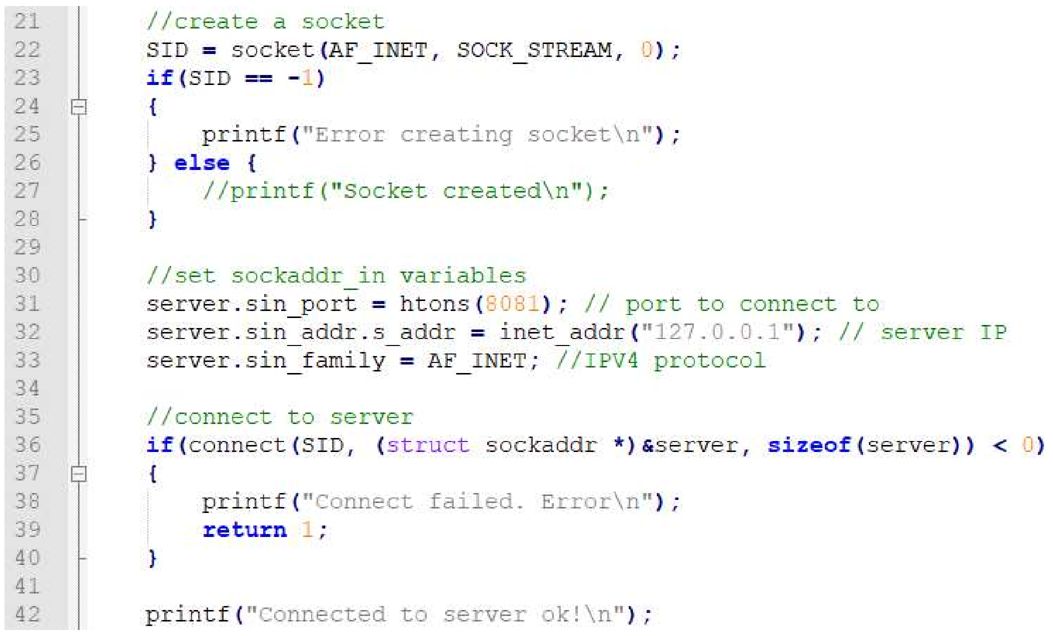
Feature 1 - Client Program

Feature one of this assignment consisted of the client program. The client program is where the user enters the required information about the file they want to transfer, which is then sent to the server to be implemented. The client program essentially connects to the server, takes in user input and sends the entered information to the server where the file transfer takes place.

The client program uses socket programming as it provides reliable two-way communication between the client and the server. The socket programming is necessary for this project as the user input is to be sent to the server in order to implement the file transfer and the server is to notify the user at the client side if the file transfer was completed successfully or not.

In order for a client and server to effectively communicate a specific socket connection sequence must be followed. Firstly, from a client perspective, the client makes a socket connection to the server. The client then sets the input and output stream used to send and receive data to and from the server. Lastly, when the file transfer is completed successfully, or the user exits the program the client closes the connection to the server.

With regard to this assignment specifically, the following was carried out to create a client program that uses socket programming to connect to a server. Firstly, the variables required to run the client were initialized. This included variables like SID and server which are both used to set up the client connection to the server. Next the socket is created, which can be seen at line 21 in the below code. Following from this, the socket variables are set, seen from lines 31 to 33 in the below code. These socket variables include setting the port for the client to connect to for communication, setting the server IP to localhost (127.0.0.1) and setting the socket to use the IPV4 protocol. Lastly, as part of the client server socket connection the client program connects to the server socket, seen at line 36 in the below code. Once the client is connected to the server the communication required to successfully complete a file transfer, as per the brief of this assignment, can be carried out.



```
    //create a socket
    SID = socket(AF_INET, SOCK_STREAM, 0);
    if(SID == -1)
    {
        printf("Error creating socket\n");
    } else {
        //printf("Socket created\n");
    }

    //set sockaddr_in variables
    server.sin_port = htons(8081); // port to connect to
    server.sin_addr.s_addr = inet_addr("127.0.0.1"); // server IP
    server.sin_family = AF_INET; //IPV4 protocol

    //connect to server
    if(connect(SID, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        printf("Connect failed. Error\n");
        return 1;
    }

    printf("Connected to server ok!\n");
```

Feature 2 – Server Program

Feature two of this assignment consisted of the server program. The purpose of the server for this assignment is that it receives a file from the client which is to be transferred to a new file location, which is specified by the user at the client end. The server also authenticates user credentials to ensure that the user has the proper authorisations to transfer the file to the specified directory, i.e. the user is part of the sales group so therefore can transfer files to the sales directory. Lastly, the server sends back messages to the client to inform the user whether the file was transferred successfully and if not why the file transfer was not successful.

