Gebze Technical University

Department of Computer Engineering

CSE344 System Programming

Midterm Report

Burak Yıldırım
1901042609

Contents

1 Introduction **4**
1.1 Project Description 4
1.2 Compilation 4

2 General Structure **5**

3 Server Implementation **6**

4 Client Implementation **7**

5 Connection **8**

6 Communication **9**

7 Error Handling **9**

8 Testing **10**
8.1 Connection with the Wrong PID 10
8.2 Connecting to a Full Server with tryConnect 10
8.3 Connecting to a Full Server with connect 11
8.4 Connecting to the Server After Waiting 11
8.5 List (> 200 Files) 12
8.6 Help 13
8.7 Help with an Invalid Command 13
8.8 Help with a Valid Command (writeT) 13
8.9 Writing to a Non-Integer Line 13
8.10 Writing without Double Quotes Around the String 13
8.11 Writing without a Line 14
8.12 Writing to a Non-Existing Line 14
8.13 Reading a Non-Existing File 14
8.14 Reading an Existing File 14
8.15 Reading an Existing Line 15
8.16 Reading a Non-Existing Line 15
8.17 Uploading a Non-Existing File 15
8.18 Uploading an Existing File 16
8.18.1 Before 16
8.18.2 After 16
8.19 Downloading an Existing File 17
8.19.1 Before 17
8.19.2 After 17
8.20 Downloading a Non-Existing File 18
8.21 Archiving the Server Directory without .tar Extension 18
8.22 Archiving the Server Directory 18
8.23 Killing the Server with killServer Command 19

8.24 Killing the Server with SIGINT 19
8.25 Disconnecting from Server with Quit Command 20
8.26 Disconnecting from Server with SIGINT 20

9 Logs **21**

1 Introduction

1.1 Project Description

The task of this project is to design and implement a file server that enables multiple clients to connect, access, modify and archive the files in a specific directory located at the server side. This system is divided into server-side and client-side programs.

1.2 Compilation

```makefile
CC = gcc
CFLAGS = -Wall -std=c99
IPCFLAGS = -pthread -lrt
DFLAGS = -g
INCDIR = include
SRCDIR = src
LOG_EXT = log
LOCK_EXT = dirlock
SERVER_TARGET = neHosServer
CLIENT_TARGET = neHosClient
UTIL_SOURCES = $(wildcard $(SRCDIR)/utils/*.c)
SHARED_SOURCE = $(SRCDIR)/shared.c
SERVER_SOURCES = $(SRCDIR)/server.c $(SHARED_SOURCE) $(UTIL_SOURCES)
CLIENT_SOURCES = $(SRCDIR)/client.c $(SHARED_SOURCE) $(UTIL_SOURCES)
HEADERS = $(wildcard $(INCDIR)/*.h) $(wildcard $(INCDIR)/utils/*.h)
VALGRIND_OPTIONS = --leak-check=full --show-leak-kinds=all --track-origins=yes
V_DIR =
V_CLIENT =
V_CONNECT =
V_PID =

export V_DIR
export V_CLIENT
export V_CONNECT
export V_PID

OS := $(shell uname)

ifeq ($(OS), Darwin)
        IPCFLAGS =
endif

all: server client

server: $(SERVER_SOURCES) $(HEADERS)
        $(CC) $(CFLAGS) -o $(SERVER_TARGET) $(SERVER_SOURCES) -I$(INCDIR) $(IPCFLAGS)

client: $(CLIENT_SOURCES) $(HEADERS)
        $(CC) $(CFLAGS) -o $(CLIENT_TARGET) $(CLIENT_SOURCES) -I$(INCDIR) $(IPCFLAGS)

debug_server:
        $(CC) $(CFLAGS) $(DFLAGS) -o $(SERVER_TARGET) $(SERVER_SOURCES) -I$(INCDIR) $(IPCFLAGS)

debug_client:
        $(CC) $(CFLAGS) $(DFLAGS) -o $(CLIENT_TARGET) $(CLIENT_SOURCES) -I$(INCDIR) $(IPCFLAGS)

clean:
        rm -f $(SERVER_TARGET) $(CLIENT_TARGET) *.$(LOG_EXT) .*.$(LOCK_EXT)

valgrind_server: debug_server
        valgrind $(VALGRIND_OPTIONS) ./$(SERVER_TARGET) $(V_DIR) $(V_CLIENT)

valgrind_client: debug_client
        valgrind $(VALGRIND_OPTIONS) ./$(CLIENT_TARGET) $(V_CONNECT) $(V_PID)
```

The provided Makefile automates the build process: executing **make** compiles the project, **make clean** removes executables, logs and lock files, and **make valgrind_server** and **make valgrind_client** launches the server and client programs in Valgrind for memory leak checks, streamlining development and debugging.

