CSE 344
SYSTEM PROGRAMMING

FINAL
REPORT

OGUZ MUTLU
1801042624

1. PROBLEM DEFINITION

The server side should be capable of handling multiple clients simultaneously, functioning as a multi-threaded internet server. Upon establishing a connection with the server, the directories on both the server and client sides need to be synchronized. This means that any new file created, deleted, or updated on the server should reflect the same changes on the client side, and vice versa.

BibakBOXServer [directory] [threadPoolSize] [portnumber]

* **Directory :** Specific area
* **threadPoolSize :** Maximum number of threads active at a time
*__portnumber:__ port of server connections

BibakBOXClient [dirName] [portnumber]

***Directory :** Specific area
***portnumber:** port of server connections

2.**PROBLEM SOLUTION APPROACH AND IMPLEMENTATION**

First of all, a common communication area was created by using socket system calls to enable communication over the socket.

```
struct file_informations
{
    char filename[512];
    char modification_time[256];
    unsigned int file_modes;
};
```

A struct was created for the files that needed to be transmitted from the source file to the target file, first holding the filenames, then the file modification times, and finally the file modes. Thanks to the struct seen in the photo on the side, it is ensured that the names and times of the files are kept on independent machines or in separate files. Information about the server can be found in the client file, and information about the client in the server file.

In addition, all these information structures for the client are defined globally for easy shopping between functions. Since each client will be connected to a separate, it is not possible to change this information on the client side, on the other client side.

```
struct client_socket
{
    char *client_src;
    int fd;
};
```

Another structure that I keep on the client side is a file name taken from the user that I have shown on the side, and the socket file descriptor, through which information about the content of this file can be transmitted and communicated.

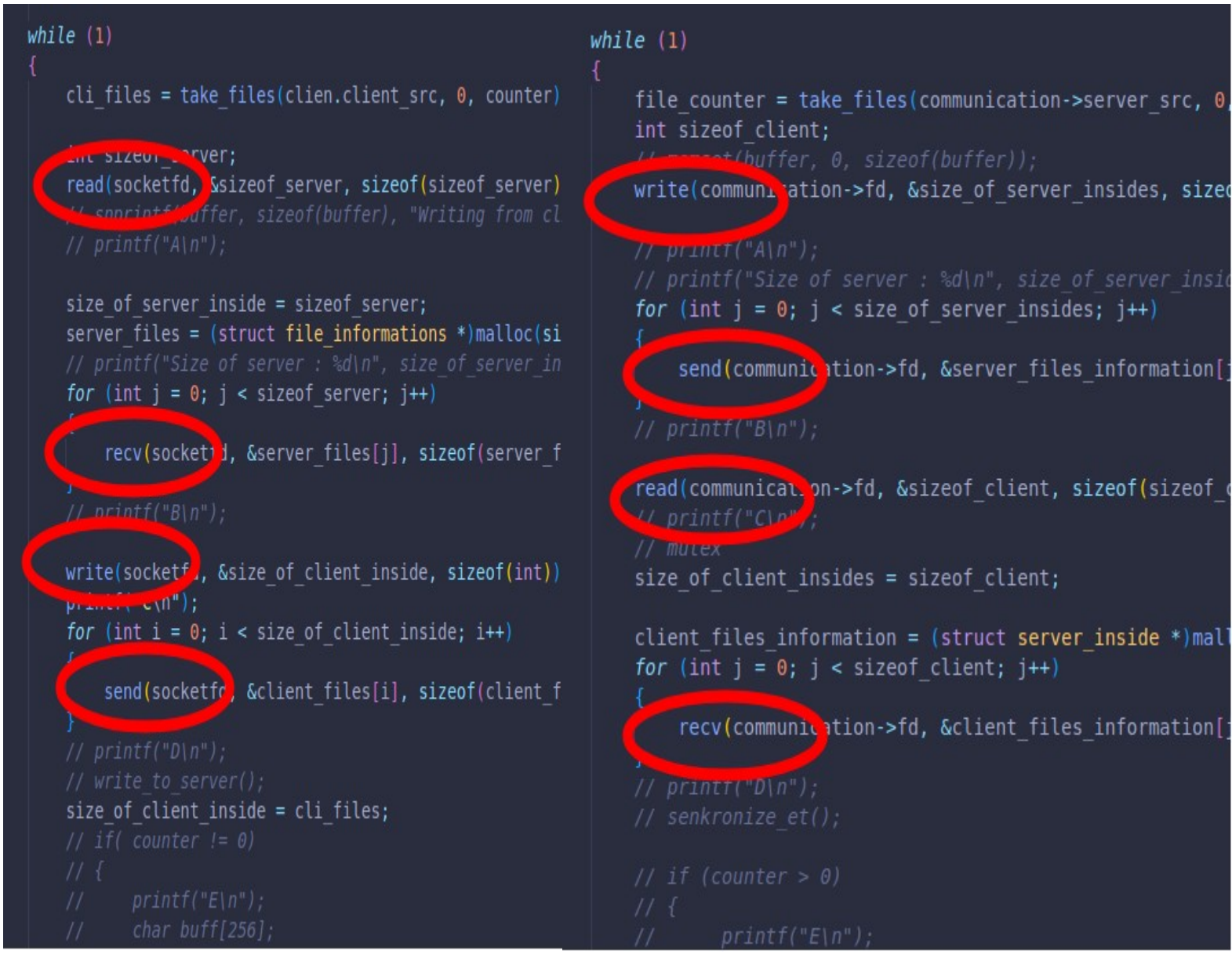
Since the above structure will have a unique value for each client, it is defined as a global value.

```
while (1)
{
    struct sockaddr_in client_addr;
    socklen_t cliend_addr_len = sizeof(client_addr);
    // memset(buffer, 0, sizeof(buffer));

    new_client_fd = accept(socketfd, (struct sockaddr *)&client_addr, &cliend_addr_len);

    srv.fd = new_client_fd;

    memset(buffer, 0, sizeof(buffer));
    snprintf(buffer, sizeof(buffer), "Client accepted\n");
    write(STDOUT_FILENO, buffer, sizeof(buffer));
    if (new_client_fd == -1)
    {
        perror("Failed to accept socket2");
        exit(EXIT_FAILURE);
    }
    int s = pthread_create(&threads[i], NULL, synchronize_server_side, &srv);
    if (s != 0)
    {
        perror("pthread_create");
        return 1;
    }
    i = (i + 1) % thread_pool_size;
}
```

