**CSE 344**

**System Programming**

**Project Report**

Yavuz Selim İkizler

1901042617

Code Design

General Structure:

The program consists of two parts: client.c and server.c. On the server side, client connection requests are received within a while loop and added to pre-created threads. These threads invoke the client handler function for each client. If a client connects to the server for the first time, it calls the sync\_directories function, which downloads the server's content to the newly connected client. After that, the server continuously receives commands from that client within a while loop and sends responses.

There are two different commands. One is called for client synchronization. If the client receives the sync command, it calls the sync\_directories function to perform the necessary operations. The other command is used to modify files on the client side and synchronize them with the server. It calls the handle\_change function to perform the necessary operations. In order to ensure synchronization for each received command, mutex lock and unlock calls are placed at the beginning and end of the respective functions. This allows for synchronization between multiple clients.

On the client side, the program is initially executed by sending the sync command to the server. Then, within a while loop, the client checks if there are any changes in its its folder. If changes are detected, the client sends the change command to the server. If no changes are found, the synchronization command is sent again. After that, the client is put to sleep for a certain period (10 second) using the sleep call. Then, the same process is repeated.

Signal handlers are available for both the client and the server.

Some global struct variables are used for operations.

typedef struct{

    char file\_name[MAX\_FILENAME\_LENGTH];

    char f\_name[MAX\_FILENAME\_LENGTH];

    char status[MAX\_FILENAME\_LENGTH];

}file\_info;

The file\_info struct stores the properties of each modified file sent from the client. It contains the following fields:

file\_name: This field holds the path of the file.

f\_name: This field stores the name of the file without the path.

status: This field indicates the action to be taken for the file. It holds information on whether the file will be modified, added, or removed.

typedef struct {

    char client\_directory[256];

    char command[256];

    file\_info send\_file[BUFFER\_SIZE];

    int file\_number;

} Command;

The Command struct is used for each command sent by the client. The data from the client arrives in this data type. It includes the following fields:

client\_directory: This field is a char array that holds the directory name of the client sending the command.

command\_variable: This field stores the type of command, either "sync" or "change".

file\_info: This field is an array of file\_info structs. It holds the information about the files to be modified. Each element of the array represents a file and its associated information.

file\_number: This field holds the count of the files sent by the client.

typedef struct {

    char source[MAX\_FILENAME\_LENGTH];

    char destination[MAX\_FILENAME\_LENGTH];

    int source\_descripter;

    int destination\_descripter;

} FileCopyInfo;

FileCopyInfo stores the file paths and descriptors of the files to be copied.

Server.c:

The server.c main function is responsible for creating and managing threads that utilize the client\_handler function. Within the client\_handler function, there are handle\_sync and handle\_change functions to handle the commands. Additionally, there are helper functions such as read\_dir, file\_copy, create\_path, and create\_file.

Main:

The code declares several variables, including server\_sock, client\_sock, and port\_no, which are used to manage the server socket and client connections.max\_thread is initialized with the value passed as the second argument (argv[2]). This specifies the maximum number of threads to be used in the thread pool.The program checks the number of command-line arguments (argc). If it's less than 4, an error message is printed, indicating the correct usage of the program, and the program exits.A signal handler for the SIGINT signal is registered using signal(SIGINT, handle\_sigint). This allows the program to handle interruptions such as pressing Ctrl+C gracefully.The port\_no variable is initialized with the value passed as the third argument (argv[3]).The program checks if the specified directory (argv[1]) exists. If it does not exist, the code creates the directory using mkdir with the appropriate permissions (0700).

A server socket is created using socket(AF\_INET, SOCK\_STREAM, 0).

The server address structure (server\_addr) is initialized and bound to the server socket using bind.The server socket starts listening for incoming connections using listen(server\_sock, 5).

Another SIGINT signal handler is registered, likely to handle interruption during the server loop.A mutex (commandMutex) is initialized using pthread\_mutex\_init to synchronize access to shared resources.The program enters a while loop that continuously accepts client connections using accept. For each accepted connection, a new thread is created to handle the client request.A ThreadArgs struct is allocated and populated with the client socket and the directory name (argv[1]) to be passed to the thread function.

The new thread is created using pthread\_create, passing the thread ID, thread attributes, the thread function (client\_handler), and the ThreadArgs struct as arguments.

If the thread creation fails, an error message is printed, and the error should be handled appropriately.The j variable is incremented to keep track of the number of threads created.

After the thread creation loop, another loop for (int i = 0; i < j; i++) is used to join all the threads using pthread\_join. This ensures that the main program waits for all threads to finish before exiting.Some cleanup is performed, including freeing dynamically allocated memory, destroying the mutex, and returning 0 to indicate successful execution.

Client\_handler:

   struct ThreadArgs\* threadArgs = (struct ThreadArgs\*) args;

      int sock = threadArgs->client\_sock;

    char\* directory = threadArgs->arg;

When sending a thread, the structure taken as a parameter is assigned. The "sock" parameter assigns the client socket number, and the "directory" parameter assigns the server directory.

    memset(buffer, 0, sizeof(buffer));

     read(sock, buffer, sizeof(buffer) - 1);

    printf("The client connected with id: %s\n", buffer);

    snprintf(buffer, sizeof(buffer), "%s", directory);

It retrieves the ID of the connected client and writes it, and as feedback, it sends the server directory.

     while(1){

        Command client\_data;

        // Lock the mutex before reading commands

      ssize\_t bytes\_received = recv(sock, &client\_data, sizeof(Command), 0);

    if (bytes\_received < 0) {

        fprintf(stderr, "Error occurred while receiving data.\n");

       // exit(1);

    }

It continuously receives commands from the client and assigns the command to the Command structure.