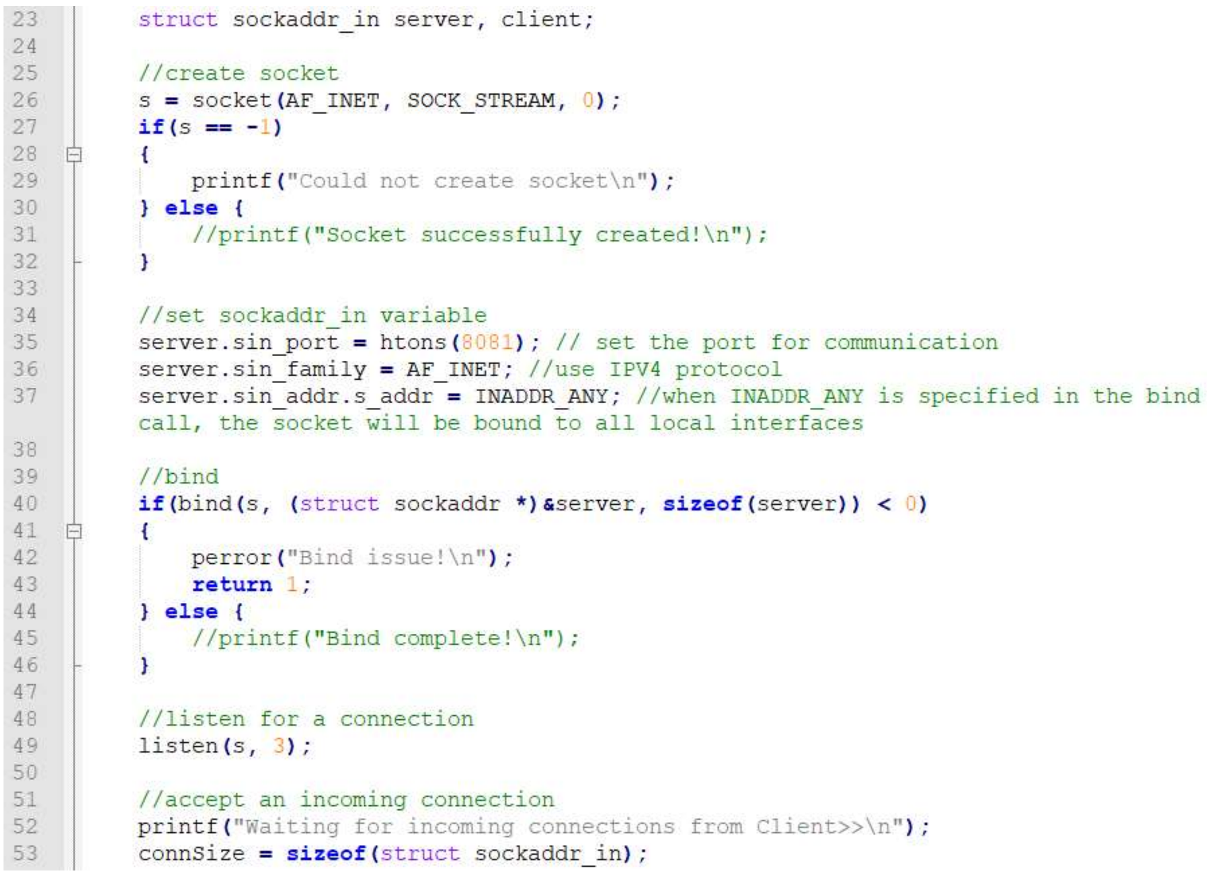
Like with the client program and for the same reasons, socket programming was also used for the server program. This socket programming provides two-way communication between the client and server programs.

The lifeline of communication between the client and server programs is managed using conditional statements that decide whether the transfer process continues or if the clients' connection to the server is closed. The following different conditions will close the connection between the client and server programs. The client program closes the connection to the server when the transfer has taken place successfully. The client also closes the connection to the server if the user decides to not complete the file transfer. However, the server closes the client connection when incorrect credentials are entered.

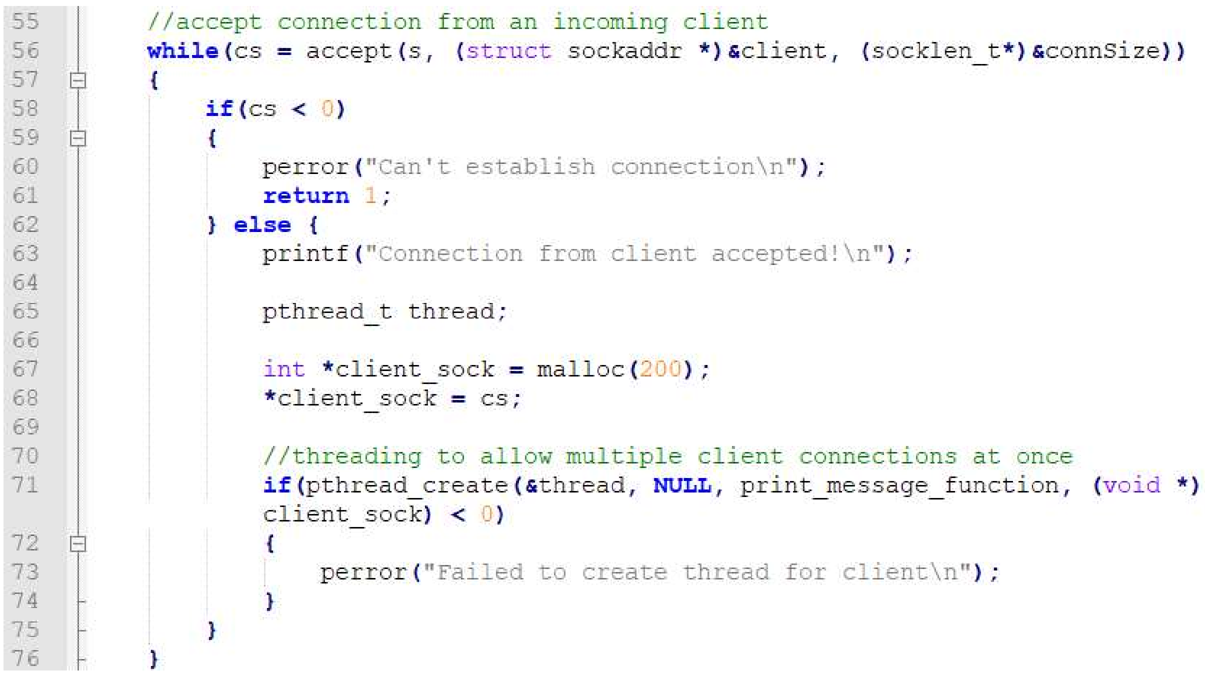
In order for a client and server to effectively communicate a specific socket connection sequence must be followed. From a server perspective the following occurs. Firstly, the server creates a socket and listens for a client connection. Next the client makes a socket connection to the server and the server decides whether to allow the connection or not.