Then, each client to be connected on the server side is expected to be accepted as it will process through a new thread. A **new struct sockaddr** value should be assigned to each accepted value because it is required to have a different socket fd for each new client, for this, after the first client connects to the server, the other while loop starts again. It will return and it will be allowed to wait in the **accept()** part. In this way, it is aimed to fulfill the multiple client request.

This process can be repeated until the maximum pool is equal to the size value. The mode receiving process at the bottom performs the rewrite process to the first client when the job reaches the end.

2.1 SOCKET SYNCHRONIZATION



```
while (1)
{
    cli_files = take_files(clien.client_src, 0, counter)

    rver;
    read(socketfd, &sizeof_server, sizeof(sizeof_server)
    // printf("A\n");

    size_of_server_inside = sizeof_server;
    server_files = (struct file_informations *)malloc(si
    // printf("Size of server : %d\n", size_of_server_in
    for (int j = 0; j < sizeof_server; j++)
    {
        recv(socketfd, &server_files[j], sizeof(server_f
    }
    // printf("B\n");

    write(socketfd, &size_of_client_inside, sizeof(int))
    printf("C\n");
    for (int i = 0; i < size_of_client_inside; i++)
    {
        send(socketfd, &client_files[i], sizeof(client_f
    }
    // printf("D\n");
    // write_to_server();
    size_of_client_inside = cli_files;
    // if( counter != 0)
    // {
    //     printf("E\n");
    //     char buff[256];
```

```
while (1)
{
    file_counter = take_files(communication->server_src, 0,
    int sizeof_client;
    // memset(buffer, 0, sizeof(buffer));
    write(communication->fd, &size_of_server_insides, sizeo

    // printf("A\n");
    // printf("Size of server : %d\n", size_of_server_insi
    for (int j = 0; j < size_of_server_insides; j++)
    {
        send(communication->fd, &server_files_information[j
    }
    // printf("B\n");

    read(communication->fd, &sizeof_client, sizeof(sizeof_c
    // printf("C\n");
    // mutex
    size_of_client_insides = sizeof_client;

    client_files_information = (struct server_inside *)mall
    for (int j = 0; j < sizeof_client; j++)
    {
        recv(communication->fd, &client_files_information[j
    }
    // printf("D\n");
    // senkronize_et();

    // if (counter > 0)
    // {
    //     printf("E\n");
```

The part shown on the left is an image from the source code written for the client side and the part shown on the right is a server side. First of all, thanks to the take_files function, each party saves the names of the files, modification times and file modes in the structure I have mentioned above, and in this way the files in the content of each resource are kept.In this structure, file keeping is done dynamically.