// printf("%s",client\_data.client\_directory);

    if(strncmp(client\_data.command, "SYNC", 4) == 0){

        handle\_sync(directory,client\_data.client\_directory);

        snprintf(buffer, sizeof(buffer), "%s", "update");

       write(sock, buffer, strlen(buffer) + 1);

    }

    if(strncmp(client\_data.command, "change", 6) == 0){

       // printf("\n %s \n",client\_data.send\_file[0].file\_name);

        strcpy(change\_file\_list[list\_size],client\_data.client\_directory);

        list\_size++;

        handle\_change(client\_data.send\_file,client\_data.file\_number,directory);

        snprintf(buffer, sizeof(buffer), "%s", "changed");

       write(sock, buffer, strlen(buffer) + 1);

    }

It checks the commands here, calls the appropriate functions, and after the functions finish the operation, it sends the appropriate feedback.Sends the functions appropirate variables includes file list of the changed files,size of the list,the source directory.

    // Close the client socket

    close(sock);

    pthread\_exit(NULL);

It frees the finished sockets and threads.

Handle\_sync:

 pthread\_mutex\_lock(&commandMutex);

    DIR\* dir;

    DIR \* dir2;

    dir = opendir(sourceDir);

    dir2=opendir(destDir);

    if (dir == NULL) {

        perror("Error opening source directory");

       // pthread\_exit(NULL);

    }

    buffer = ( FileCopyInfo\*)malloc(MAX\_BUFFER\_SIZE \* sizeof( FileCopyInfo));

It opens the directory of the files to be copied and sends them to the read\_dir function to read the files.It allocates the buffer to hold the files.

read\_dir(dir,dir2,sourceDir,destDir);

    file\_copy();

    free(buffer);

It calls the read\_dir function to read the files, and then calls the copy\_file function to write the read files to the client's directory.It frees the buffer after the operation is completed.

Read\_dir:

 int j=0;

    while ((entry = readdir(dir)) != NULL) {

        // Skip "." and ".." directories

        if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0) {

            continue;

        }

        // Create source and destination file paths

        snprintf(sourcePath, MAX\_FILENAME\_LENGTH, "%s/%s", sourceDir, entry->d\_name);

        snprintf(destPath, MAX\_FILENAME\_LENGTH, "%s/%s", destDir, entry->d\_name);

It reads the files one by one using the dirent structure, and combines the paths for the source and destination directories using snprintf.

if(entry->d\_type == DT\_DIR){

             struct stat st;

    if (stat(destPath, &st) == 0) {

        if (S\_ISDIR(st.st\_mode)) {

          //  printf("Directory already exists.\n");

           // return ;

    else{

        // Create the directory if it doesn't exist

    if (mkdir(destPath, 0700) == 0) {

      //  printf("Directory created successfully.\n");

    } else {

        printf("Failed to create the directory.\n");

    }

    }

    dir2\_second=opendir(destPath);

    dir2=opendir(sourcePath);

    read\_dir(dir2,dir2\_second,sourcePath,destPath);

    closedir(dir2);

    closedir(dir2\_second);

If the read file is a directory, it checks if the corresponding destination directory exists in the client's directory. If it doesn't exist, it creates a subdirectory in the client's directory. It opens the subdirectory to read files in it and recursively calls itself to read files in that subdirectory as well.

  // Copy file information into the buffer

        strcpy(buffer[bufferIndex].source, sourcePath);

        strcpy(buffer[bufferIndex].destination, destPath);

        // Open source file for reading

    buffer[bufferIndex].source\_descripter = open(buffer[bufferIndex].source, O\_RDONLY);

    if (buffer[bufferIndex].source\_descripter == -1) {

        perror("Error opening source file");

        return ;

    }

    // Create or truncate destination file for writing

    buffer[bufferIndex].destination\_descripter= open(buffer[bufferIndex].destination, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

    if (buffer[bufferIndex].destination\_descripter== -1) {

        perror("Error opening destination file");

        close(buffer[bufferIndex].source\_descripter);

        return ;

    }

        bufferIndex++;

It saves the source and destination paths to the buffer. It also opens as a read modde for source (server) for destination(client) and saves the descriptors of the files. The buffer index is then incremented by 1.

Then it reads the client directory and compares it with the files on the server. If there is a different file in the client directory, it deletes it.

File\_copy:

It reads the files in the buffer one by one and calls the copy\_file function, passing the source and destination directories, to perform the copy operations. When all the files have been read from the buffer and the buffer index reaches 0, it exits the loop.

Copy\_file:

It writes the files read from the source to the destination using the descriptors in the buffer.

Handle\_change:

It takes the list of files to be modified as a parameter and the server directory.

It runs a for loop based on the number of files in the list. It removes the part of the path that includes the client directory and replaces it with the server directory. It calls appropriate helper methods based on whether the files in the list need to be removed, added, or modified. These file operations are performed by extracting the status of the files and comparing them.