2 General Structure

```
Midterm
├── include
│   ├── utils
│   │   ├── array.h
│   │   ├── io.h
│   │   ├── log.h
│   │   ├── string.h
│   │   └── type.h
│   ├── client.h
│   ├── server.h
│   └── shared.h
├── src
│   ├── utils
│   │   ├── array.c
│   │   ├── io.c
│   │   ├── log.c
│   │   └── string.c
│   ├── client.c
│   ├── server.c
│   └── shared.c
└── Makefile
```

This is the folder structure of the project.

- utils: utility functions used by multiple files.
- client: client-specific functions and macros.
- server: server-specific functions and macros.
- shared: common functions and macros used by client and server.
- Makefile: compile project and clean executables, logs and lock files.

3 Server Implementation

Structs and Enums

```
typedef enum {
    RUNNING = 0,
    WAITING = 1
} ProcessState;

typedef struct {
    int id;
    pid_t server_pid;
    pid_t client_pid;
    ProcessState state;
} ProcessTableEntry;

typedef struct {
    int size;
    int num_running;
    int num_waiting;
    int process_count;
    ProcessTableEntry *entries;
} ProcessTable;
```

The server is implemented following these steps:

1. Server directory is created.
2. `SIGCHLD`, `SIGINT`, `SIGUSR1` are handled.
3. `ProcessTable process_table` is initialized with an entry size that is double the given number of maximum clients.
4. 2 semaphores and 1 shared memory segment, all named using the server's PID for uniqueness, are created for managing connections.
5. 2 semaphores and 1 shared memory segment, all named using the server's PID for uniqueness, are created for managing terminated waiting clients to remove them from the queue.
6. After a connection request if the queue is not full, a child process is created using `fork()` syscall. 2 semaphores and 1 shared memory segment, all named using the client's PID for uniqueness, are opened for managing the communication between the client and the child process.
7. `SIGTERM`, `SIGINT`, `SIGUSR2` are handled for child process.
8. After that, the child runs in an infinite loop, processing the commands from the client.

For parent process, `SIGUSR1` is used for removing the terminated waiting clients from the queue, `SIGCHLD` is used for removing the finished clients from the queue and putting waiting clients in the running part of the queue, `SIGINT` is used for sending each child process a `SIGTERM` and exiting. For child server, `SIGUSR2` is used for logging an error message and exiting the child process if the client encountered an error, and `SIGTERM` is used for sending a `SIGTERM` signal to the client and exiting the child process if the parent process is terminated, and `SIGINT` is ignored since `SIGINT` is handled in the parent process.

