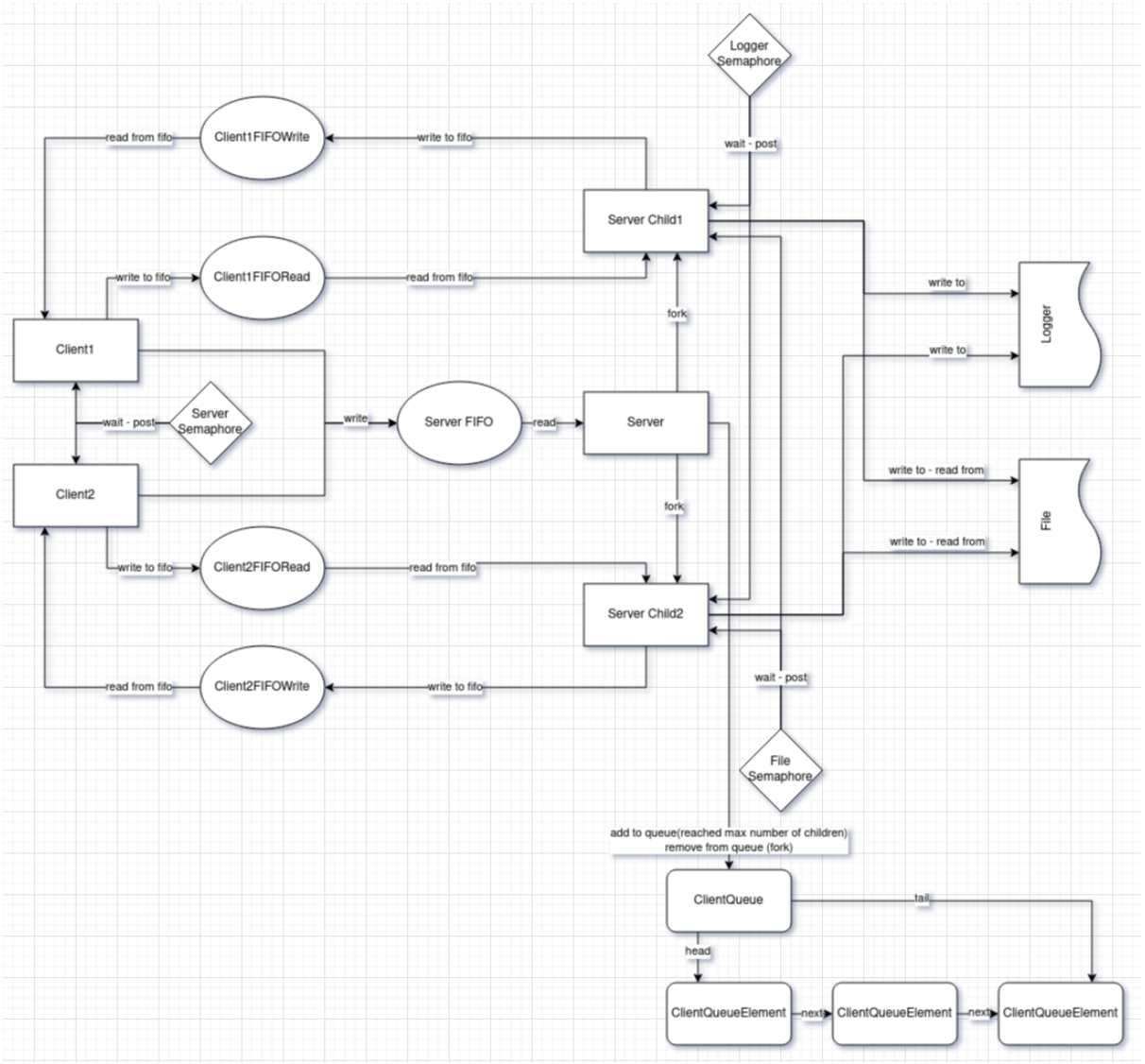
CSE344 Systems Programming

Midterm Report

Ömer Faruk Çolakel

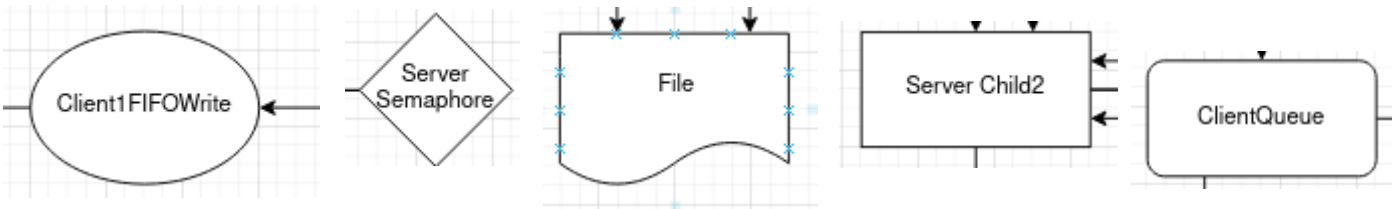
200104004043

System Architecture



I have added this PNG in the report folder, in case you can't see or read something. You can also import the drawio file in this website.

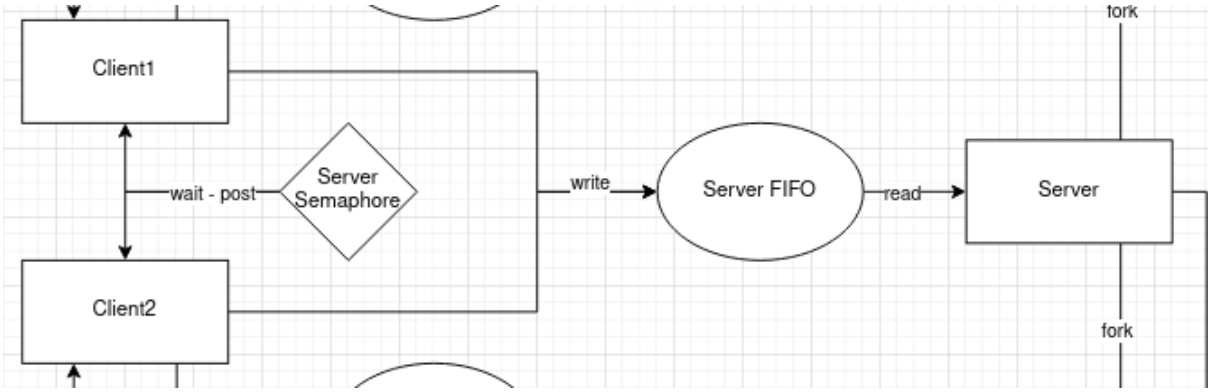
Symbols and Their Meanings



FIFO Semaphore File Process Queue

Design Decisions

Connecting to Server as Client



Server creates a fifo and a semaphore in the tmp folder using its pid. This pid is used in their names so that they are unique to the server. Every client gets that pid using command line arguments. Using the given pid, they try to open the fifo and if it fails, exits the process with failure. In case they don't fail, wait for the semaphore that the server created to be available. The second that it is available, writes a Request object to the fifo. The server is blocked until there is something to read from the fifo. The purpose of the semaphore is to prevent clients from writing at the same time and potentially corrupt the Request objects. These requests have the pid of the client that is trying to connect. There is more information about the Request struct in the next section.