CreatePath:

The createPath function creates a directory based on the provided string.

CreateFile:

The CreateFile function reads and writes files based on the provided string path.

Client.c:

The main function includes the commands to perform, such as change\_directory and sync\_directories. It also includes the check\_difference function to check for directory changes, and the read\_folder and read\_dir functions for file reading operations.

Main:

It defines an integer variable file\_number and initializes it to 0.It checks if the command line argument count (argc) is less than 4. If it is, it prints an error message indicating the correct usage of the program and exits with an error code.It sets up a signal handler for the SIGINT signal (interrupt signal).It converts the second command line argument (argv[2]) to an integer using atoi() and assigns it to the variable port\_no.It creates a socket using the socket() system call. If the socket creation fails, it prints an error message and exits.It uses the gethostbyname() function to get the server's host information based on the third command line argument (argv[3]). If the host information is not found (server is NULL), it prints an error message and exits.It initializes the server\_addr structure with zeros using memset() and sets the address family to AF\_INET.It copies the server address obtained from gethostbyname() to server\_addr.sin\_addr.s\_addr and sets the port number to port\_no.It attempts to establish a connection to the server using the connect() system call. If the connection fails, it prints an error message and exits.

First, it sends the IP address to connect to the server. It receives the server's directory name as a notification from the server. It checks if the client directory name received from argv exists or not. If it doesn't exist, it creates a directory with that name.

Afterward, it calls the sync\_directories function to retrieve the files from the server. Then, it allocates the before variable and sends it to the read\_folder function. This function saves the descriptors of the existing files, and before serves as a list to hold those descriptors.

Then, a while loop is initiated. Within this while loop, first, the current files in the client directory are read by calling the read\_folder function, with after being passed as a parameter, which serves as a list to hold file descriptors. Then, the check\_difference function is called, passing before and after variables. This function compares these lists, and if there is a difference, it returns 1 and the necessary parameters are passed to call the change\_server function to send the change command to the server. If there is no difference, the sync\_directories function is called to send the synchronization command. Finally, the loop is paused using sleep, and the same set of operations continues continuously.

Sync\_directories:

It creates a sync command to send to the server and initiates the synchronization process on the server side.

Read\_folder:

It opens the client directory, calls the read\_dir function, and passes it to read the directory.

Read\_dir:

It reads the files in the current directory within the loop, saves their descriptors to the FileSave list, and writes them. To handle subdirectories, it is recursively called.

Chaneg\_server:

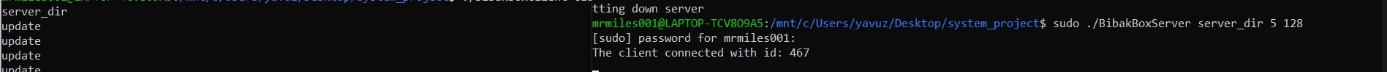
It adds the files to be modified to the appropriate locations in the command structure, creates a change command with that structure, sends it to the server, and waits for a response.

Check\_Differance:

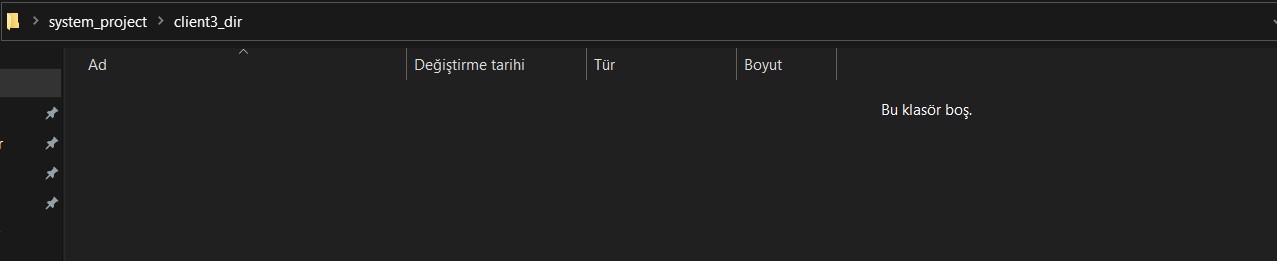
It compares the two file save lists, one being before and the other being after, to detect the differences. If there are more files in after compared to before, it takes those additional files, sets their status as "add," and adds them to the file\_info struct array. If a file is missing in after compared to before, it sets its status as "remove" and adds it to the file\_info array. If a file is different, it sets its status as "change." Finally, it closes the descriptors in before and sets before equal to after.