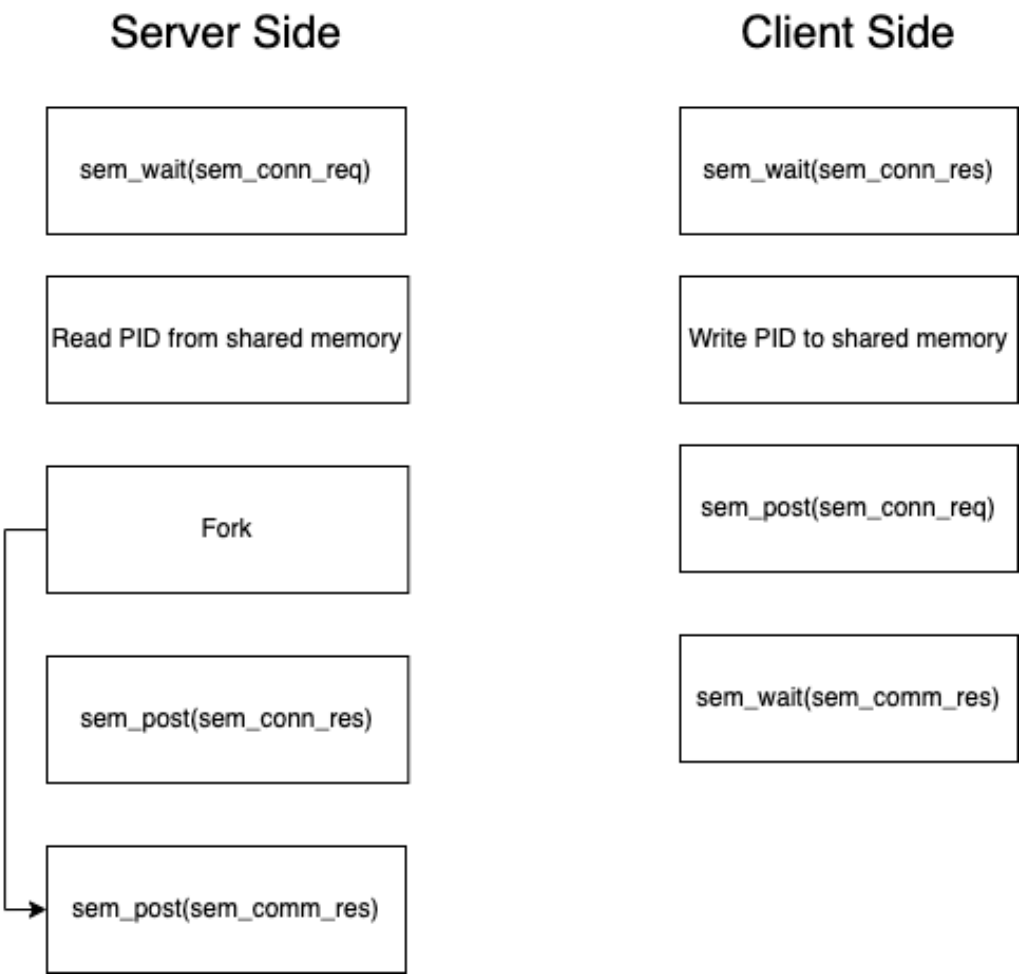
4 Client Implementation

The client is implemented following these steps:

1. `SIGINT`, `SIGTERM`, `SIGUSR1`, `SIGUSR2` are handled.
2. 2 semaphores and 1 shared memory segment, all named using the parent server's PID for uniqueness, are opened for connection.
3. 2 semaphores and 1 shared memory segment, all named using the parent server's PID for uniqueness, are opened for managing termination if a `SIGINT` is produced while the client is waiting.
4. If server queue is not full, client connects to the child server, and 2 semaphores and 1 shared memory segment, all named using the client's PID for uniqueness, are created for managing the communication between the client and the child server.
5. After that the client runs in an infinite loop, dispatching commands from the user.

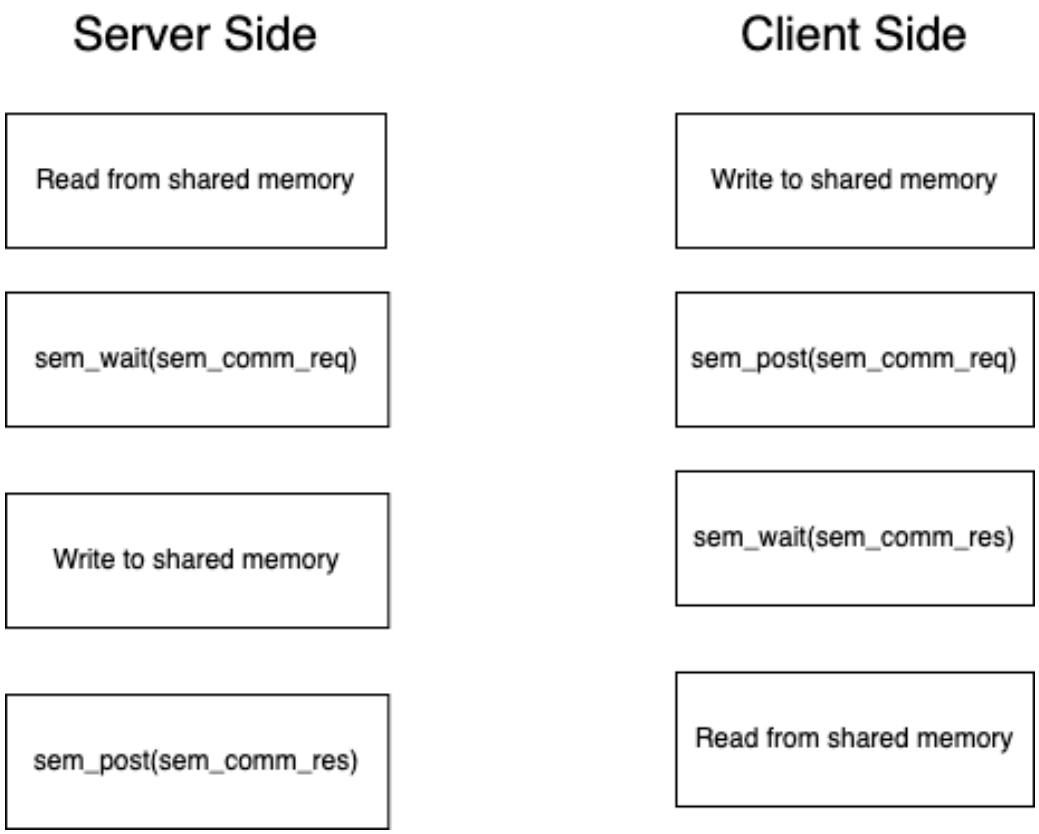
`SIGINT` is used for sending a `SIGTERM` signal to the child server and exiting if the client is connected to the server, otherwise sending a `SIGUSR1` signal to the parent server. `SIGTERM` is used for printing a message indicating that the server is terminated and exiting. `SIGUSR1` is used for printing a waiting message if "connect" option is used, otherwise exiting.