Once the client connection is accepted the transmission of data can take place until the client closes its connection to the server.

With regard to this assignment specifically, the following was carried out to create a server program that uses socket programming to accept a client connection. Firstly, the server variables required to run the server program are declared and initialised. These variables include the socket descriptor and the client socket connection variable. From there the socket is created, as can be seen at line 26 in the below code. This socket is then configured. This configuration includes the setting of the port number for communication, as seen at line 35 in the below code. This port number assigned by the server allows the client to connect to the server socket using the server IP address, in this case localhost, and the port number. Once this is done the socket configuration is then bound to the socket, shown at line 40. Once the socket is bound the server continuously listens for a connection from the client. When the client connects to the server and the server accepts the connection, shown at line 56, the processes required to transfer a file to a specified directory takes place. This is done, using the input from the client, to transfer the file and its content to the correct new directory that the user has access to.



```
23      struct sockaddr_in server, client;
24
25      //create socket
26      s = socket(AF_INET, SOCK_STREAM, 0);
27      if(s == -1)
28      {
29          printf("Could not create socket\n");
30      } else {
31          //printf("Socket successfully created!\n");
32      }
33
34      //set sockaddr_in variable
35      server.sin_port = htons(8081); // set the port for communication
36      server.sin_family = AF_INET; //use IPV4 protocol
37      server.sin_addr.s_addr = INADDR_ANY; //when INADDR_ANY is specified in the bind
        call, the socket will be bound to all local interfaces
38
39      //bind
40      if(bind(s, (struct sockaddr *)&server, sizeof(server)) < 0)
41      {
42          perror("Bind issue!\n");
43          return 1;
44      } else {
45          //printf("Bind complete!\n");
46      }
47
48      //listen for a connection
49      listen(s, 3);
50
51      //accept an incoming connection
52      printf("Waiting for incoming connections from Client>>\n");
53      connSize = sizeof(struct sockaddr_in);
```



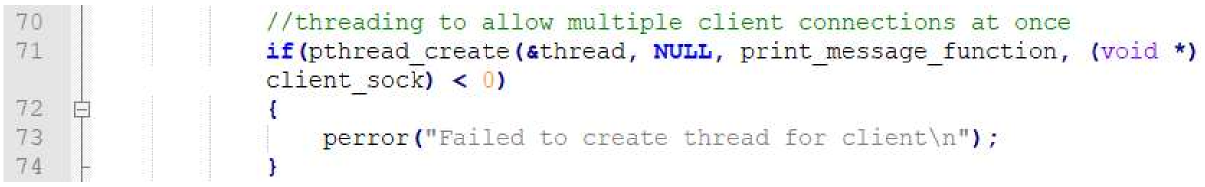
```
55      //accept connection from an incoming client
56      while(cs = accept(s, (struct sockaddr *)&client, (socklen_t*)&connSize))
57      {
58          if(cs < 0)
59          {
60              perror("Can't establish connection\n");
61              return 1;
62          } else {
63              printf("Connection from client accepted!\n");
64
65              pthread_t thread;
66
67              int *client_sock = malloc(200);
68              *client_sock = cs;
69
70              //threading to allow multiple client connections at once
71              if(pthread_create(&thread, NULL, print_message_function, (void *)
                client_sock) < 0)
72              {
73                  perror("Failed to create thread for client\n");
74              }
75          }
76      }
```

Feature 3 - Multithreaded connections

Feature three of this assignment consisted of multithreaded connections. The server program was able to offer concurrent client connections using multithreading techniques. This made the server program capable of handling multiple clients simultaneously. By including threading techniques into the server program of this assignment multiple connected clients are able to be run concurrently by the server on different threads. Each of these threads has a different path of execution and therefore essentially allows for the multitasking of client connections.

With regards to this assignment specifically, thread-based multitasking was used. This allowed the server program to take in multiple client connections and perform the task associated with each client concurrently. This essentially allowed the server to perform multiple tasks at once.

