Systems Software Report CA2

DT228
BSc in Computer Science

Aron O'Neill
C16466214

School of Computer Science
TU Dublin – City Campus

02/04/20

Table of Contents

x

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

Table of Figures

Feature 1 - Client Program

Detailed description of the implementation and architecture choices made for the client program.

This project's client program has its own Makefile. As the majority of this assignment's functionalities are executed on its server, a client Makefile was not necessary. Despite this, it is best practice to follow the Separation of Concerns (SoC) and Single Responsibility principles.

Separation of Concerns (SoC) is the principle whereby its main purpose is to ensure that different concerns are no located in the same code base. Similarly, the Single Responsibility Principle states that aspects of a problem are in fact wo separate responsibilities and therefore should be implemented in separate classes.

The client program's architecture was centred around the implementation of various 'send()' requests. These requests were incorporated so that the server could handle the data being sent before sending back the appropriate responses, with these responses then being handled by the client. An example of this can be seen from the login functionality shown in the figure 1. Here, the 'LOGIN' message gets sent to the server so that it can keep track of the functionality being implemented. Following this, the entered username and password are sent, with their response then being handled so that the appropriate message can be displayed to the client.

```
send(sfd, "LOGIN", strlen("LOGIN"), 0);

printf("Username: ");
scanf("%s", username);
send(sfd, username, strlen(username), 0);

printf("Password: ");
scanf("%s", password);
send(sfd, password, strlen(password), 0);

// Receive Authentication
recv(sfd, response, 10, 0);
if (strcmp(response, "200") == 0) {
    puts("\nLogin Successful\n");
}
else {
    puts("\nLogin Unsuccessful\n");
    return 1;
}
```

Figure 1: Client Architecture (Login functionality)

After the client has successfully logged in, they will be met with a recursive menu so that the available options, transfer and logout are displayed to them. The client program's architecture was implemented in this way so that the client can transfer multiple files without the need to continuously login. Choosing the option to logout, sees their client connection close.

Feature 2 – Server Program

Detailed description of the implementation and architecture choice for the server program. The lifeline of the communication between the client and the server program and how this is managed should be described in detail.

The implemented server program also has its own makefile as shown in figure 2 so that it successfully follows the Separation of Concerns and Single Responsibility principles mentioned previously. Executing this Makefile enables the server, manager and date programs to be compiled using a single 'make' command.

```
CC = gcc
objects = Server.o date.o manager.o
headers = date.h manager.h
files = Server.c date.c manager.c
name = myprog
CFLAGS = -pthread

myprog : $(objects)
    $(CC) ${CFLAGS} -o $(name) $(objects)

Server.o : Server.c $(headers)
    $(CC) -c Server.c -lrt

manager.o : manager.c
    $(CC) -c manager.c

date.o : date.c
    $(CC) -c date.c

Clean:
    rm $(name) $(objects)
```

Figure 2: Server Makefile

This architectural choice was made so that the server's role would be to handle multiple connections being made using the threads created within it. Each thread then has its numerous functionalities executed within the 'manager.c' program. The date header file is then accessed within it so that the current date and time could be logged with each transfer carried out.

The server's architectural pattern was implemented was implemented in a similar manner to that of the clients'. After a client has connected to the server via its own thread, the server then calls the manager program so that the following while loop may be executed. Once here, the server will receive multiple client requests using the 'client_message' variable. Implementing the server architecture in this way enables the server to easily keep track of the functionality being carried out and send the appropriate responses back to the client.

```
while((read_size = recv(socket, client_message, MSG_SIZE, 0)) > 0) {

    // Login Functionality
    if (strcmp(client_message, "LOGIN") == 0) {
        printf("Login Started");

        // Server recevies user inputted username & password
        recv(socket, username, 50, 0);
        recv(socket, password, 50, 0);
```