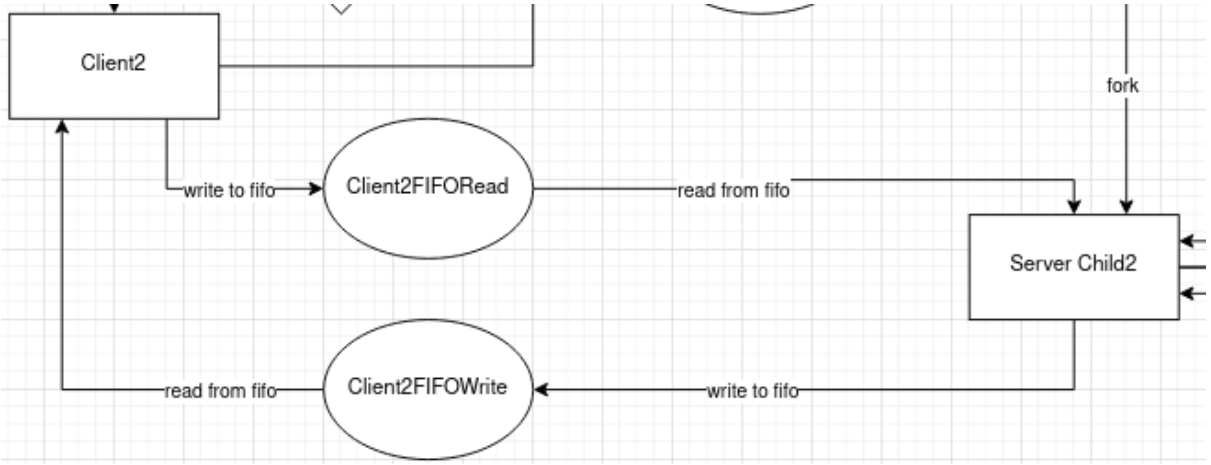
Request Struct

```
enum RequestType // type of reques
{
    ERROR = -1,
    TRY_CONNECT = 1,
    CONNECT = 2,
    HELP = 3,
    LIST = 4,
    READFROM = 5,
    WRITETO = 6,
    UPLOAD = 7,
    DOWNLOAD = 8,
    ARCHSERVER = 9,
    KILLSERVER = 10,
    QUIT = 11
};
typedef struct Request // request
{
    enum RequestType requestType;
    char request[256];
} Request;
```

This struct is the main way of sharing data among processes. It has a fixed size which is 260 bytes. There is an enum both in the client and server called RequestType that takes 4 bytes and the rest is a char array with the size of 256. When a client tries to connect, it first creates a Request object and sets the type as TRY_CONNECT or CONNECT and writes its pid to the request array. The server reads 260 bytes and writes it to the request object that it has created. The server or clients can use this struct to send other types of request as well. Once the client is connected to a child of the server, it sends request objects to the ClientFIFO{clientpid}Read. The child reads the request and processes it and it may return an error response as well. These error typed responses are handled in the client.

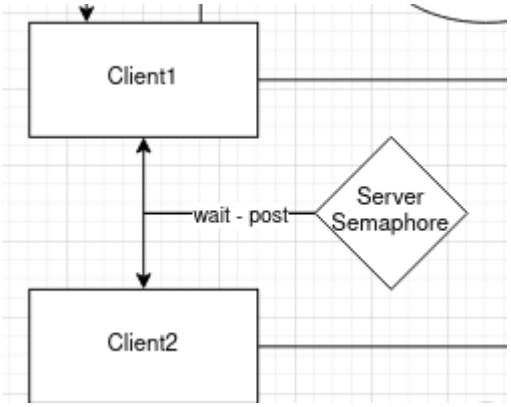
FIFOs

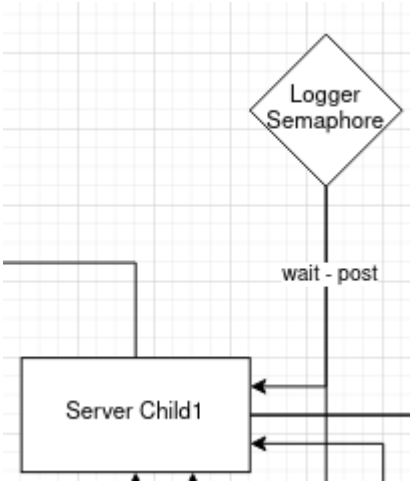
As you can see from the architecture, every client has two fifos. ClientFIFO{clientpid}Read is the fifos that the children of the server reads and ClientFIFO{clientpid}Write is the one that they read. Since there is an incoming and an outgoing lane for data transfer and no processes other than the child and the client can read or write from/to these fifos, I didn't use any semaphores for them. These fifos are unlinked and deleted during the termination process of the child and the client.

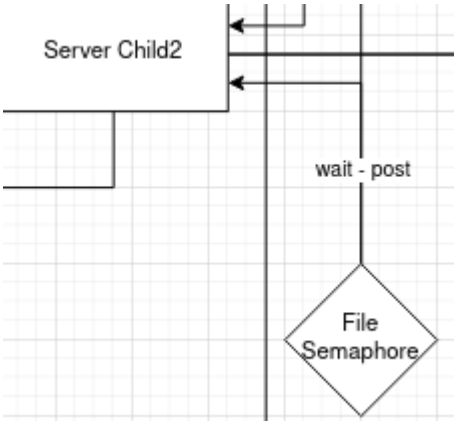


Semaphores

There are three types of semaphores used: ServerSemaphore, LoggerSemaphore and FileIOSemaphore.
Server semaphore is created by the server and used by the clients. Every client waits for this semaphore to be available and posts after they are done.
Logger semaphore is used so the children of the server don't write to the logger at the same time. There is a function called writeToLogger that writes to the logger file. This function forks another process so that it can write to the logger. Posts the semaphore after using it.
FileIOSemaphore prevents the children from writing/reading at the same time. Semaphore is waited before every request that requires files. After the request is done, it is posted.







Logger

When the server is started, a new client is connected, a client is disconnected, server killed, a request comes and a request is finished; the server and its children write everything to the logger. The information that was produced by these requests are not being written to the logger (for example, if the client requests to read from a file; the request itself, pid of the client and how the request handled are written to the logger but not the line that was sent to the client.).