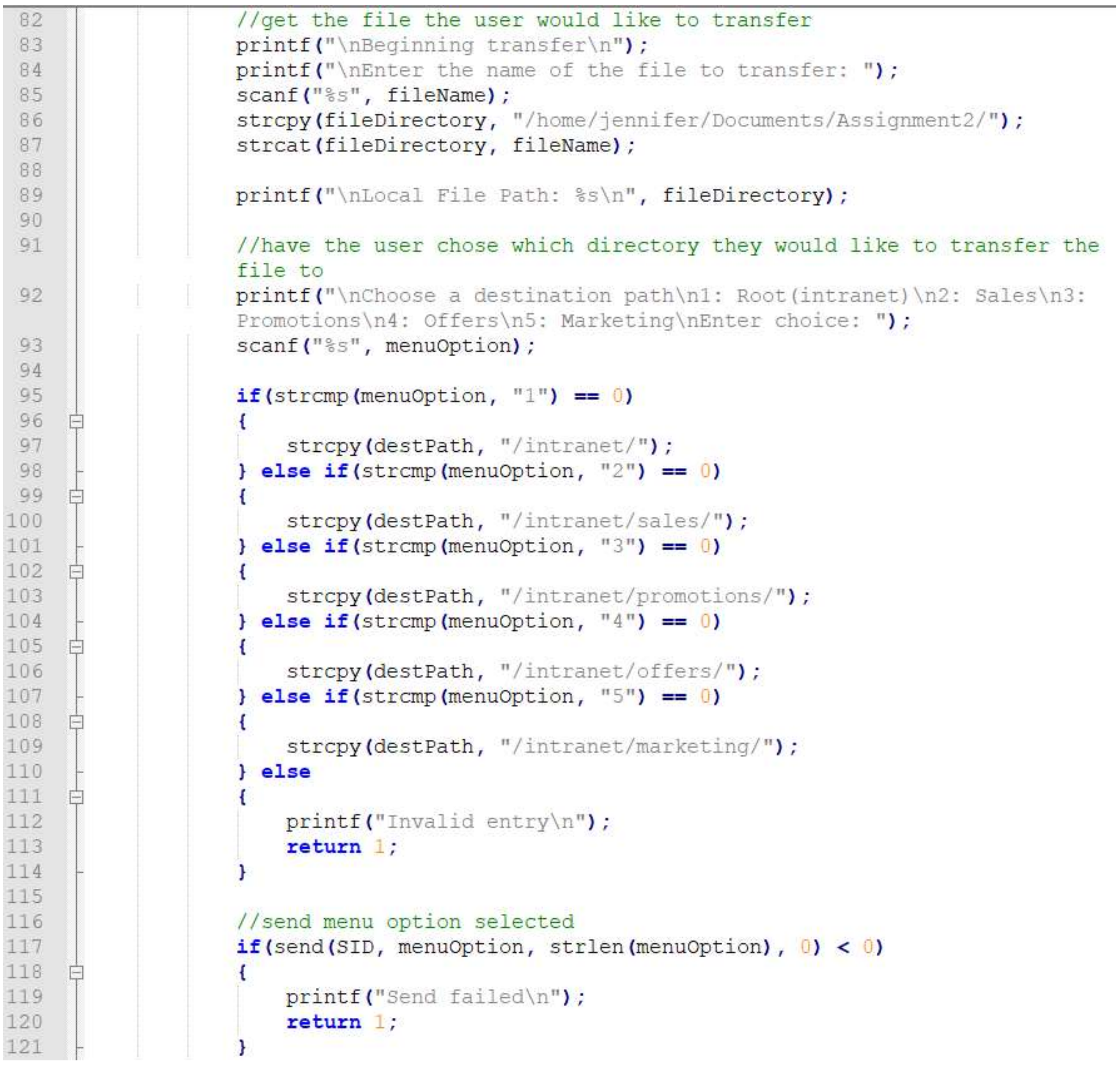
The below code snippet demonstrates how the multithreading was carried out as part of this assignment. After the client connection is accepted by the server program, the server program threads the connected client and this thread runs the function that carries out the file transfer process. This thread creation can be seen at line 71 in the below code. This threading technique allows for the connection of multiple clients to the server so that multiple file transfers can be executed by different clients concurrently.



```
70              //threading to allow multiple client connections at once
71              if(pthread_create(&thread, NULL, print_message_function, (void *)
                client_sock) < 0)
72              {
73                  perror("Failed to create thread for client\n");
74              }
```

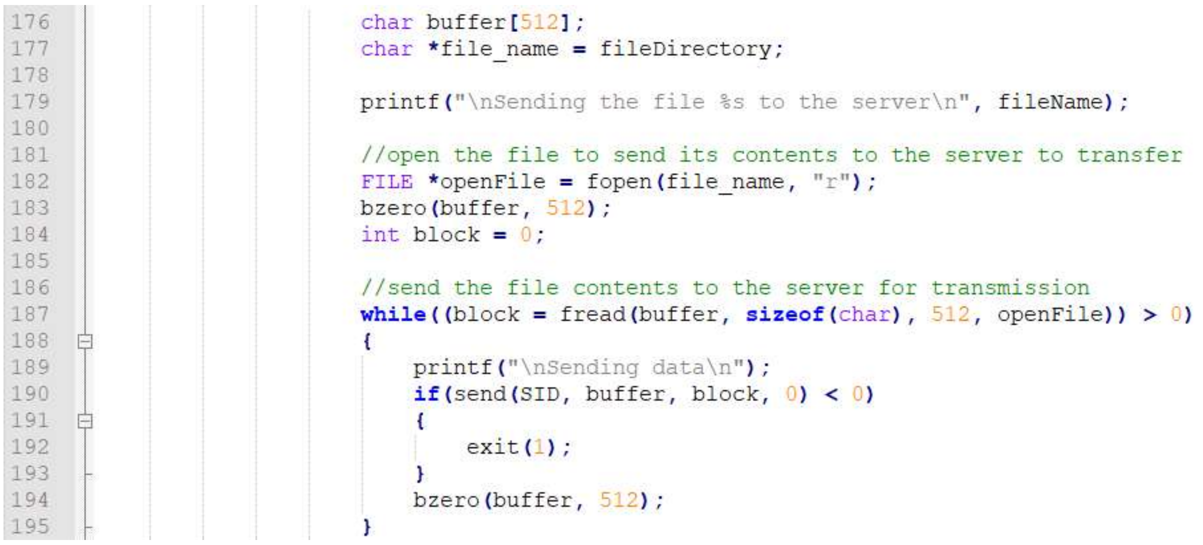
Feature 4 - File Transfer

Feature four of this assignment consisted of transferring files by sending a file from the client to the server, where the file is transferred. From the client programs perspective, the file name that the user would like to transfer is gathered from the users input at the console on the client side, shown in the below code at line 85. The user then enters the destination directory they would like the file to be transferred to. This destination directory and entered file name are concatenated together to be sent to the server for the file transfer, shown at line 141 in the below code.