Test Outputs

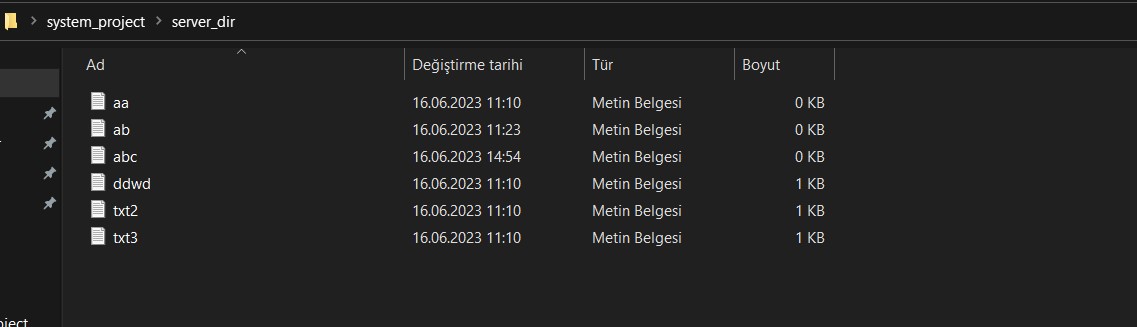
Connect:



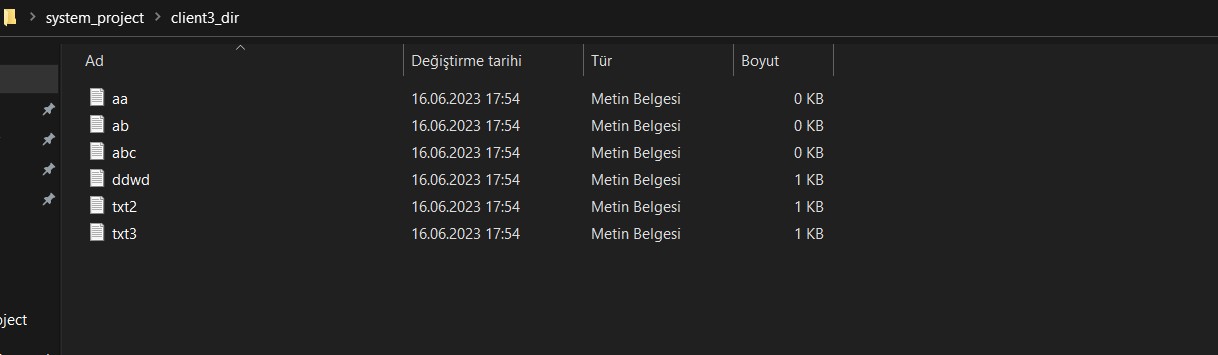
Client3\_dir(before):



Server\_dir:



Client3\_dir(after):



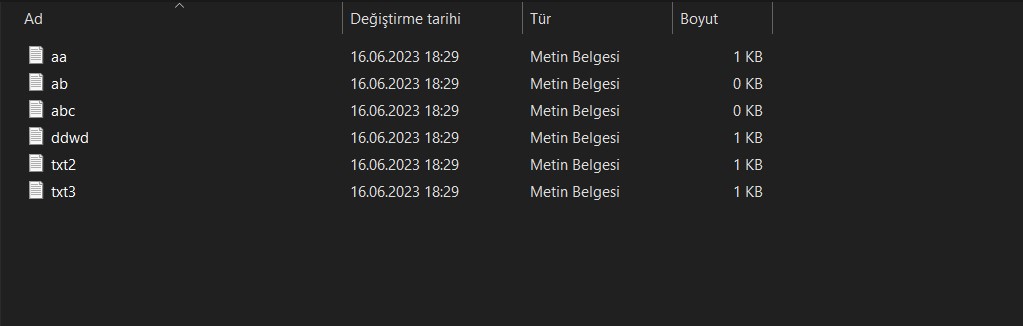
Sig Handler:

Server, Client:



Server Operations:

Server: server\_dir



Client: client1\_dir, client2\_dir, client3\_dir, client4\_dir, client5\_dir

metin, ekran görüntüsü, siyah içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, multimedya yazılımı, yazılım, grafik yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, multimedya yazılımı, grafik yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu

ekran görüntüsü, metin, multimedya yazılımı, yazılım içeren bir resim

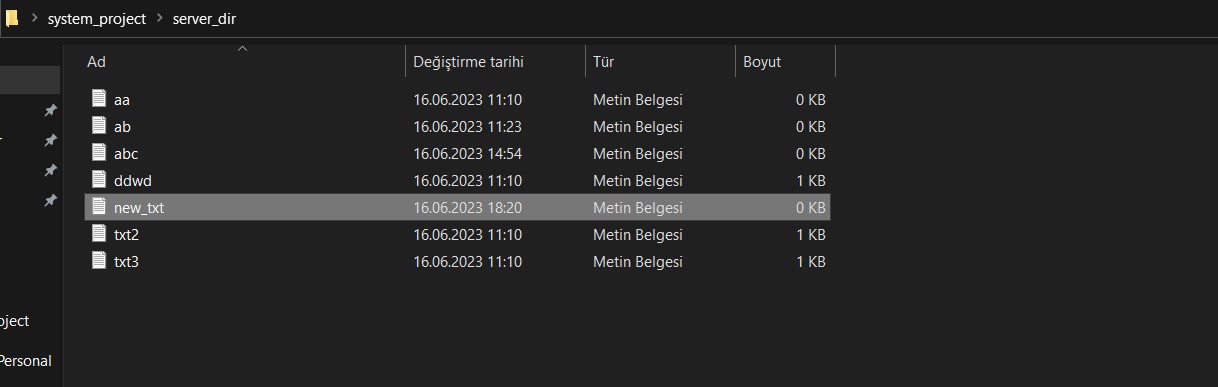
Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazılım, multimedya yazılımı içeren bir resim