Signal Handling

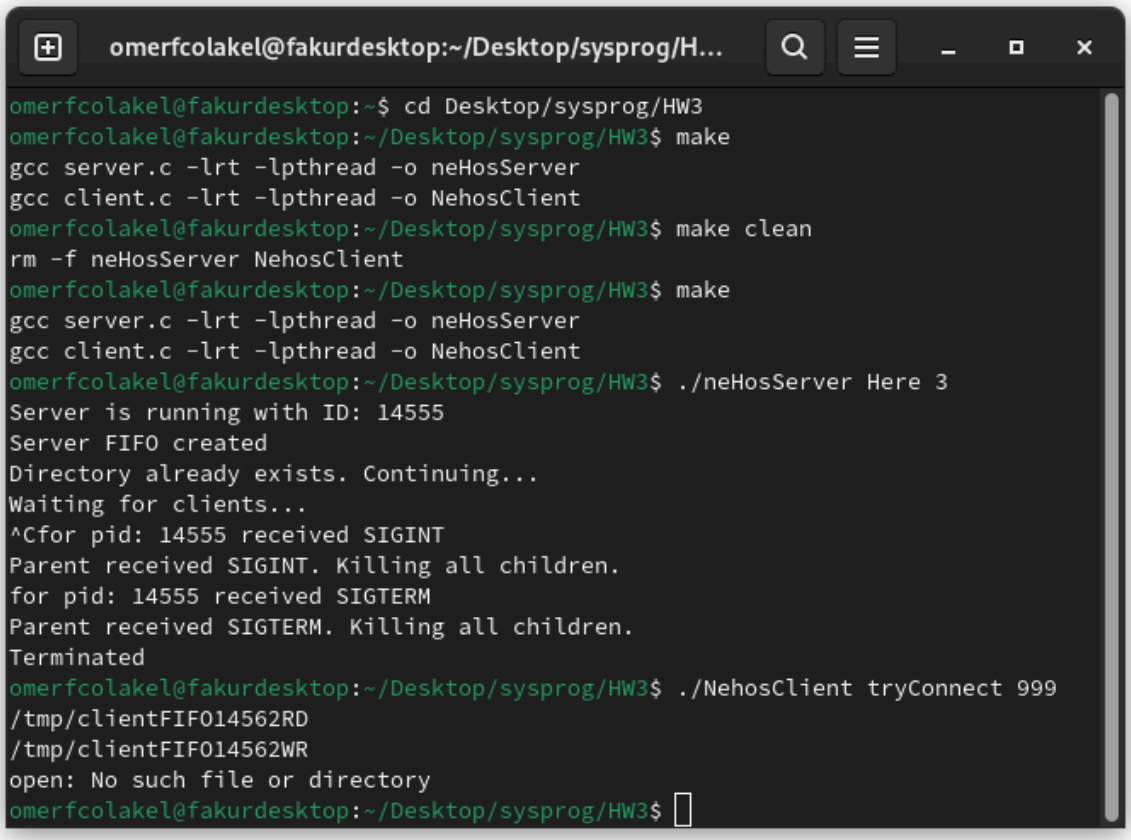
The SIGCHLD, SIGTERM and SIGINT signals are handled by the functions named {signal name}Handler.
The handling of the SIGTERM and SIGINT is almost the same: the server and its children are killed. While they are dying, they send a kill signal to the clients as well.
Handling the SIGCHLD is necessary because the server has to know if one of the children is dead or not to handle the client queue. It checks if there is a client that waits in the queue to connect. If there is none, it just decreases the number of clients. If there is, it forks to create a child to handle the client's requests.

Client Queue

The clients that tried to connect using "connect" and not "tryConnect" but failed due to reaching the maximum number of clients are added to this queue. When the server gets a SIGCHLD signal, the first client in the queue receives a request object that it is finally its turn. The client is removed from the queue.

Continue on the next page =>

How to Compile and Run



I have a makefile to compile and clean both the server and client code. If you want to compile both of them the source codes should be in the same folder. You don't have to put them in the same folder for them to work. I designed them to work from different folders. To clean both of the compiled files, they have to be in the same folder as well. As you can see from the example, the server runs by typing "./neHosServer {folder name} {positive integer number}". To run the client, you can type "./NehosClient {tryConnect/Connect} {server pid}".

Continue on the next page =>

Functions, Their Purposes and How They Work

1) It prints the given message with the given length to the standard output. (both)

```
int printMessage(char *message, int length);
```

```
void printUsage();
```

2) This prints how to run the program. (./NehosClient connectionType pid) (both)

```
int createLoggerFile(char *dirName);
```

3) If there are no logger files, it creates and opens one and returns the file's descriptor. If there is, it opens it in append mode and returns the file's descriptor. (server only)

4)
```
char *createWorkingDirectory(char *dirName);
```
Creates a directory with the given name to store files. (server)

5)
```
sem_t *createSemaphore(int serverID, char *semaphoreName);
```
Used to create semaphores with a given name and server's id. Returns the semaphore.(server)

6)
```
void handleClientRequest(int serverFIFORead, int logFile, sem_t *serverSemaphore, sem_t *serverFileIOSemaphore, sem_t *loggerSemaphore);
```
Handles a client's request to connect to the server. Requires server's fifo to read from, logger's descriptor, semaphore for server's fifo, semaphore for file IO and semaphore for logger. Reads a request from server fifo if the semaphore is available. This request is checked if it is TRY_CONNECT or CONNECT. If it is TRY_CONNECT, it checks the number of working children. If it is less than the max number of clients, it forks to create a child that will handle the client. If the server is full, send an ERROR response (with the format of Request object). If the client wants to connect via CONNECT and if the server is not full, it works the same way as the TRY_CONNECT. If the server is full, it creates a ClientQueueElement and enqueues it to the ClientQueue. (server)

```
Request request;
int bytesRead = read(serverFIFORead, &request, sizeof(Request));
```

7) `int handleClient(int clientPID, sem_t *serverFileIOSemaphore, int logFile, sem_t *loggerSemaphore, pid_t parentID);`

This function works in a child process and its only purpose is to handle client's requests in an endless loop. Reads a request from the user and checks its type. Writes the taken order to the standard output and to the logger. Waits for the semaphores if necessary. Calls the function to handle that order and gives its parameters. After the function is done, it posts the semaphores that are used. (server)

8) `int handleHelp(Request request, int clientFIFORead, char *clientFIFOWR);`

Only the handleClient can call this function. Writes the response to clientFIFOWrite as a request object.If the request object's request array is empty sends this message:

`"\nAvailable commands are:\nhelp, list, readF, writeT, upload, download, archServer, killServer, quit\n\n"`

If the client wants help with a command then it sends a message about that command. For example:

`readF <filename> <n>: Reads n th line of the file\n`

`void handleHelp(char *clientFIFOReadName, char *clientFIFOWriteName, char *command);`

The previous one was on the server. The same function is present on the client as well. It sends the help request to the server. Reads the response from the fifo and writes it to the standard output.

`int handleList(Request request, int clientFIFORead, char *clientFIFOWR);`

9) Handles the list command.
In the server, it can only be called by the handleClient function. FileIOSemaphore is waited before this function and posted after. It takes the names of the files in the server's working directory and writes them to a buffer. Buffer's size increases if it's full. The buffer size is divided into 256 byte packets. Packet count is sent to the client and then the packets themselves. These packets have a part of the buffer inside them.

`void handleList(char *clientFIFOReadName, char *clientFIFOWriteName);`

On the client side, this function sends a list request to the server first. Then waits for the number of packets to be expected. In a loop, reads the fifo and saves the read bytes to request objects. The packets are printed to the standard output before returning.