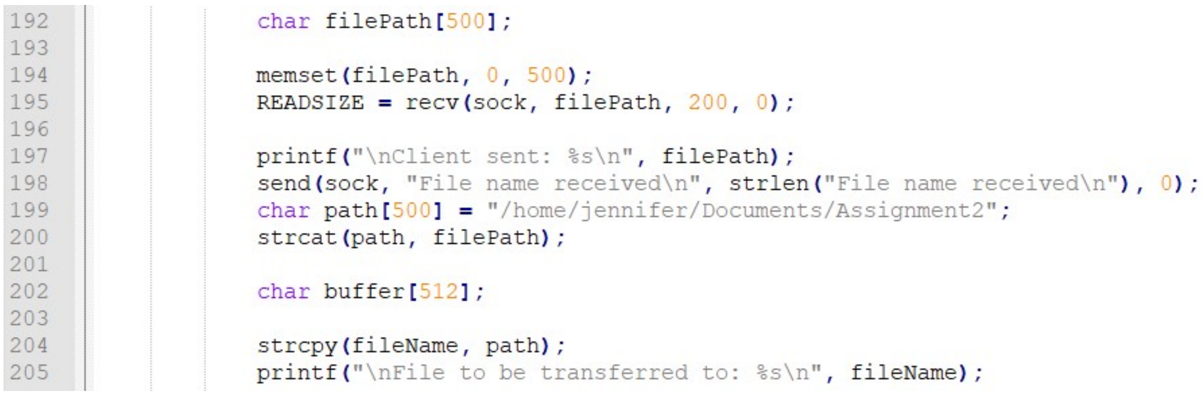
```
82          //get the file the user would like to transfer
83          printf("\nBeginning transfer\n");
84          printf("\nEnter the name of the file to transfer: ");
85          scanf("%s", fileName);
86          strcpy(fileDirectory, "/home/jennifer/Documents/Assignment2/");
87          strcat(fileDirectory, fileName);
88
89          printf("\nLocal File Path: %s\n", fileDirectory);
90
91          //have the user chose which directory they would like to transfer the
            file to
92          printf("\nChoose a destination path\n1: Root(intranet)\n2: Sales\n3:
            Promotions\n4: Offers\n5: Marketing\nEnter choice: ");
93          scanf("%s", menuOption);
94
95          if(strcmp(menuOption, "1") == 0)
96          {
97              strcpy(destPath, "/intranet/");
98          } else if(strcmp(menuOption, "2") == 0)
99          {
100             strcpy(destPath, "/intranet/sales/");
101         } else if(strcmp(menuOption, "3") == 0)
102         {
103             strcpy(destPath, "/intranet/promotions/");
104         } else if(strcmp(menuOption, "4") == 0)
105         {
106             strcpy(destPath, "/intranet/offers/");
107         } else if(strcmp(menuOption, "5") == 0)
108         {
109             strcpy(destPath, "/intranet/marketing/");
110         } else
111         {
112             printf("Invalid entry\n");
113             return 1;
114         }
115
116         //send menu option selected
117         if(send(SID, menuOption, strlen(menuOption), 0) < 0)
118         {
119             printf("Send failed\n");
120             return 1;
121         }
```

Once the server has verified that the current user has access to the directory the file is to be transferred to, the transfer can begin. As shown from lines 187 to 194 in the below code the client begins sending the contents of the transfer file to the server so that this file and its contents can be transferred completely to its new directory.



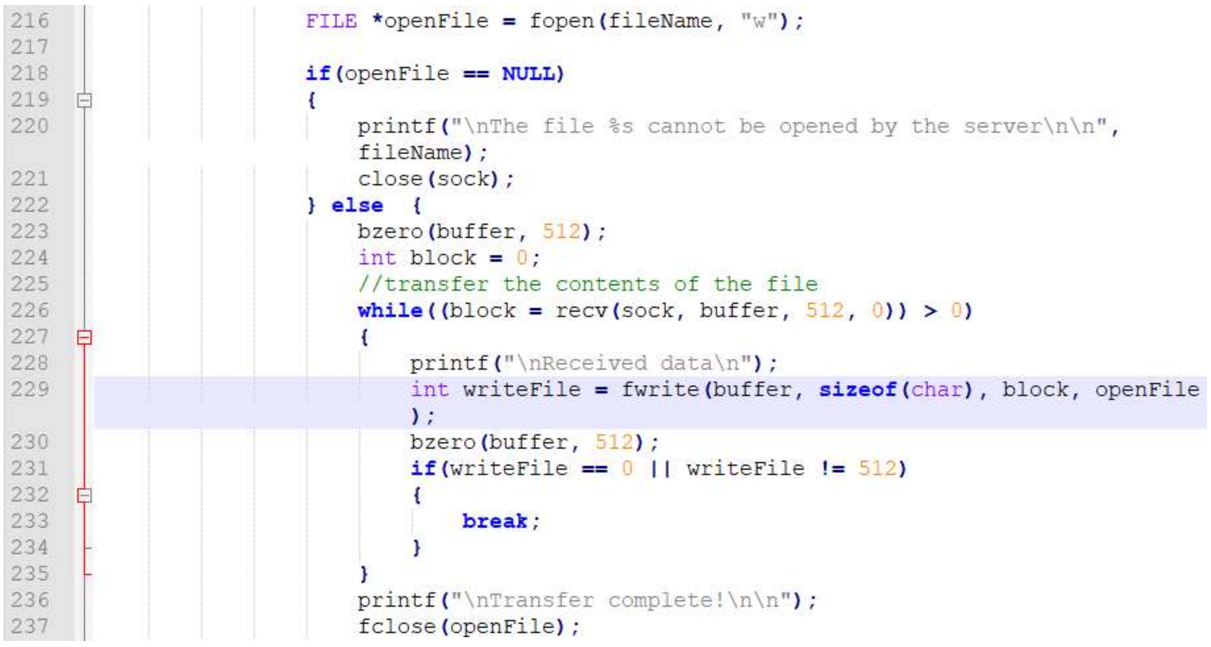
```
176 char buffer[512];
177 char *file_name = fileDirectory;
178
179 printf("\nSending the file %s to the server\n", fileName);
180
181 //open the file to send its contents to the server to transfer
182 FILE *openFile = fopen(file_name, "r");
183 bzero(buffer, 512);
184 int block = 0;
185
186 //send the file contents to the server for transmission
187 while((block = fread(buffer, sizeof(char), 512, openFile)) > 0)
188 {
189     printf("\nSending data\n");
190     if(send(SID, buffer, block, 0) < 0)
191     {
192         exit(1);
193     }
194     bzero(buffer, 512);
195 }
```