The content of the struct array is initialized with a size of 5, and every time it fills up, 5 more places are added. Then, the phase of waiting for the contents of the files that will come from the server for the client side begins.

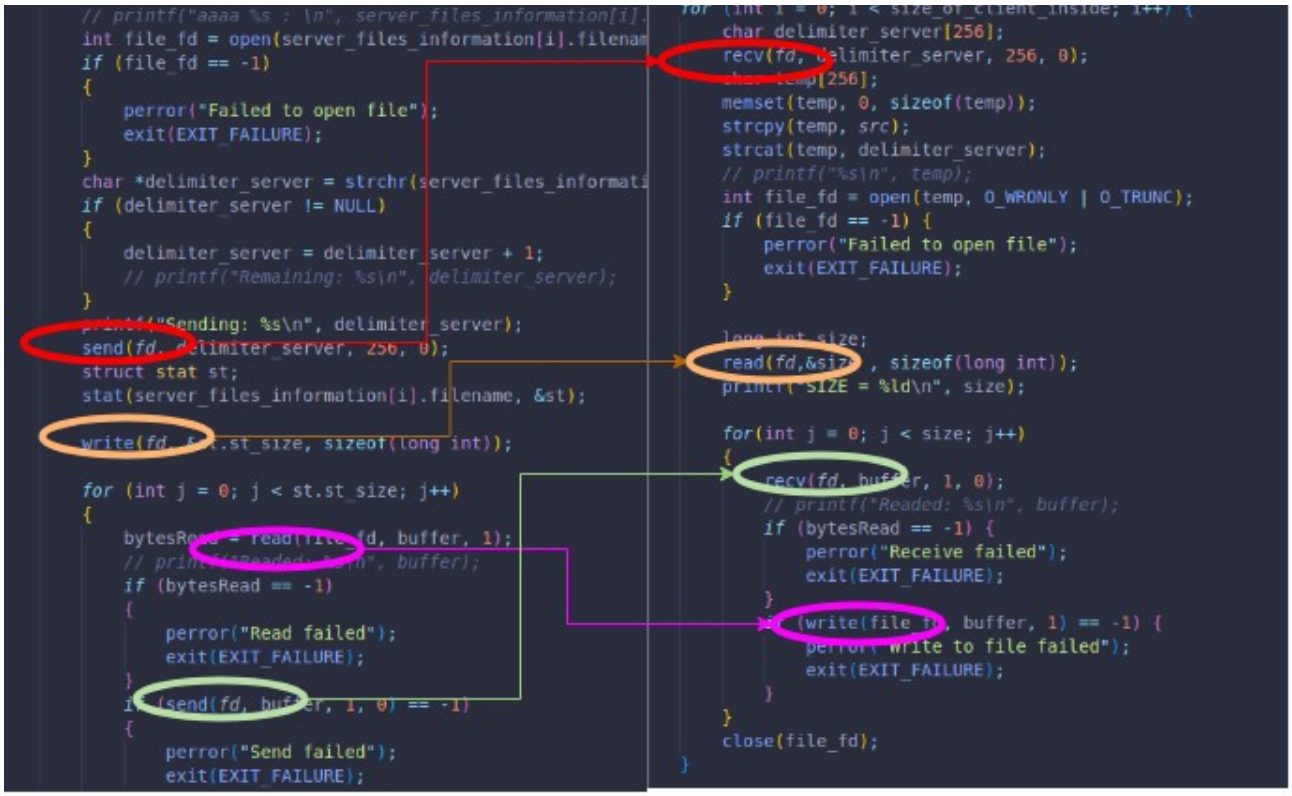
After this information is obtained, this time it takes the struct with all the names one by one in a for loop to create the files on its side.

This process then happens exactly the opposite, this time the server waits for the information of how many files are on the client side, and after this information comes, it keeps the names of these files in the client_files_information variable.

NOTE: Since it is not specified which of the files will be synchronized first, I synchronized all the files on the client and server first, and when the program first starts, all files in all sources are written to each other.

When all this information is obtained, the two files must now be compared (according to the modification time and names). This comparison takes place in the function called compare(). This function, which is a separate function for the client and server side, finds the differences in the two files and completes the missing ones.