10) Handles reading from a file.

```
int handleReadFrom(Request request, int clientFIFORead, char *clientFIFOWR);
```

On the server side, it tries to open the file and if it can't, writes an error response to the fifo. If the line number is given, it tries to find that line. If it can, it writes that line to a buffer. Then the buffer is sent to the client the same way that the handleList packets work. If it can't find the line, it sends an error response. If the line number is absent, it writes the whole file to a buffer and sends it to the client.

```
void handleReadF(char *clientFIFOReadName, char *clientFIFOWriteName, char *filename, char *lineNumber);
```

On the client side, first it checks if the user has given enough parameters. If he/she did, it makes a request to the server. Waits for a response from the server and if it is an error, writes the error to the stdout. If there are no errors, it takes the packet count and takes request objects as much as the packet count. Writes those packets to stdout.

11) Handles writing to a file.

```
void handleWriteT(char *clientFIFOReadName, char *clientFIFOWriteName, char *filename, char *text, char *lineNumber);
```

On the client side, it checks if there is a file name and text. Those two are mandatory, so in their absence, the function doesn't make any requests to the server. It also checks the line number but only to write it to the request object. It can be null.

```
int handleWriteTo(Request request, int clientFIFORead, char *clientFIFOWR);
```

On the server side, first tries to open the file. Opening the file may fail since there is no guarantee that there is a file with the given name. If it fails, it sends an error response to the client. If there is a line number, it reads from the file until that line and then writes to the file. If the file ends until the given number returns an error response. In the case of no line number, it just appends the file with the given text.

12) Handles uploading a file to the server.

```
int handleUpload(Request request, int clientFIFORead, char *clientFIFOWR);
```

On the server side, it first checks if there is a file with the given name. If there is, it asks the user if he/she wants to overwrite it. If the response from the client side is "y", overwrites the file. If the response is "n", it cancels the task. If there are no files with that name, it creates one and writes the incoming bytes into it. Instead of using packets, this time the client sends the size of the file as an integer and then the file itself. So the server side first reads an integer and then the file.

```
void handleUpload(char *clientFIFOReadName, char *clientFIFOWriteName, char *filename);
```

On the client side, first makes the request and then waits for an error response. If there is, it handles the response as I stated in the previous paragraph. If there are no errors or after they are handled, it writes the size of the file and then writes the file to the fifo. Lastly, there is another response from the server stating if it has been completed successfully or not.

13) Handles downloading a file from the server to the client.

```
int handleDownload(Request request, int clientFIFORead, char *clientFIFOWR);
```

On the server side, it checks the folder if there is a file with the given name. Writes an error response if there is not. If the file is found, it writes the size of it to the fifo and the file itself.

```
void handleDownload(char *clientFIFOReadName, char *clientFIFOWriteName, char *filename);
```

On the client side, it first checks if there is a file with the given name at the client's folder. If there is, it asks to overwrite it or not. There will be no requests if the user chooses not to overwrite. The request is sent after the user agrees to overwrite or there is no file with that name. If the response from the server is not error then checks the request array to get the size of the file. Then reads those bytes from the fifo and saves the file.

14) Handles archiving the files.

```
int handleArchServer(Request request, int clientFIFORead, char *clientFIFOWR)
```

On the server side, first it gets the filename and checks if it is null. Then forks to create a child that utilizes exec command to create a tar file in the tmp folder of the system. This file is then sent to the client after its size is written to the fifo. If there was a problem, an error response is sent instead.

```
void handleArchServer(char *clientFIFOReadName, char *clientFIFOWriteName, char *filename);
```

On the client side, it checks if the filename is null or not. Filename can't be null and if it's null, there are no requests. After making the request, the client waits for a response. If it is an error response, the function writes the request array to the standard output and returns. If it's an archServer request, then creates a fileSize integer and assigns the value in the response. Then reads the fifo with the given fileSize and saves the file.

15)

```
void writeToLogger(int logFile, char *message, sem_t *loggerSemaphore);
```

Writes the given string to the logger. Only used on the server side.

16)

```
void killClient();
```

Sends SIGTERM to the client from the child process. Only used on the server. The parent process can't send a signal because the client PID is -1 inside it and it is checked if the PID is positive in the function.

17)

```
int handleExit();
```

It is used on both sides. Cleans the fifos and semaphores.

18)
```
void sigTermHandler(int signo);
```
Handles SIGTERM and it is used on both sides.
On the server side; if the parent gets the SIGTERM, kills all children and waits for them. After that, it calls the handleExit function. If a child gets this signal, they kill the client and exit.
On the client side, it prints that the process got the SIGTERM signal and cleans the fifos.

19)
```
void sigIntHandler(int signo);
```
Handles SIGINT and it is used on both sides.
On the server side; it works the same way as sigTermHandler.
On the client side, sends a quit request to the server, cleans the fifos and exits.

20)
```
void sigChldHandler(int signo);
```
Handles SIGCHLD on the server. It checks if there is a client waiting in the queue. If there is none, it just decreases the number of clients (global variable). If there is, it removes it from the queue, sends a response to awaken the client and forks and calls the handleClient function.

21)
```
void initClientQueue();
```
Initialized the global ClientQueue.

22)
```
void initClientQueueElement(ClientQueueElement *element, int clientPID);
```
Initializes a ClientQueueElement with the given client pid. Sets the next as NULL.

23)
```
void enqueueClient(int clientPID);
int dequeueClient();
void printQueue();
```
Adds an element to the end of the queue with the given pid.
Removes the first element in the queue.
Prints the queue.
Every function that is about ClientQueueu and its elements are on the server side.

24)
```
int checkArguments(int argc, char *argv[]);
```
It is used on the client side to check the command line arguments at the start of the program. Returns 0 if there are no problems and -1 if there is one. Prints the problem.

25)
```
sem_t *initializeSemaphore(int serverID);
```
The client needs the server semaphore to make a connection request. This function opens the semaphore and returns it.

26) `int createClientFIFOs();`

Every client needs to make two fifos in order to communicate with the server after making a connection. This function creates those fifos.

27) `int makeConnectionRequest(char *command, int serverID, sem_t *sem);`

Makes a connection request to the server via writing a request to the server fifo. Waits for the semaphore and posts it after using it. The request type is either TRY_CONNECT or CONNECT and it is determined by the command. If the command is not one of them exits the program with failure. The request also has the client's pid. If the connection type is TRY_CONNECT, the function checks the response type and acts accordingly: Either exits immediately or returns with success. If the connection type is CONNECT then waits for a response. If the response is not an error, it means the client is connected. If it is an error, it waits for a response in the client fifo. So it waits until the connection is successful.

28) `char **tokenizeBuffer(char *buffer);`

On the client side, the user enters input from the terminal. These inputs are parsed and put into a 2D char array (string array) and returned. The last string is always NULL.

29) `void handleKillServer(char *clientFIFOReadName, char *clientFIFOWriteName);`

Makes a kill server request to the server.

I didn't implement a handleKillServer function on the server side and a handleQuit function on the client side. But they are there and working. I just didn't put the code into those functions.