From the server programs perspective, the server first receives the file path for the file the user wants to transfer, shown at line 195 in the below code. The server also receives the file path for where the file will be transferred to, i.e. the marketing, root, offers, promotions or sales directories, the result of which gives the filename variable, shown at line 205 in the below code.



```
192 char filePath[500];
193
194 memset(filePath, 0, 500);
195 READSIZE = recv(sock, filePath, 200, 0);
196
197 printf("\nClient sent: %s\n", filePath);
198 send(sock, "File name received\n", strlen("File name received\n"), 0);
199 char path[500] = "/home/jennifer/Documents/Assignment2";
200 strcat(path, filePath);
201
202 char buffer[512];
203
204 strcpy(fileName, path);
205 printf("\nFile to be transferred to: %s\n", fileName);
```

After the server has verified that the user has the correct credentials to transfer the file to the directory, i.e. the user is part of the sales group so therefore can transfer files to the sales directory, the file and its contents received from the client, shown at line 226 in the below code, are transferred to the new directory. Once this is finished the transfer is complete and the user is informed via a message sent from the server to the client.

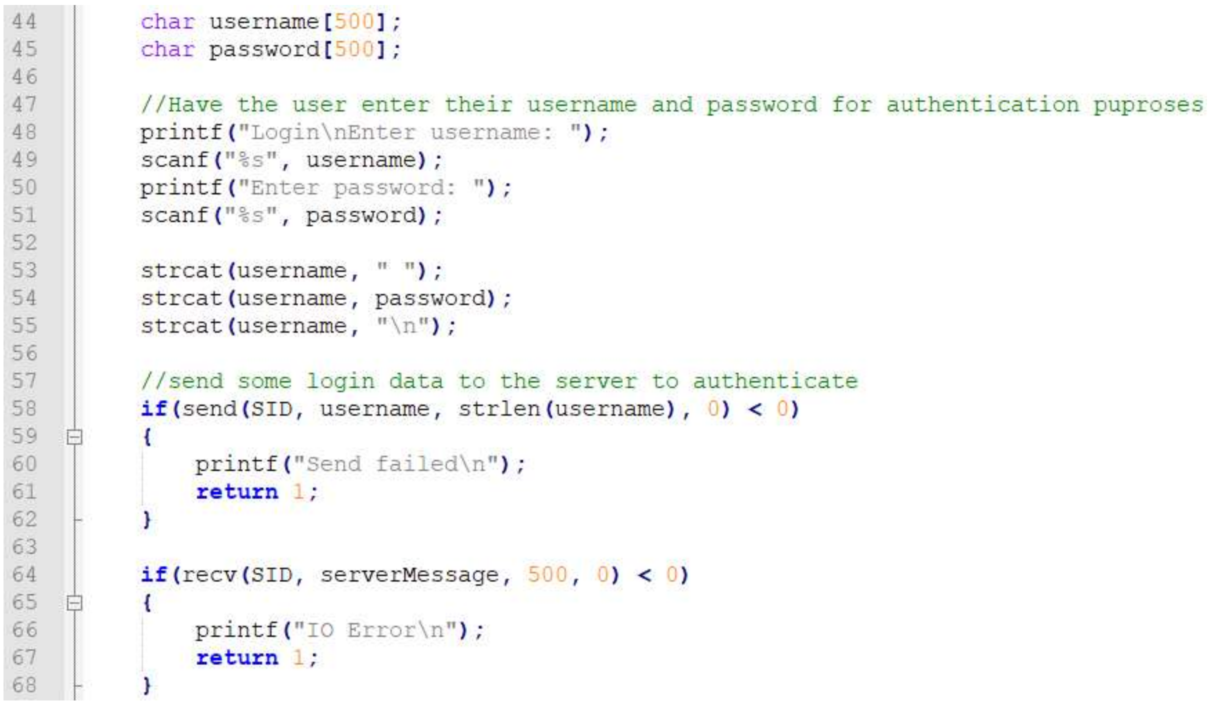


```c
FILE *openFile = fopen(fileName, "w");

if(openFile == NULL)
{
    printf("\nThe file %s cannot be opened by the server\n\n",
    fileName);
    close(sock);
} else  {
    bzero(buffer, 512);
    int block = 0;
    //transfer the contents of the file
    while((block = recv(sock, buffer, 512, 0)) > 0)
    {
        printf("\nReceived data\n");
        int writeFile = fwrite(buffer, sizeof(char), block, openFile
        );
        bzero(buffer, 512);
        if(writeFile == 0 || writeFile != 512)
        {
            break;
        }
    }
    printf("\nTransfer complete!\n\n");
    fclose(openFile);
```

Feature 5 - Transfer Authentication using Real and Effective ID's

Feature five of this assignment consisted of authenticating users using their ID's to determine if they can transfer a file to a specified directory. From a client perspective, the client program simply takes in the username and password that the user enters through the command line, as shown in the below code from lines 48 to 51. These user credentials, username and password, are then sent to the server to be verified, as shown at line 58 in the below code. The client is then informed from the server whether the user credential are valid or not.