After the found files are saved in a struct again, the synchronization process is started. This synchronization process is provided as follows.



write_to_client

read_from_serverx

We send the file names on the server side one by one to the write_to_client function. The part marked in red contains the name of the file whose content will be copied. then we get how many bytes of data are in it with the help of stat(). After sharing this information we have received with the help of socket, one side will now read the file and the other side will perform the writing of the file. This process will take place for both server and client.

According to the changing byte number in the file, all values are saved in the buffer one by one and writing to the file is done.

2.2 UNSUCCESSFULL ATTEMP

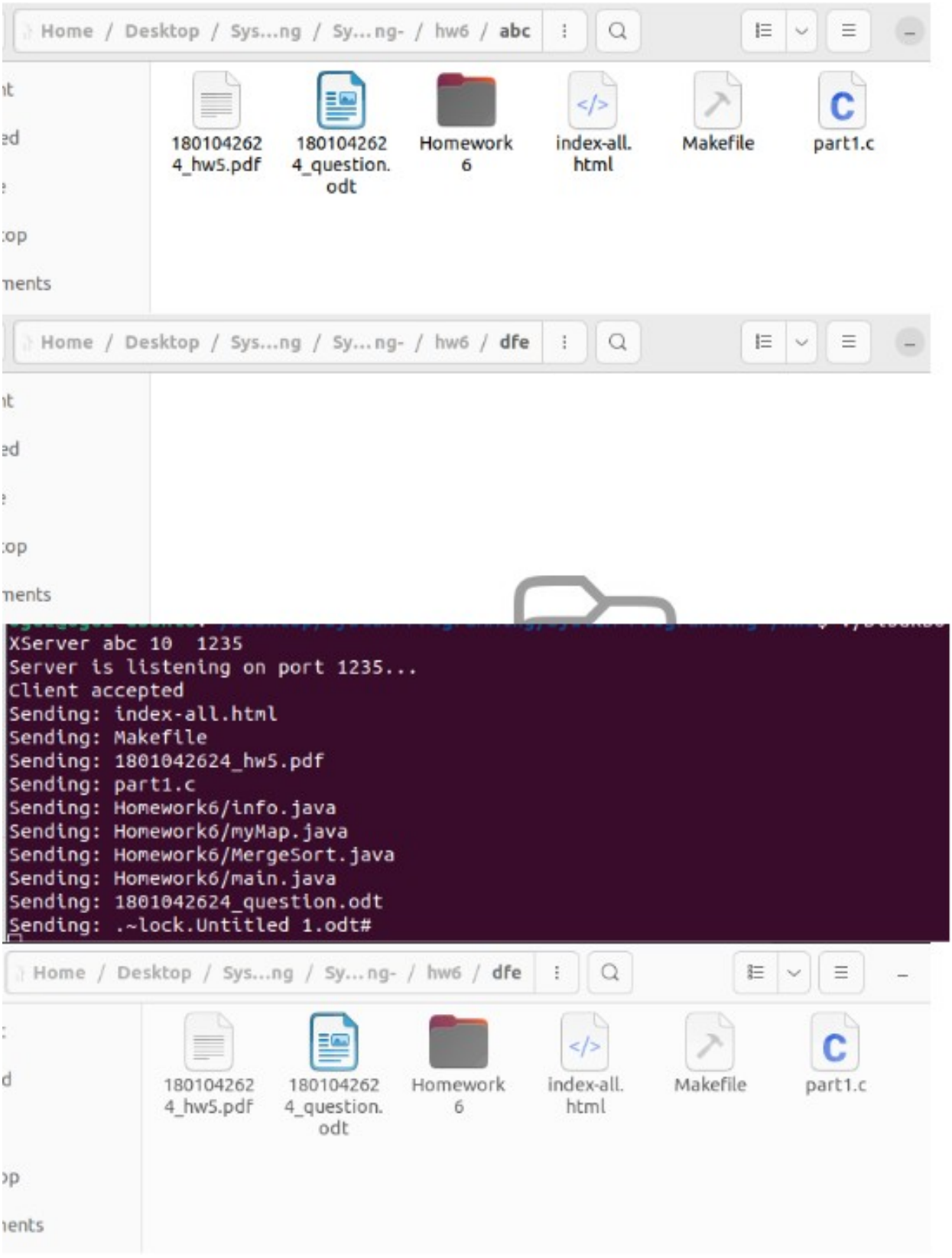
```c
if(flag == 1)
{
        char buff[256];
        printf("G\n");
        if(i != size_of_server_insides -1){
            printf("H\n");
            snprintf(buff, sizeof(buff), "continue");
            send(communication->fd,buff, sizeof(buff),0);

        }
        else{
            memset(buffer, 0, sizeof(buff));
            snprintf(buff, sizeof(buff), "quit");
            send(communication->fd,buff, sizeof(buff),0);
        }
        printf("Writing to a server ");
        write_client(delimeter_srv, i);
}
```

```c
if( counter != 0)
{
    printf("E\n");
    char buff[256];
    strcpy(buff, "continue");
    while(strncmp(buff, "quit", 5)!= 0)
    {
        printf("F\n");
        memset(buff, 0, sizeof(buff));
        recv(socketfd, buff, sizeof(buff), 0);
        printf("BUFFFF : %s\n", buff);
        read_from_server(clien.client_src);
    }
}
```