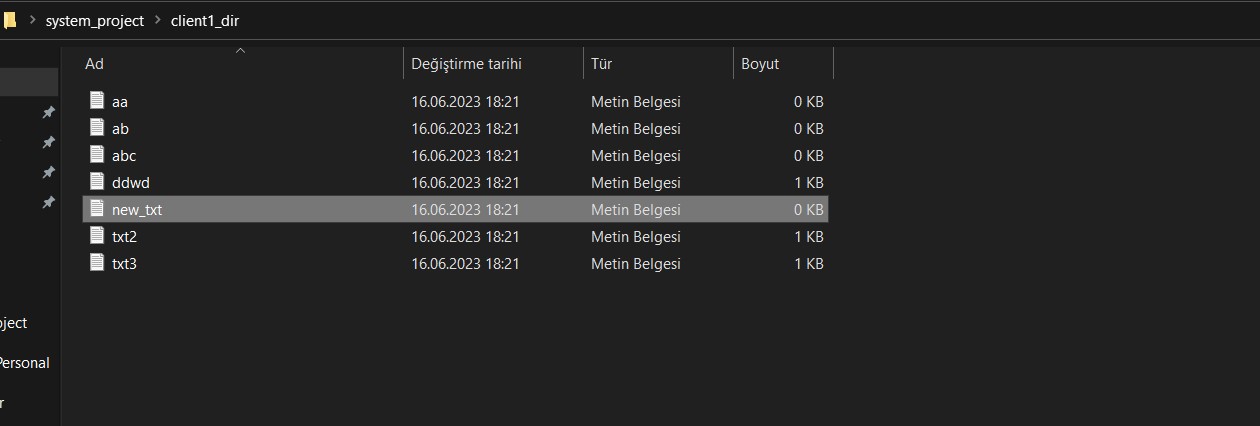
Açıklama otomatik olarak oluşturuldu

Add:

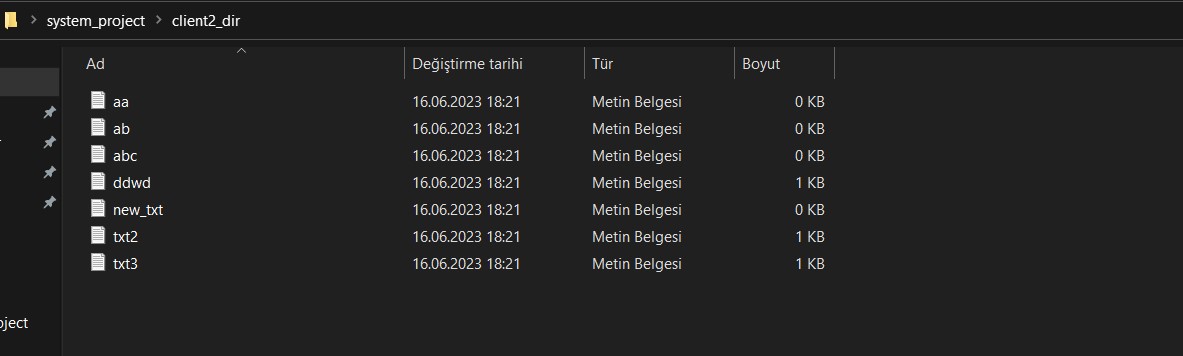
Server



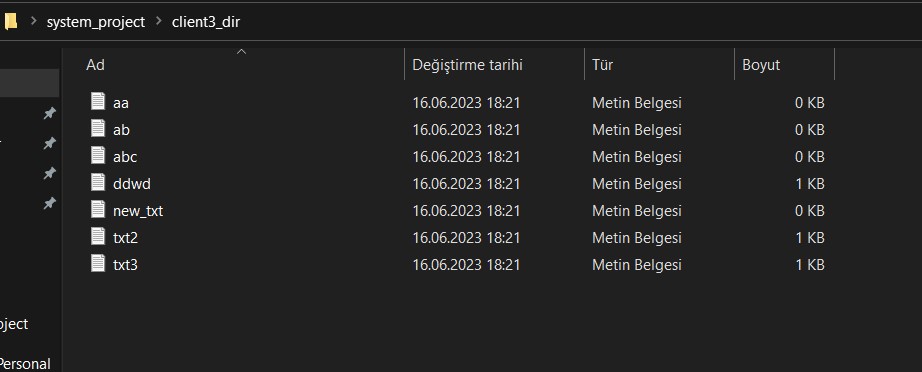
Client1



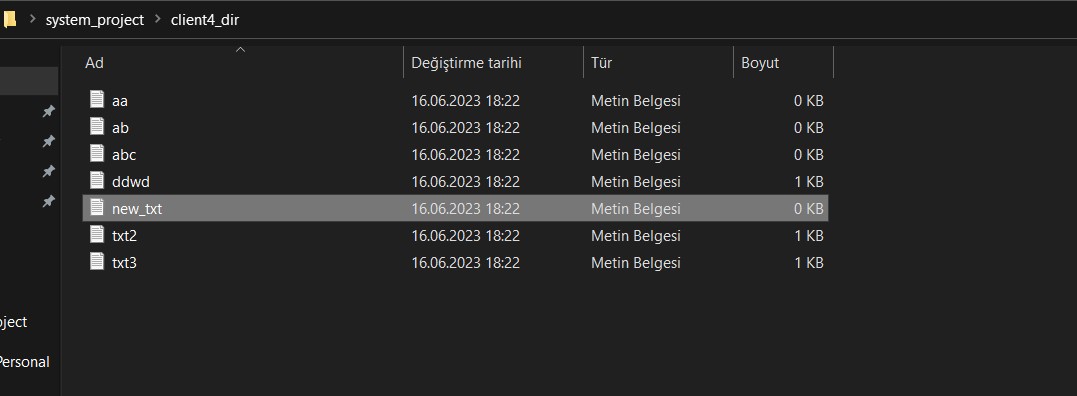
Client2



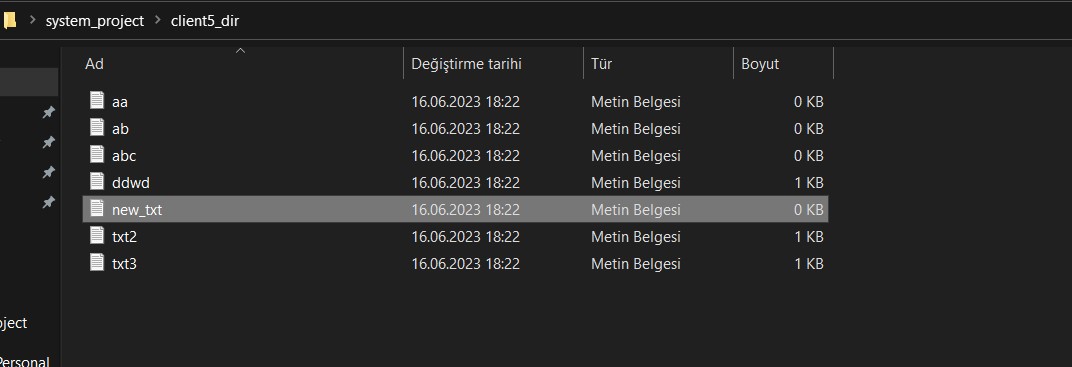
Client3



Client4



Client5

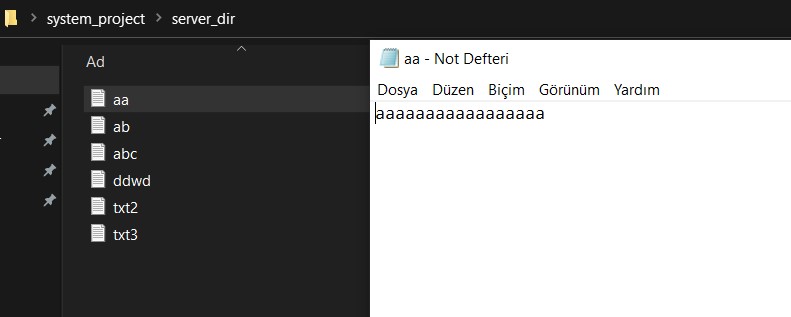


Change:

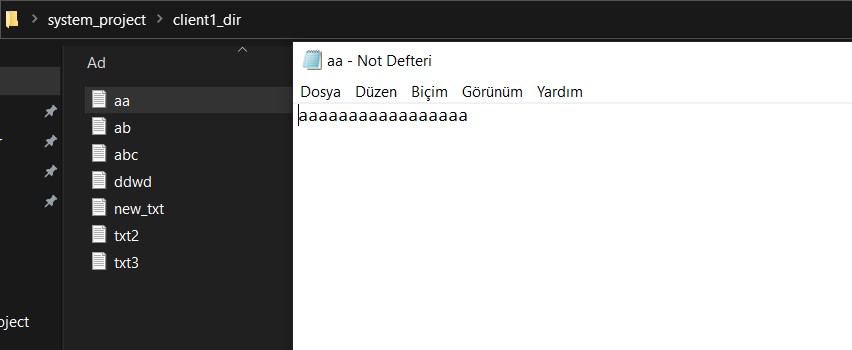
metin, beyaz, makbuz, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