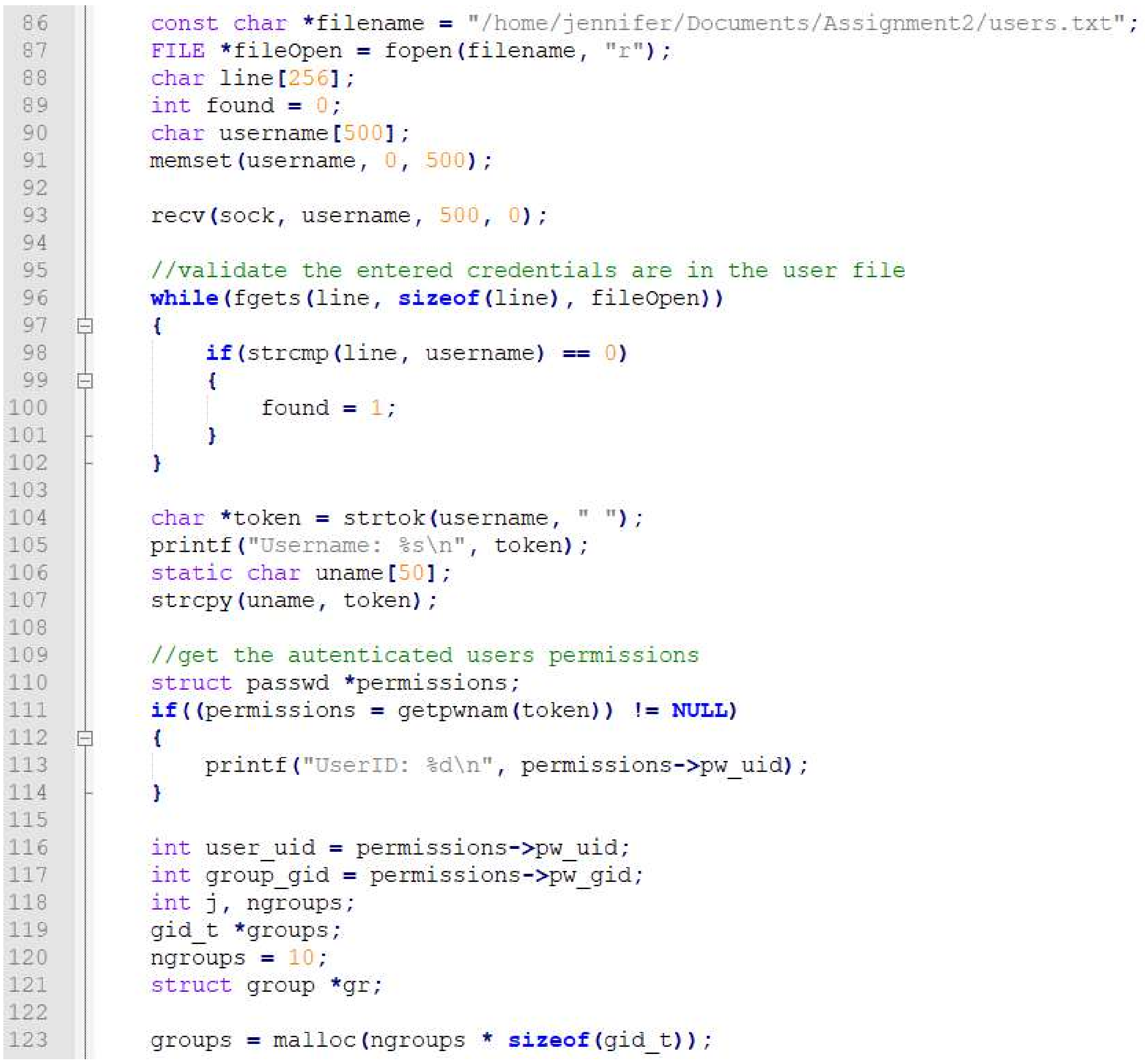


```c
44      char username[500];
45      char password[500];
46
47      //Have the user enter their username and password for authentication puproses
48      printf("Login\nEnter username: ");
49      scanf("%s", username);
50      printf("Enter password: ");
51      scanf("%s", password);
52
53      strcat(username, " ");
54      strcat(username, password);
55      strcat(username, "\n");
56
57      //send some login data to the server to authenticate
58      if(send(SID, username, strlen(username), 0) < 0)
59      {
60          printf("Send failed\n");
61          return 1;
62      }
63
64      if(recv(SID, serverMessage, 500, 0) < 0)
65      {
66          printf("IO Error\n");
67          return 1;
68      }
```

From a server perspective, the server program receives the credentials entered into the client and compares these to the user credentials on the system, as shown from lines 93 to 102 in the below code. The users created specifically for this assignment have their credentials stored in the user.txt file. The following users, stored in the system and in the users.txt file, have permissions to access the following directories:

- jen – offers, promotions, intranet
- admin – sales, marketing, offers, promotions, intranet
- user – marketing, intranet
- test – offers, intranet
- sarah – sales, intranet