Figure 3: Server & Client Communication Lifeline

In this example, the client entered detials are checked against that of the stored clients within the 'loginCredentials.txt' file. The communication between the client server is then influenced based on the verified counter variable shown in figure 4. If the client has been verified correctly, the server will send back a "200" so that the program may continue. However, should the user enter invalid details, the client program is sent a "401" error message and the current thread is exited. Implementing the server architecure is this way enables the client to be closed after an unsuccessful login while the server stays open.

```
while (fgets(line, 80, cred_file)) {
    sscanf(line, "username: %s password: %s UserUID: %s", user_auth, pass_auth, userUID);

    // Compares users details with stored users
    if((strcmp(username, user_auth) == 0) && (strcmp(password, pass_auth) == 0)) {
        userFound = 1;
    }
}
fclose(cred_file);

// Sends the appropriate resposne to the client if the user was found
if (userFound == 1) {
    send(socket, "200", sizeof("200"), 0);
} else {
    send(socket, "401", sizeof("401"), 0);
    puts("\nLogin Aborted");
    pthread_exit(NULL);
}
```

Figure 4: Example of Server Architecture using Stored User Credentials

After the client has logged in, the server is sent the client message "TRANSFER" in the same manner shown in figure 3, so that the communication between one another can continue to be maintained.

Feature 3 - Multithreaded connections

Describe how the socked server program has offered concurrent connections.

The socket server program offers concurrent connections by executing a while loop so that it can continuously listen for new connections. This is carried out in the 'server.c' program file and can be seen in the figure below. Here, a new thread is created for each client connection. Should the thread fail to connect, the appropriate error is the dispalyed. The number of clients who have connected to the server is the dispayed displayed using the count variable shown. Designing the server's achitectural pattern in the form of a while loop enables the server to handle any encountered errors while keeping its connection open.

```
// Initialise the thread to be used
c = sizeof(struct sockaddr_in);
    pthread_t thread_id;

// Accept new connections by creating a new socket for new client by implementing threads
while( (client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c))) {
    // Display No. of connection using counter
    count += 1;
    printf("\nNew connection accepted");
    printf("\nJob %d started\n", count);

    if( pthread_create( &thread_id, NULL, manager, (void*) &client_sock) < 0) {
        perror("could not create thread");
        return 1;
    }
    printf("Handler assigned\n\n");
}
```

Figure 5: Server Multithreading Connections

This is shown by implementing the provided logout functionality, entering an invalid user or file and attempting to transfer a file without the correct permissions. In any of these scenarios, the client will display the appropriate error message and close, while the server will continue to be open. In addition to this, multiple clients are able to access the server at the one time, with the ability to transfer being given to one client at a time so that the server does not risk losing modified data. This is shown in the figure below and emphasised in the video attached to this submission.

```
New connection accepted
Job 1 started
Handler assigned

Login Started
Login Aborted

New connection accepted
Job 2 started
Handler assigned
```

Figure 6: Multithreading Server Displayed Message

Feature 4 - File Transfer

Detailed description of the implementation process for sending a file from the client to the server.

The first step involved in this process was the client selecting their desired file to send followed by their chosen directory. After this, these options are then sent to the server where they are error checked as shown in the figure below.

```
// Receive & Handle filename & path
if ((bytes = recv(socket, client_filename, MSG_SIZE, 0)) == 0) {
    printf("No data recevied");
    free(socket_desc);
    pthread_exit(NULL);
    send(socket, "401", sizeof("401"), 0);
    puts("Transfer Aborted");
}

if ((bytes = recv(socket, client_filepath, MSG_SIZE, 0)) == 0) {
    printf("No data recevied");
    free(socket_desc);
    pthread_exit(NULL);
    send(socket, "401", sizeof("401"), 0);
    puts("Transfer Aborted");
}
```