5 Connection



In the client side, first client waits the connection response from the server in case of a new client is connecting (sem_conn_res is initialized with 1 when created at server side). Then the client writes its PID to the shared memory and and posts the connection request. After that it waits the communication response which is only posted if the child server is created. In the server side, the server waits for connection request. Then the server read the PID of the client and checks if the queue is full. If not, then the server creates a child process using fork and posts connection response for possible future clients to connect. After the fork, the child server posts the communication response and the connection between the client and the server is done.

6 Communication



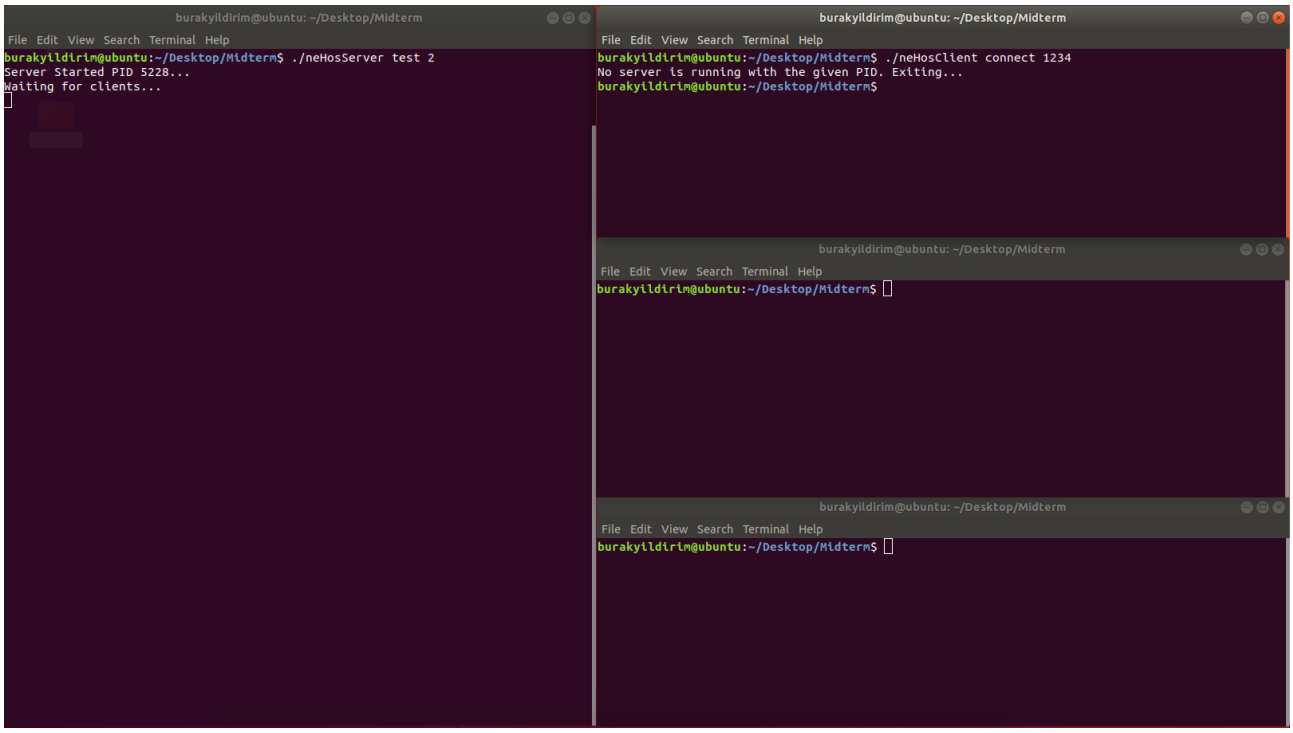
In the client side, the client writes the data to the shared memory and posts a communication request. After that it waits for a communication response. In the server side, server reads from the shared memory by means of pointer casting, so that even if the shared memory content changes while it's waiting, the data it read will be the latest. After receiving a communication request, it process the request and writes the outcome to the shared memory. Then it posts a communication response. When the client receives a communication response, it reads the response from shared memory. This is done in an infinite loop.