30) Client Side Main Function
    - Checks the arguments
    - Initializes server semaphore
    - Sets the signal handlers
    - Creates client fifos
    - Makes connection request
    - In an endless loop, waits for the user to enter an input, parses it and calls the needed functions and gives its parameters.
    - Cleans the fifos at the exit

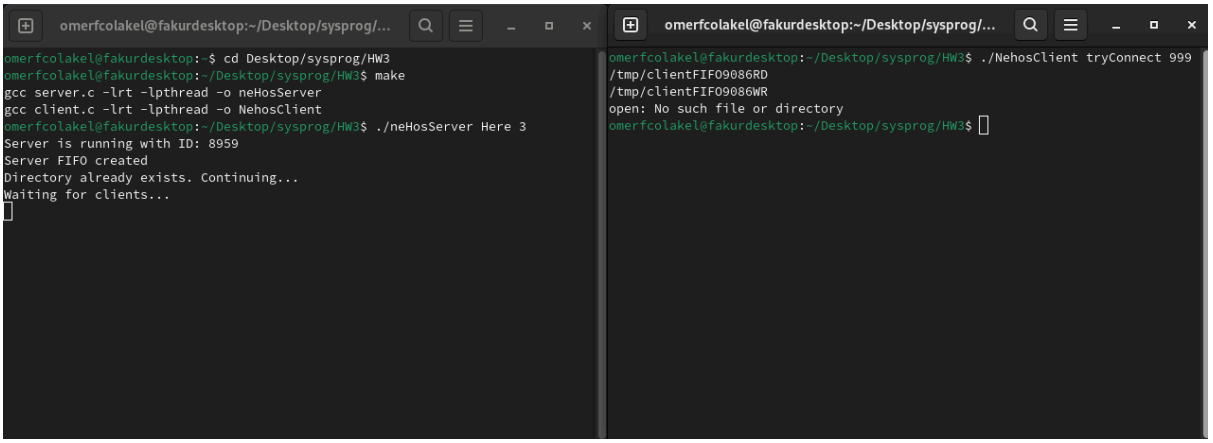
31) Server Side Main Function
    - Checks the arguments
    - Sets the signal handlers
    - Initializes global variables: clientPID, globalNumOfClients, globalMaxClients, clientQueue
    - Creates the server fifo
    - Creates the working directory (“Here” folder for example)
    - Creates or opens the logger file
    - Initializes the semaphores
    - Calls the handleClientRequest function in an endless loop
    - Cleans before the program ends

Continue on the next page =>

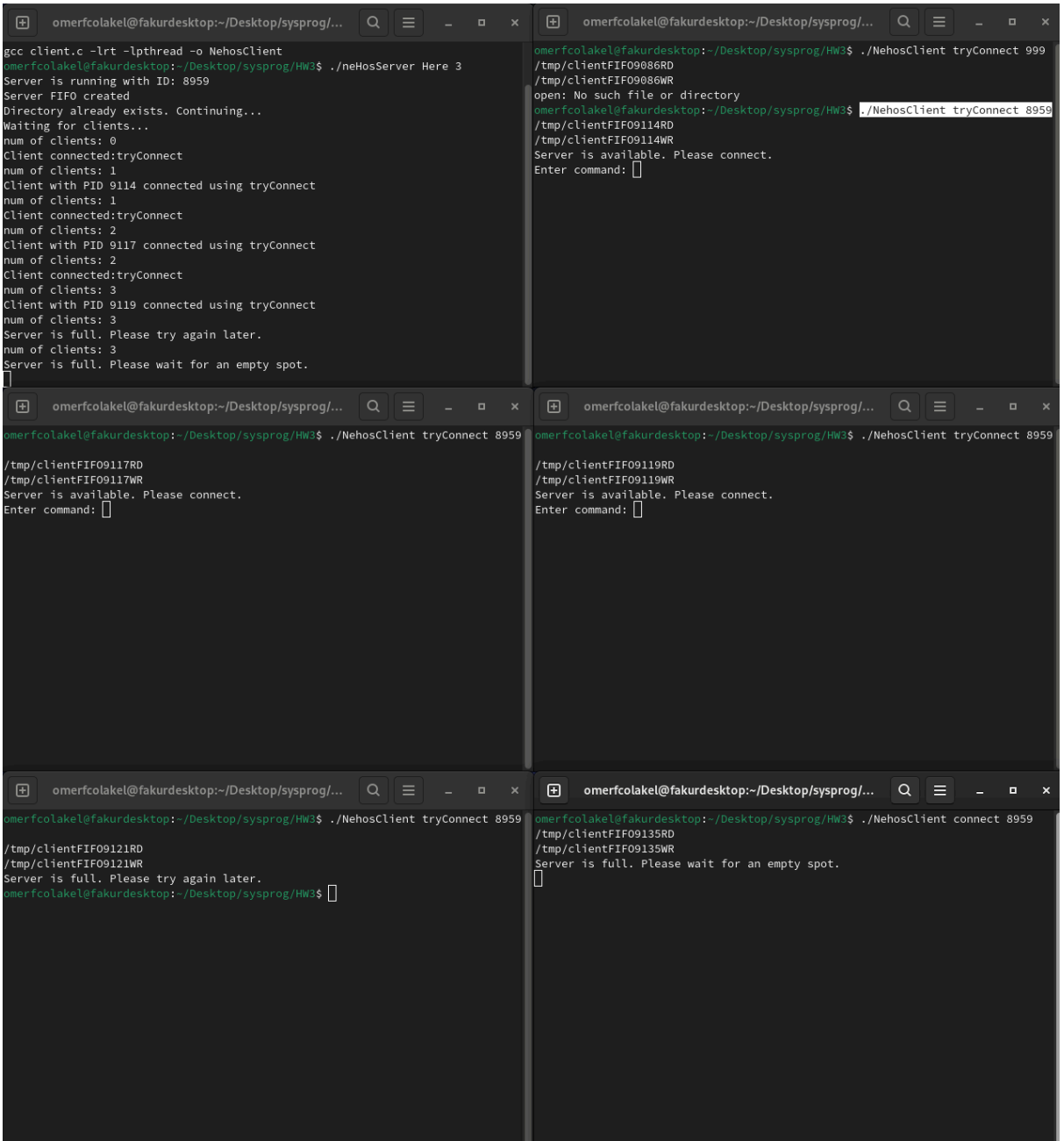
Example Input and Outputs

If you can't see what is in the images, you can check the Report folder.