Figure 7: Server receiving filename & file path from client

If the 'client_filename' and 'client_filepath' are invalid the appropriate error message is sent to the client so that their session may be ended. However, if successful, the server then checks the current user's permissions as described in feature 5 before sending its relevant response to the client so that it can be handled as shown below.

```
// Receive the repsonse & display to the client if they have invalid permissions
recv(sfd, response, 10, 0);
if (strcmp(response, "200") == 0) {
    puts("\nChosen Directory sent Successful");
} else {
    printf("\n *** Invalid Permissions *** \n");
    return 0;
}

FILE *fp = fopen(message, "r");
if(fp == NULL){
    perror("File");
    return 0;
}
```

Figure 8: Client handling transferring permissions

If the current user has the required permissions, the file to be transferred is then opened, before iterated over, using a 'sendbuffer' so that it's contents can be sent. This is achieved using the loop shown in figure 9. The file is read until all its contents have been sent. Following this, the client then waits for a response from the server to display whether the transfer has been a success or not.

```
strcat(file, "?");

strcpy(sendbuffer, message);
send(sfd, sendbuffer, sizeof(sendbuffer), 0);
bzero(sendbuffer, sizeof(sendbuffer));
recv(sfd, sendbuffer,300, 0);
bzero(sendbuffer, sizeof(sendbuffer));

// After opening the chosen file, send its contents to the server
while((b = fread(sendbuffer, sizeof(char), sizeof(sendbuffer), fp))>0 ){
    send(sfd, sendbuffer, b, 0);
    bzero(sendbuffer, sizeof(sendbuffer));
}

// Resceive & handle whether the transfer has been a success
recv(sfd, response, 10, 0);
if (strcmp(response, "200") == 0) {
    printf("\n *** Transfer Successful *** \n\n");
} else {
    printf("\n *** Transfer Failure *** \n");
    return 0;
}
fclose(fp);
```

Figure 9: Client sending file contents

The client receives this information in a similar manner, by sending a "200" message in response to every message being sent from the client. It implements this until all its contents have been read into the newly desired file as outlined by the path, 'server_filepath'.

```
// Read in the file to be transferred contents
while((read_size = recv(socket, client_message, 2000, 0)) > 0)
{
    char *ptr = strtok(client_message, delimiter);
    strcat(base_path, testFileName);

    // Write to the desired path created previously
    fp = fopen(server_filepath, "w");

    if(fp != NULL) {
        fwrite(client_message, sizeof(char), read_size, fp);
        send(socket, "200", sizeof("200"), 0);
    }
    else {
        perror("File Write");
    }
    fclose(fp);

    write(socket, client_message, strlen(client_message));
    memset(client_message, 0, 2000);

} // end while
```

Figure 10: Server receiving file contents

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

Detailed description the process used to determine if a specific user is permitted to transfer a file to the Marketing/Offers/Sales/Promotions folders in the root Intranet folder.

After the chosen file and path have been received correctly as detailed in the previous feature, the 'groupList.txt' is opened and read in as shown in the figure below.

```
// Open groups file needed to check privileges
FILE *group_file = fopen(GROUP_DIR, "r");
if (group_file == NULL) {
    printf("Error opening userCredentials file\n");
    exit(1);
}
while (fgets(line, 80, group_file)) {
    sscanf(line, "group: %s gUID: %s users: %s", group_auth, guid_auth, users_auth);

    // Checks if the group read in matches the users directory choice
    if((strcmp(client_filepath, group_auth) == 0)) {
        printf("\n%s Group Users: %s\n%s Group IDs: %s\n", client_filepath, users_auth, client_filepath, guid_auth);

        // Checks if the chosen group contains the current user's GUID
        if(strstr(guid_auth, userUID) != 0) {
            verified = 1;
        }
    }
}
fclose(group_file);
```

Figure 11: Server transfer authentication mechanism

Here, it can be seen how each group is checked against the users desired path so that only their desired option is retrieved. Following this, the list of stored UIDs are checked to see if they contain the current user's UID, and if so, verified is given the value of 1. This acts as a counter to track whether the current user has the necessary privileges to transfer the file.