Once the username and password, that were entered into the client, are verified, then the users file permissions, i.e. which files the user has access to, are gathered, shown from lines 110 to 128 in the code below.



```
 86        const char *filename = "/home/jennifer/Documents/Assignment2/users.txt";
 87        FILE *fileOpen = fopen(filename, "r");
 88        char line[256];
 89        int found = 0;
 90        char username[500];
 91        memset(username, 0, 500);
 92
 93        recv(sock, username, 500, 0);
 94
 95        //validate the entered credentials are in the user file
 96        while(fgets(line, sizeof(line), fileOpen))
 97        {
 98            if(strcmp(line, username) == 0)
 99            {
100                found = 1;
101            }
102        }
103
104        char *token = strtok(username, " ");
105        printf("Username: %s\n", token);
106        static char uname[50];
107        strcpy(uname, token);
108
109        //get the autenticated users permissions
110        struct passwd *permissions;
111        if((permissions = getpwnam(token)) != NULL)
112        {
113            printf("UserID: %d\n", permissions->pw_uid);
114        }
115
116        int user_uid = permissions->pw_uid;
117        int group_gid = permissions->pw_gid;
118        int j, ngroups;
119        gid_t *groups;
120        ngroups = 10;
121        struct group *gr;
122
123        groups = malloc(ngroups * sizeof(gid_t));
```

```c
    if(getgrouplist(token, group_gid, groups, &ngroups) == -1)
    {
        printf("No groups found\n");
    }
```

If the user has the required privileges to access the directory that is being transferred to, i.e. the name of the group privilege is the same as the selected transfer directory chosen by the user, then permissions to transfer the file to that directory is granted, as shown from lines 171 to 175 in the below code. However, if this is not the case then the user is informed that they do not have the right privileges and cannot transfer a file to that directory. However, because all users have access to the root intranet directory, every user who wants to transfer to the root intranet directory can, as shown at line 171 in the below code.

```c
164         //determine whether the authenticated user has access to the desired
            transfer directory
165         for(int j = 0; j < 10; j++)
166         {
167             gr = getgrgid(groups[j]);
168             if(gr != NULL)
169             {
170                 //printf("%s\n", gr->gr_name);
171                 if(strcmp(fileName, gr->gr_name) == 0 || strcmp(fileName, "intranet"
                    ) == 0)
172                 {
173                     access = 0;
174                     break;
175                 }
176                 else
177                 {
178                     access = 1;
179                 }
180             }
181         }
```

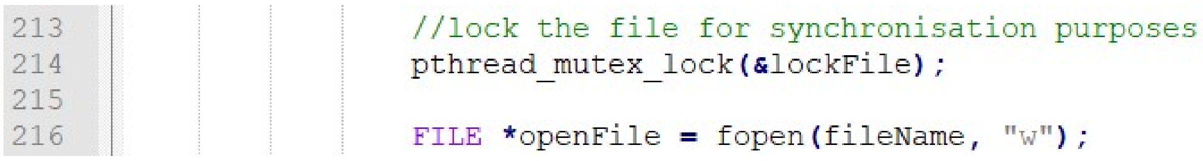
Feature 6 - Synchronisation (Mutex Locks)

Feature six of this assignment consisted of the synchronisation using mutex locks functionality. Mutex file locking is used in the server program when files are being transferred. The purpose of this was to ensure file synchronization. The mutex locks the specified file entered by the user. The mutex lock is then attached to the given file. As a result of using a mutex lock, if a thread needs to change the data in the file with the mutex lock attached to it then the thread must obtain access to the mutex lock. This means that any other threads that are trying to access the file at the same time cannot use the file until it has been released by the current thread using it.

In the code shown below, at line 15, the mutex lock is first initialized for the purpose of file synchronization.

```c
14  //file synchronization
15  pthread_mutex_t lockFile;
```

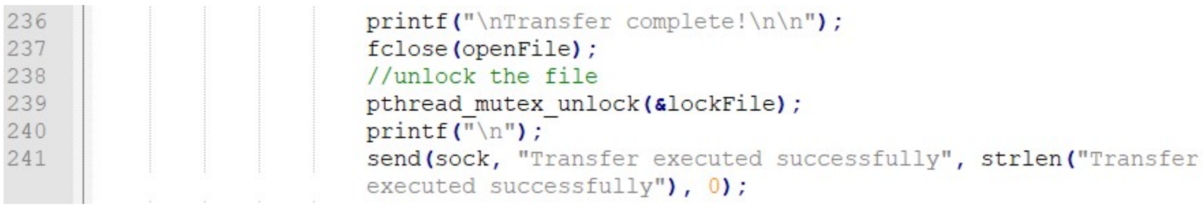
Once the file transfer is ready to begin the file being transferred is locked so that another thread cannot edit the same file when it is being transferred. If this locking were not conducted then it would cause file consistency issues. By locking the file being transferred no other thread can access the file until it is unlocked thus ensuring synchronization.



```c
//lock the file for synchronisation purposes
pthread_mutex_lock(&lockFile);

FILE *openFile = fopen(fileName, "w");
```

Once the file transfer has been completed successfully the file is unlocked so that another thread can use it for its own transfer.



```c
printf("\nTransfer complete!\n\n");
fclose(openFile);
//unlock the file
pthread_mutex_unlock(&lockFile);
printf("\n");
send(sock, "Transfer executed successfully", strlen("Transfer
executed successfully"), 0);
```

Conclusion

As seen from the above report all of the criteria set out in the brief of this assignment was met. The following features were implemented in this assignment solution:

- A client program that connects to the server socket program was developed
- A server socket program was developed
- The system is able to handle multiple client connections and transfers simultaneously
- File transfer restrictions are in place depending on the users group authorisations
- Any files that are successfully transferred are attributed, logged, to the user who conducted the transfer
- The client successfully takes in a file name through the console and transfers this file name onto the server
- The server program informs the client when a file transfer has been successful
- Any synchronization issues are resolved with the use of mutex locks

All of the above criteria was fully developed, tested and error handled as part of this assignment solution.