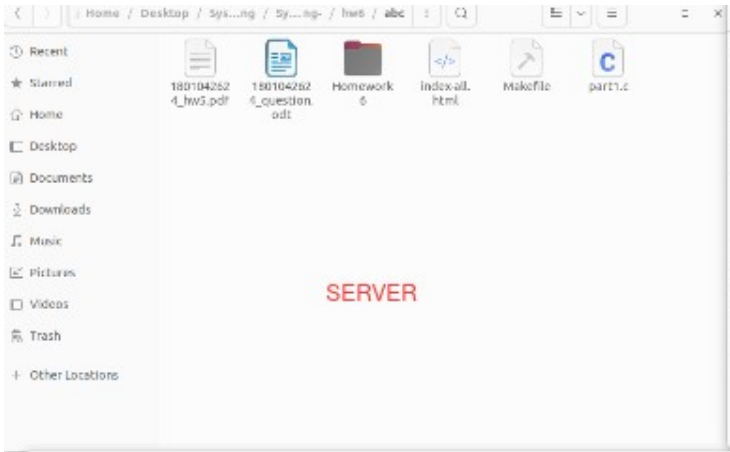
CLIENT SERVER

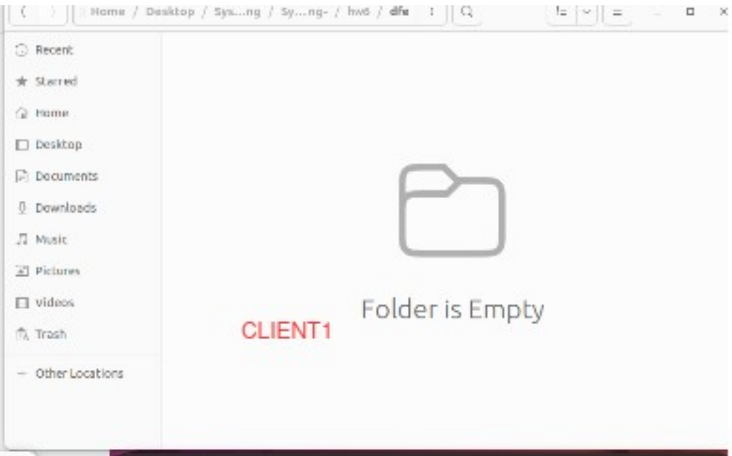
The type of synchronization I was trying to do was based on entering an if condition when there was a difference, after entering this condition, I tried trying to copy each file as it processes until it exited the while loop, this way there would be no need to use a for loop more than once, what will continue to write to the client when the first string continue value comes? If the time coincides with the end of the for loop, this time it would send quit and prevent it from doing busy waiting. My main idea was based on this, but since there were problems in practice, the above-mentioned solution was returned.

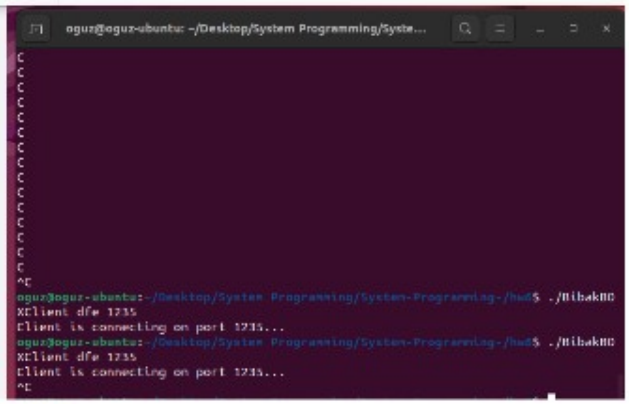
3.TEST CASES / RESULTS



test-case 2







AFTER SYNCHRONIZATION