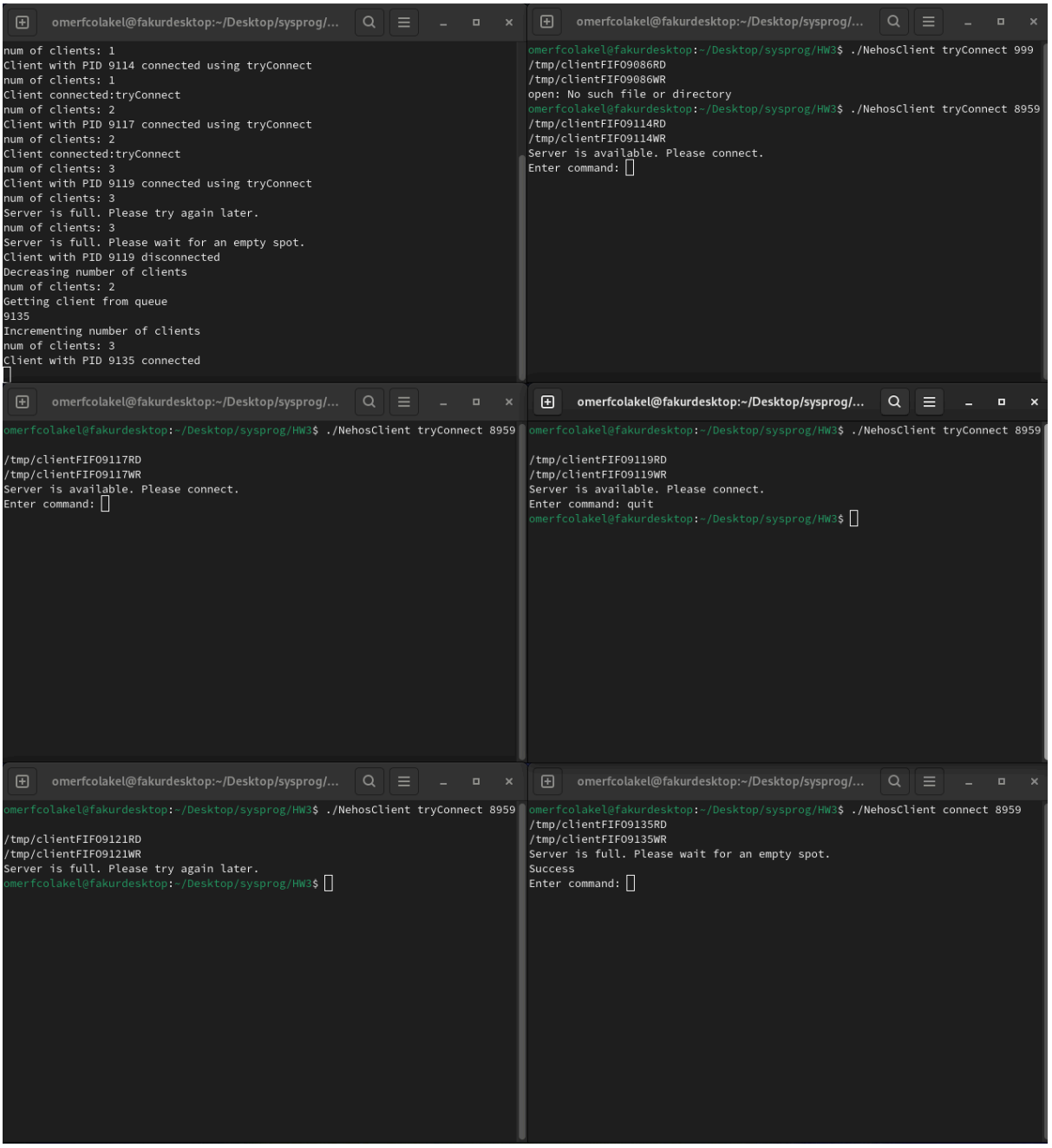
1. Trying to connect to a server that doesn't exist