7 Error Handling

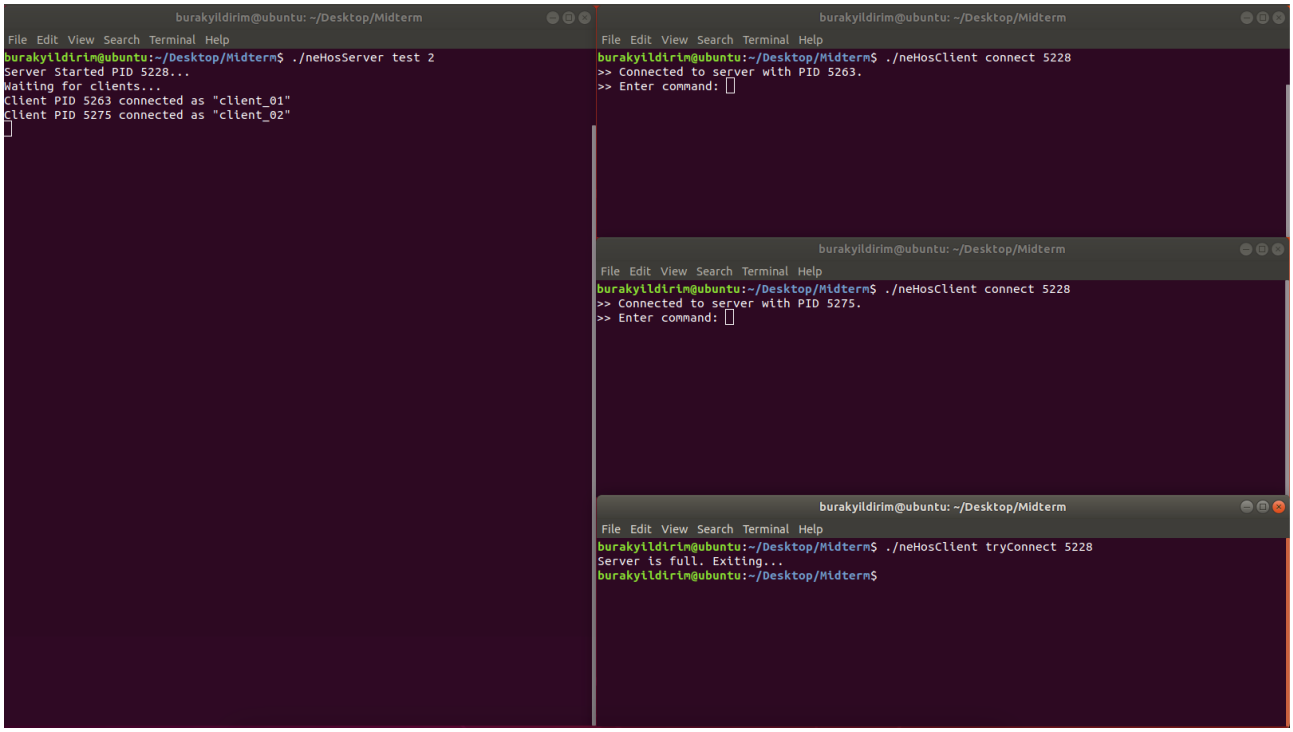
After each syscall (open, write, etc.) a check is done to see if any error has happened. If it has then appropriate messages are printed using `perror` and exited.