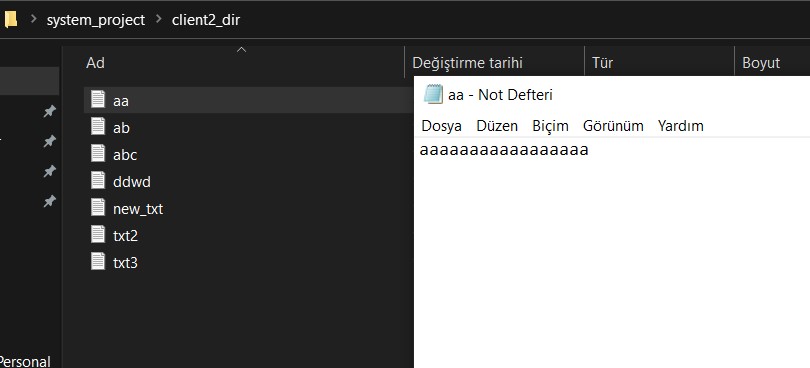
Server



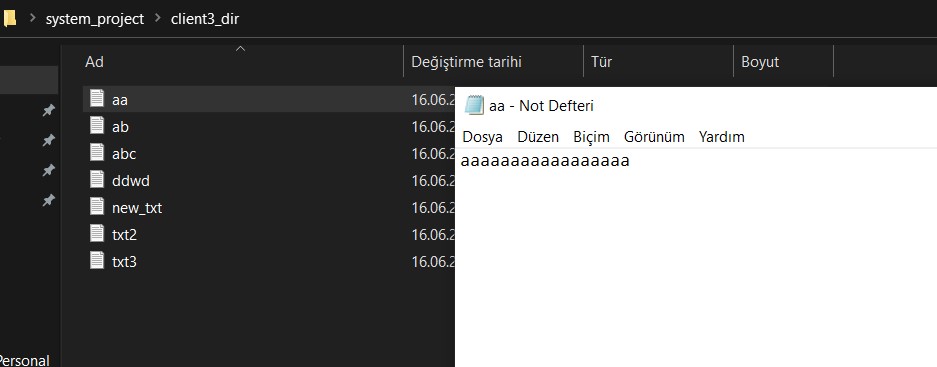
Client1



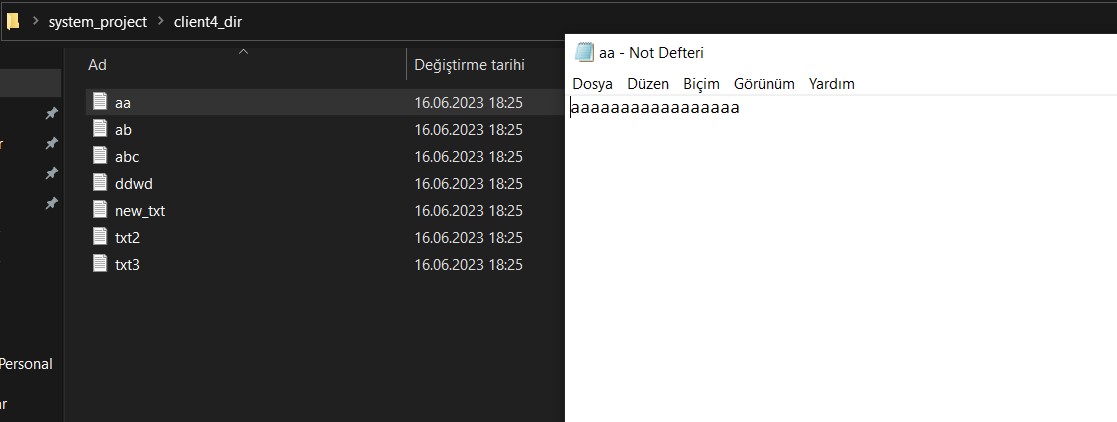
Client2



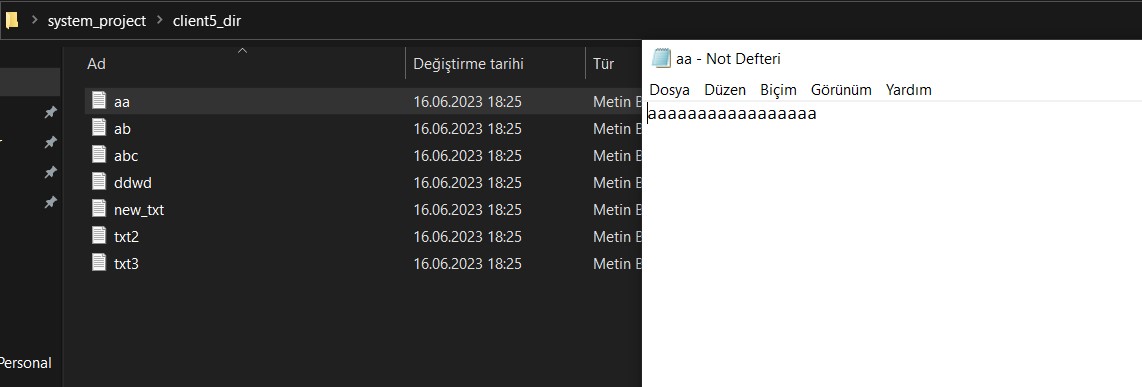
Client3



Client4



Client5



Remove:

Server

metin, ekran görüntüsü, siyah içeren bir resim

Açıklama otomatik olarak oluşturuldu

Client1

metin, ekran görüntüsü, siyah içeren bir resim

Açıklama otomatik olarak oluşturuldu

Client2

metin, multimedya yazılımı, yazılım, grafik yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu

Client3

metin, ekran görüntüsü, multimedya yazılımı, grafik yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu

Client4

ekran görüntüsü, metin, multimedya yazılımı, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

Client5

metin, ekran görüntüsü, yazılım, multimedya yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu

Client Operations:

Add:

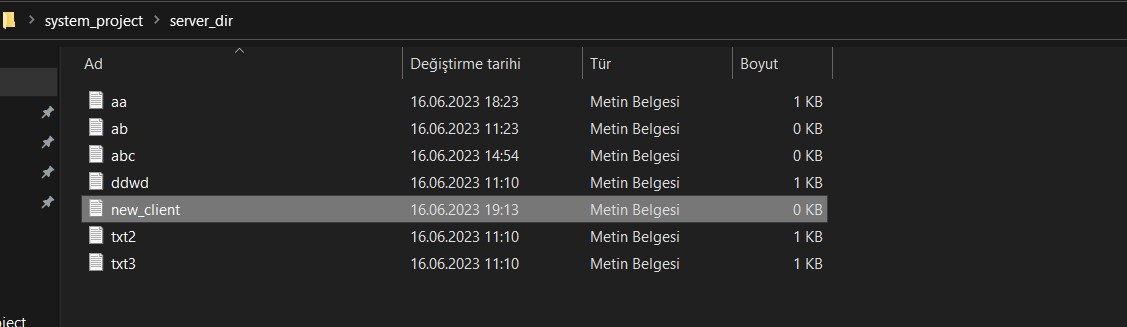
Add a file named new\_client

Client1

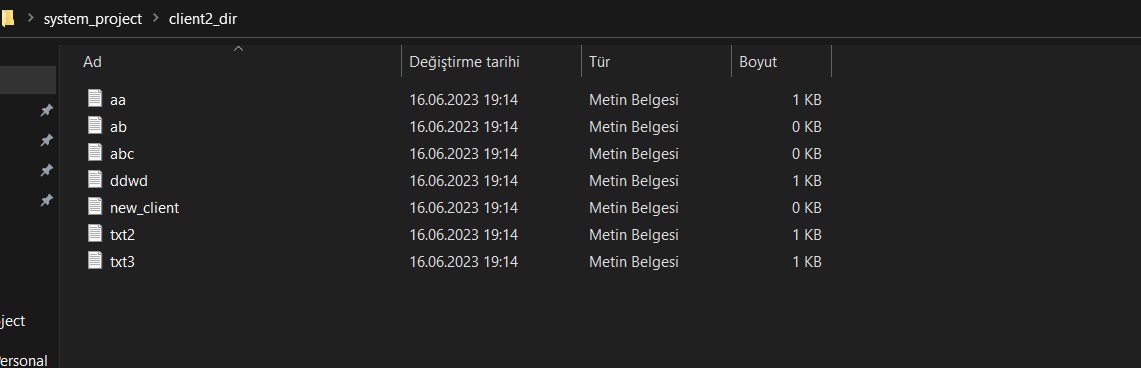
metin, ekran görüntüsü, multimedya yazılımı, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

Server



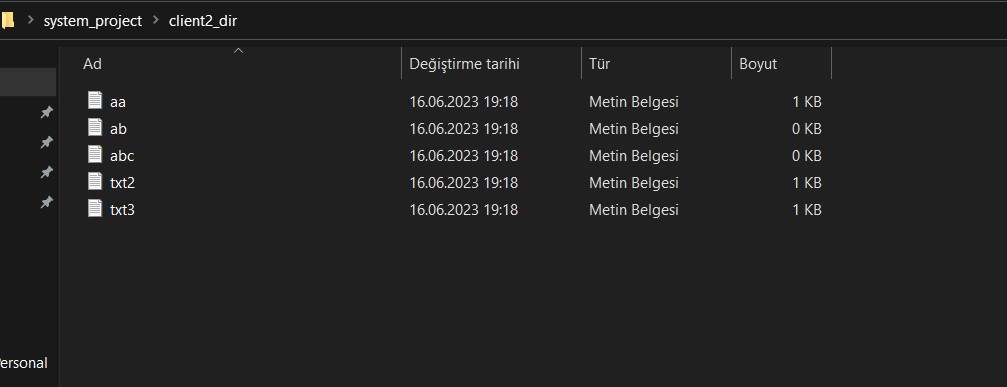
Client2



Change:

Remove:

Remove new\_clien ü,ddwd from client1

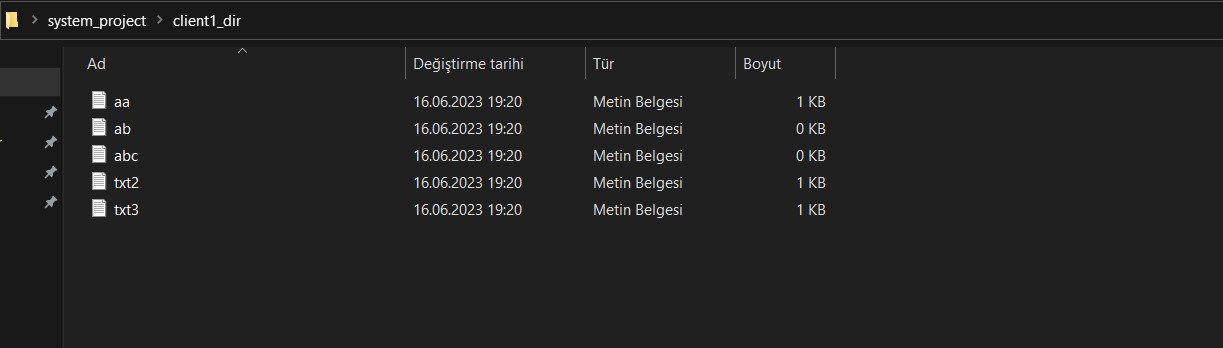
Server:

Client2

metin, ekran görüntüsü, multimedya yazılımı, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

Client1



Note:

Running example

Server: sudo ./BibakBoxServer server\_dir [client\_number] [port number]

Client: ./BibakBoxClient client\_dir [port number] [IP adress]