2. Three clients fill the server. Fourth one can't connect with tryconnect, fifth waits since its connect

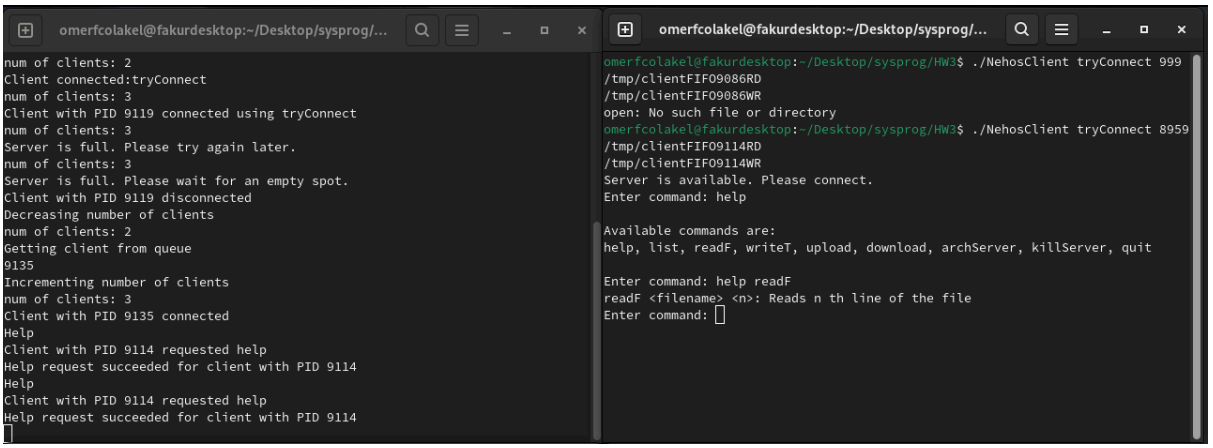


3. Connect client connects after a client quits.

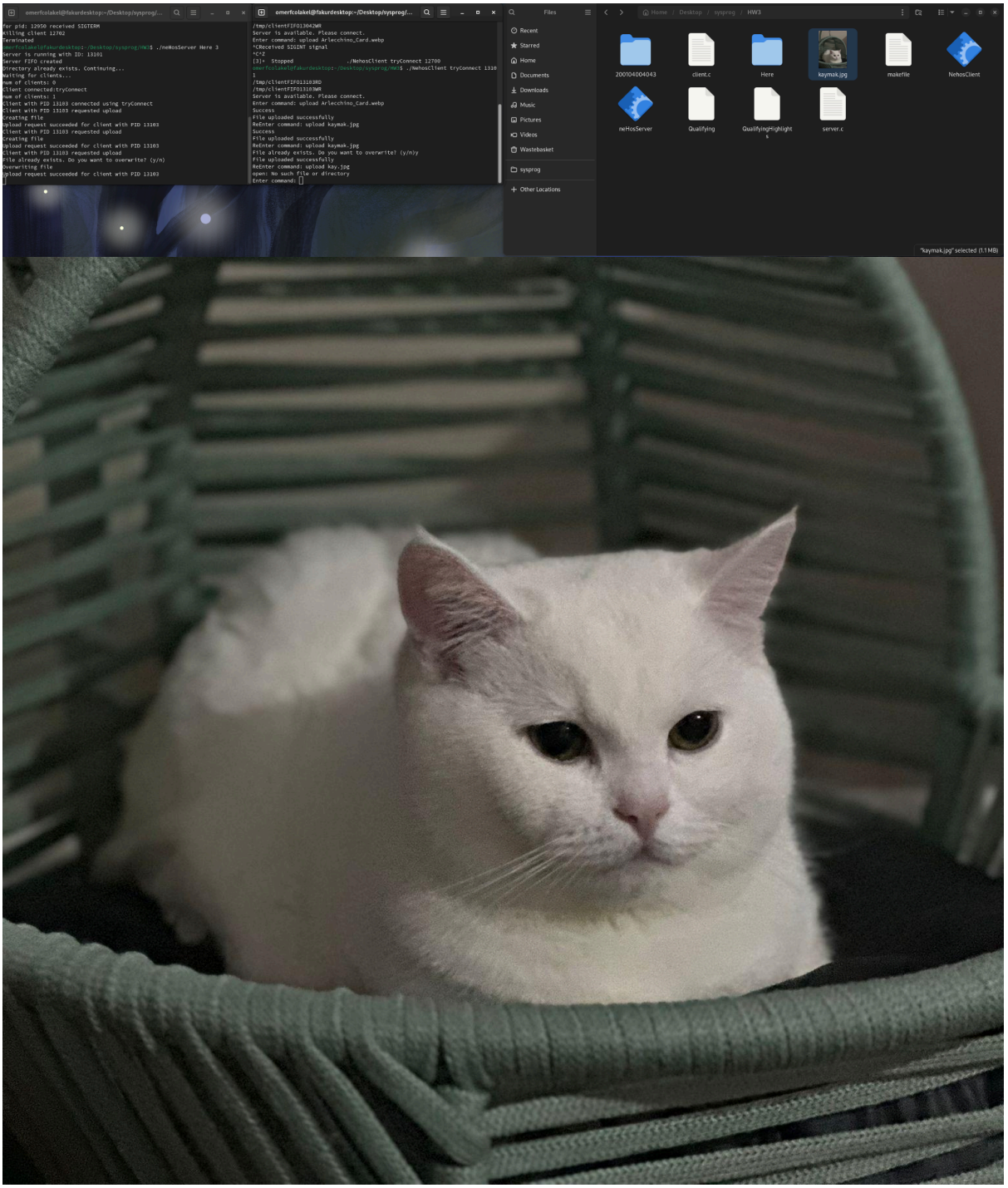


Continue on the next page =>

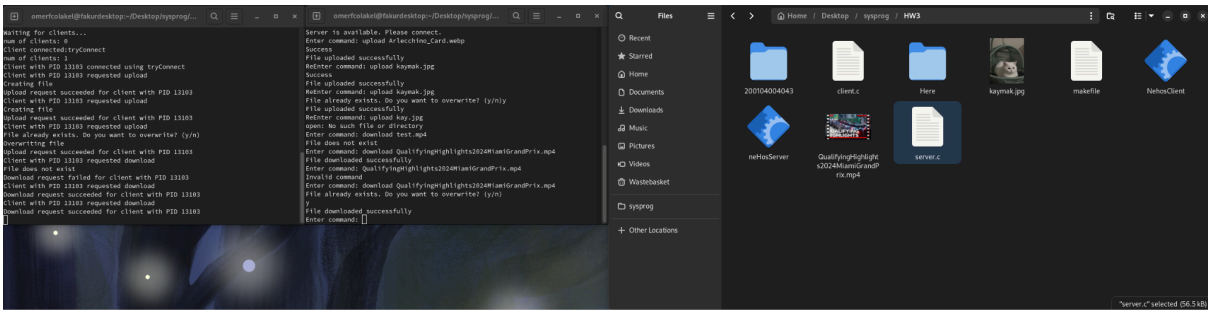
4. help and help readF



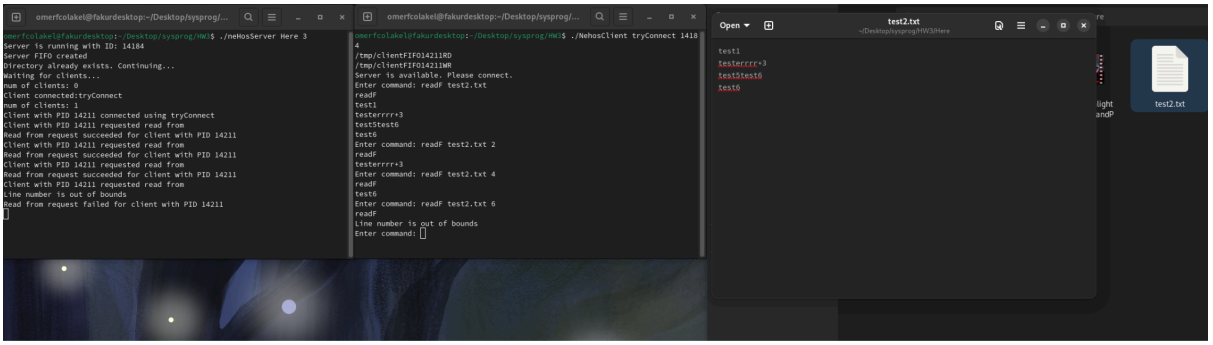
5. Uploading a file, an already existing file on the server and the file not existing on the client.



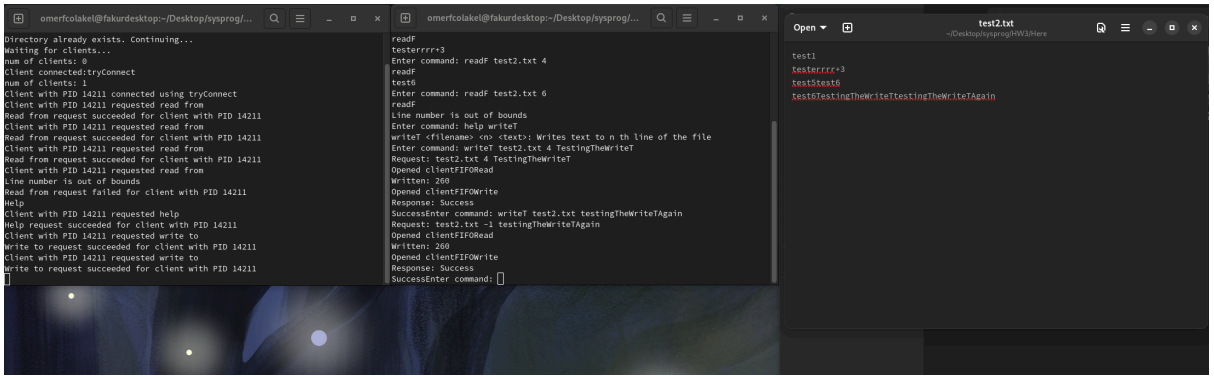
6. Downloading a file, overwriting it and the file not existing in the server. The file itself is 300MB. Unlike kaymak.jpg, I didn't include the video in the Report folder. You might not want to download a file that big.