After using this variable to check if the user has been verified, the file's owner is then retrieved using real and effective IDs, with this then being displayed to the server. Despite the file and directory being given every possible permission, it would not alter the file's owner. This seems to be a common problem experienced with VirtualBox. The workings of the real and effective IDs are illustrated in figure 13.

Despite this, the server's logging system clears shows the user who has made each change to the file, the current directory and time of which it occurred. This is shown in figure 12.

```
uid_t uid = getuid();
uid_t gid = getgid();
uid_t ueid = geteuid();
uid_t geid = getegid();

// Display original file owner UIDs
printf("\nOriginal File Owner IDs are:\nUser ID: %d\n", uid);
printf("Group ID: %d\n", gid);
printf("E User ID: %d\n", ueid);
printf("E Group ID: %d\n", geid);

int myUID = atoi(userUID);
printf("Current User UID is %d\n", myUID);

// Set current user's UID as file owner UIDs
setreuid(myUID, uid);
setregid(myUID, gid);
seteuid(myUID);
setegid(myUID);

printf("\nNew File Owner IDs are:\nUser ID: %d\n", getuid());
printf("Group ID: %d\n", getgid());
printf("E User ID: %d\n", geteuid());
printf("E Group ID: %d\n", getegid());
```

Figure 12: Server Real & Effective IDs

```
User: aron
Date: 03-05-2020---07:29:50PM
Updated File: test.txt
Updated Directory: root
----------------------------------------
User: aron
Date: 03-05-2020---07:34:51PM
Updated File: test.html
Updated Directory: sales
----------------------------------------
User: aron
Date: 03-05-2020---09:53:45PM
Updated File: test.txt
Updated Directory: sales
----------------------------------------
```

Figure 13: Log.txt Output Detailing Transfer

If the client choses to send the file to the root directory on the server called 'intranet', the server executes the following code so that its file path is edited according. This allows the same functionality to be carried out.

```
// Modifies the created file path based on if they have selected intranet folder or not
if((strcmp(client_filepath, "intranet") != 0)) {
    strcat(server_filepath, client_filepath);
    strcat(server_filepath, "/");
}
```

Figure 14: Intranet (Root) directory server code

Feature 6 - Synchronisation (Mutex Locks)

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

Synchronisation was achieved using mutex locks on the server so that the required resources may be locked when a client is implementing a transfer. It is implemented after the client has been verified to confirm that they have the required permissions needed to transfer the file to their chosen directory.

Executing the 'pthread_mutext_lock' command enables the transfer functionality to be disabled for any other clients trying to access it on another thread. This ensures that two clients can't modify a resource at the same time.

```
// If the user has the required privileges
if (verified == 1) {
    /* Used to lock the thread to ensure other clients can't access the thread and
       modify data at the same time so that each thread will have to be seperate
       for each user */
    pthread_mutex_lock(&lock);
```

Figure 15: Synchronisation - Mutex Lock

After the file has been transferred, the mutex lock is then unlocked as shown below so that the rest of the server functionality can be carried out. This includes logging the transfer information. Unlocking the mutex before the logging functionality allows other clients to transfer their desired files with the least waiting time. Thus, ensuring a more efficient system.

```
/* Unlocks the thread to allow other client's to transfer their file using
   another created thread */
pthread_mutex_unlock(&lock);

fwrite(client_message, sizeof(char), read_size, fp);
printf("\nSuccessfully transfered file\n");
```

Figure 16: Synchronisation - Mutex Lock

Conclusion

Summary of the implementation and achievement

The implemented project achieved all of the CTO's requirements and can now be used as a template for other projects. It also helped to enhance my technical skills in a Linux environment.

The created project is extremely beneficial to teams who need to monitor their resources with absolute precision. Implementing the multithreading, transferring, synchronisation and logging features ensure that this is indeed the case. These features then allow teams to monitor who has access to each directory and who has made the most recent changes.