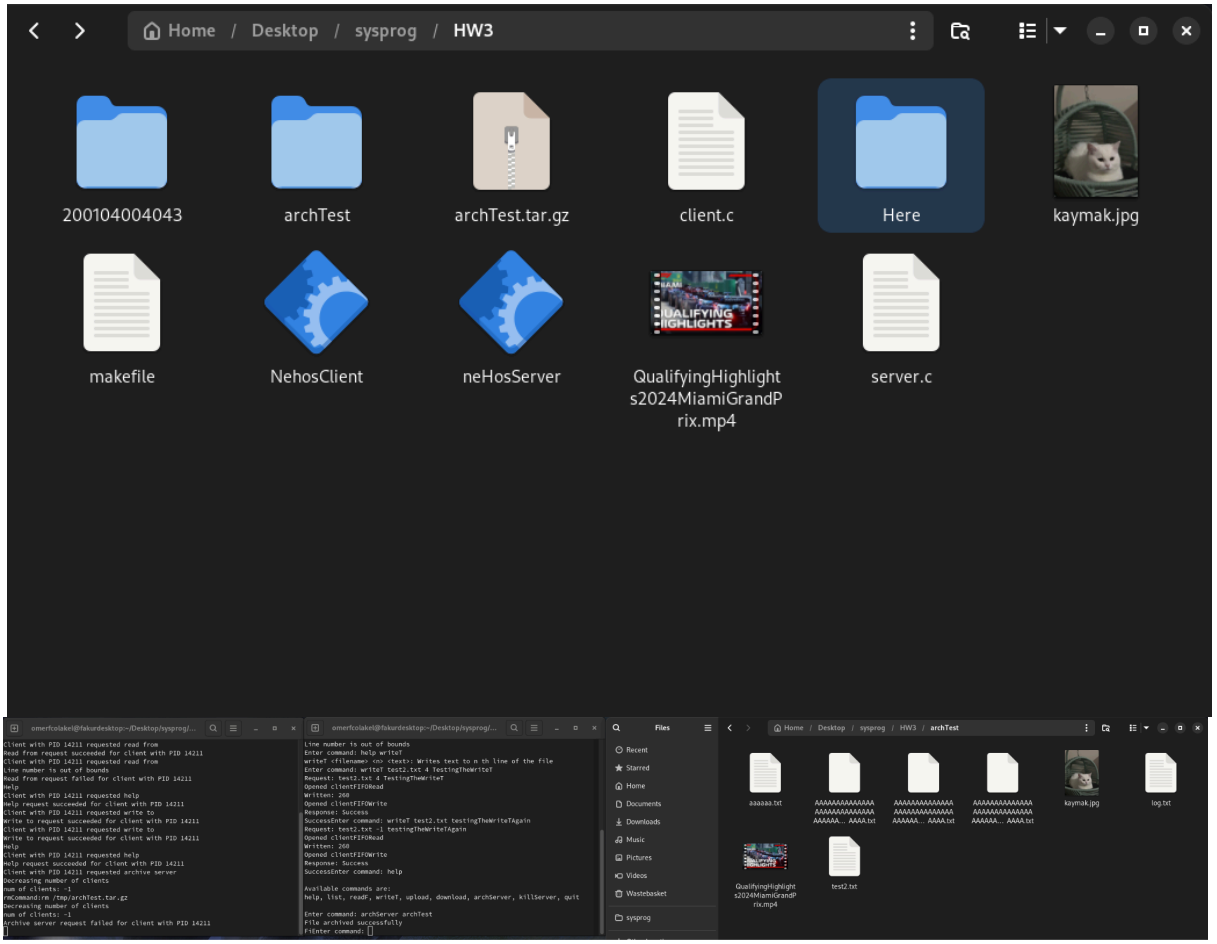
7. readF



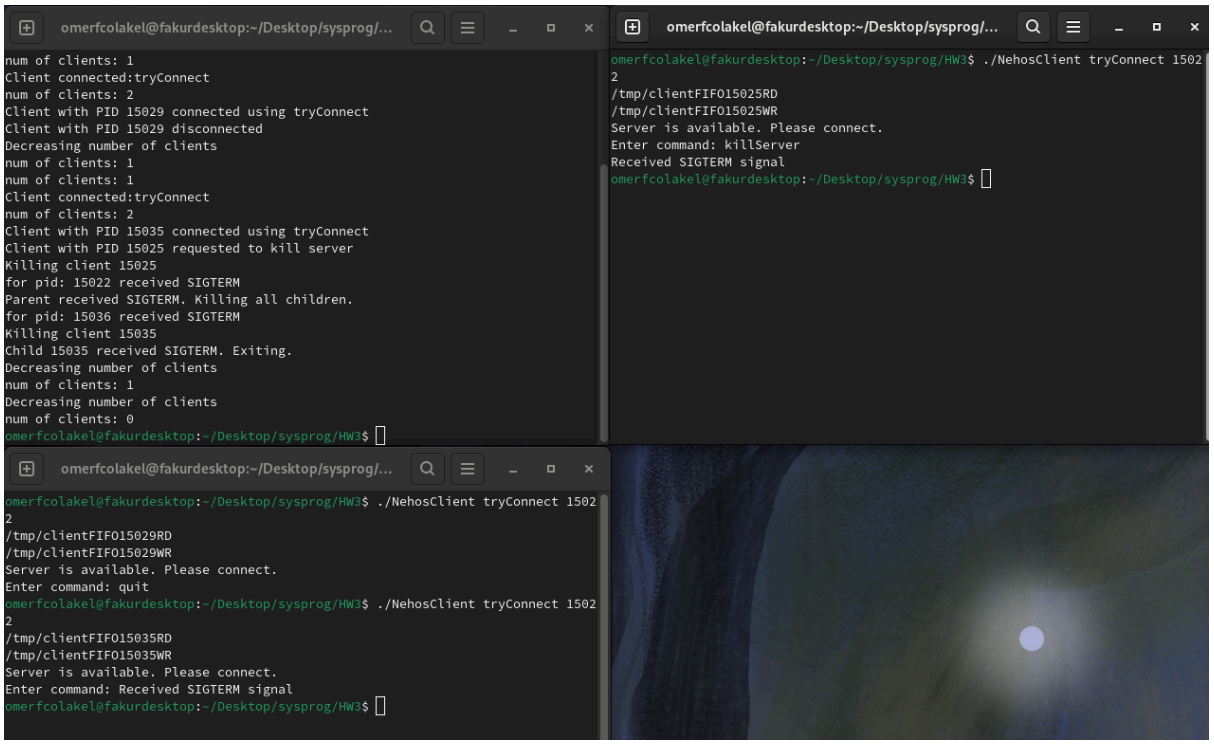
8. writeT



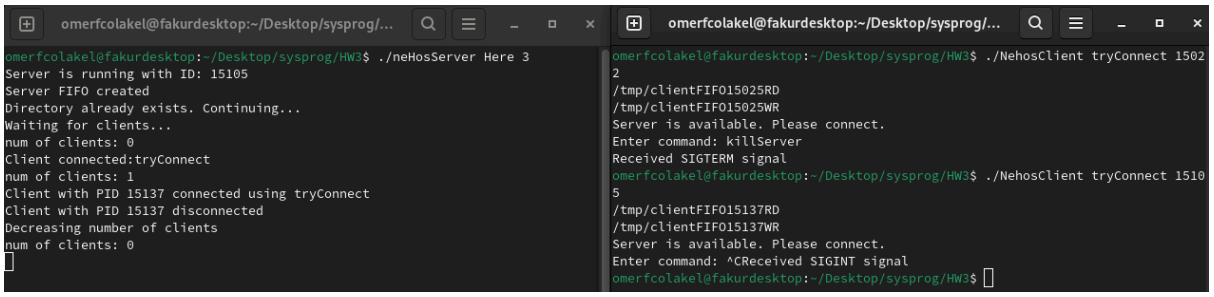
9. archServer



10. killServer



11. Quitting client with CTRL + C



12. Killing Server with CTRL + C