8 Testing

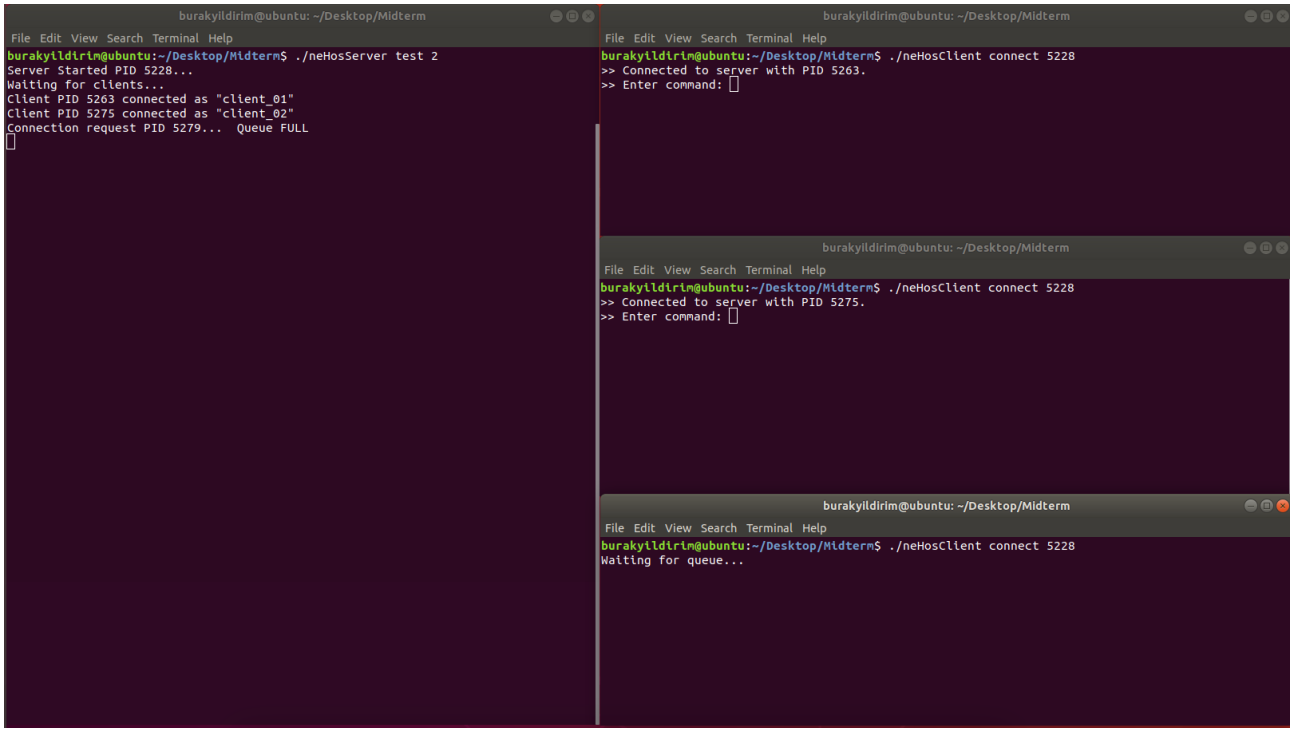
8.1 Connection with the Wrong PID



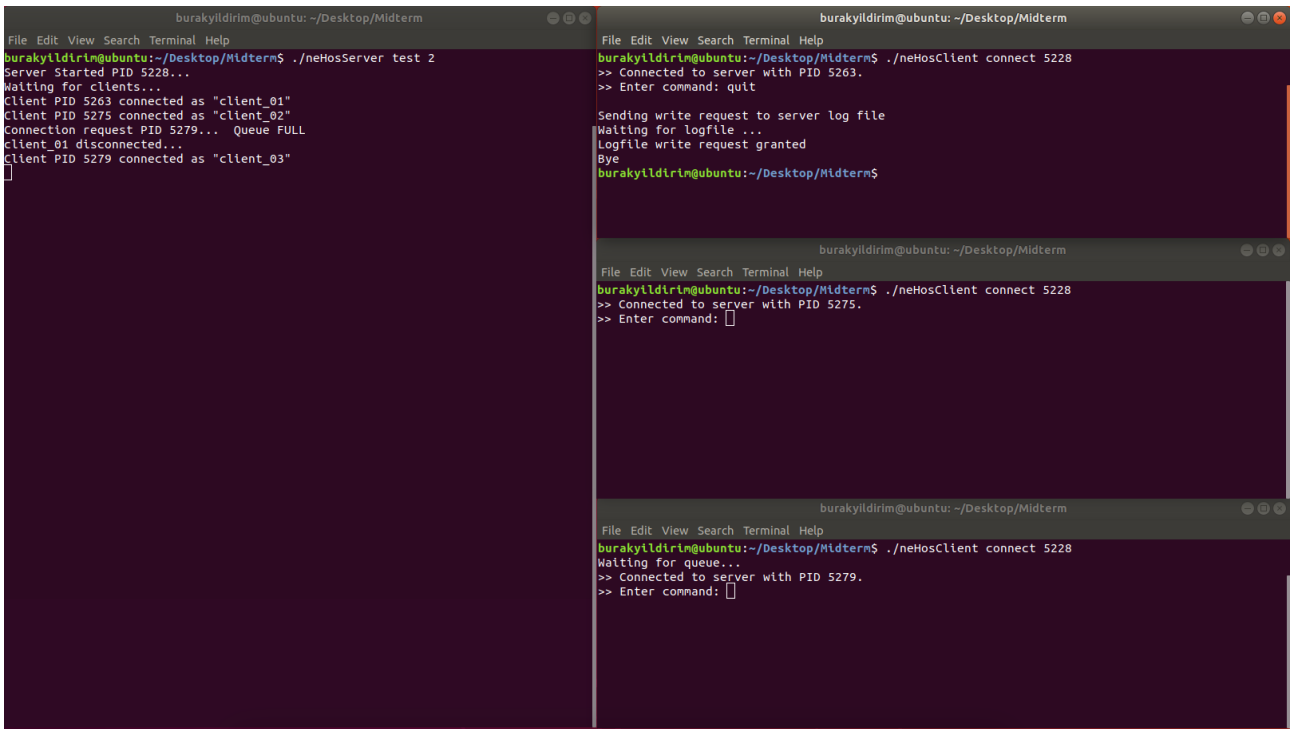
8.2 Connecting to a Full Server with tryConnect



8.3 Connecting to a Full Server with connect



8.4 Connecting to the Server After Waiting



8.5 List (> 200 Files)

```
>> Enter command: list

3_some_long_name.txt
64_some_long_name.txt
file.txt
10_some_long_name.txt
58_some_long_name.txt
114_some_long_name.txt
124_some_long_name.txt
19_some_long_name.txt
78_some_long_name.txt
91_some_long_name.txt
22_some_long_name.txt
31_some_long_name.txt
49_some_long_name.txt
77_some_long_name.txt
37_some_long_name.txt
90mb.txt
126_some_long_name.txt
80_some_long_name.txt
54_some_long_name.txt
90_some_long_name.txt
103_some_long_name.txt
98_some_long_name.txt
6_some_long_name.txt
73_some_long_name.txt
15mb.txt
92_some_long_name.txt
52_some_long_name.txt
70_some_long_name.txt
95_some_long_name.txt
.DS_Store
109_some_long_name.txt
113_some_long_name.txt
120_some_long_name.txt
71_some_long_name.txt
25_some_long_name.txt
45_some_long_name.txt
24_some_long_name.txt
29_some_long_name.txt
12_some_long_name.txt
59_some_long_name.txt
2_some_long_name.txt
26_some_long_name.txt
35_some_long_name.txt
89_some_long_name.txt
5mb.txt
```

8.6 Help

```
>> Enter command: help

Available commands are:
help, list, readF, writeT, upload, download, archServer, killServer, quit
```

8.7 Help with an Invalid Command

```
>> Enter command: help invalid

Invalid command. Type 'help' to see the list of commands.
```

8.8 Help with a Valid Command (writeT)

```
>> Enter command: help writeT

writeT <file> <line #> <string>
  Writes the content of <string> to the #th line of the <file>, if no line number is given writ
es to the end of <file>. If the <file> does not exists in the server's directory creates and ed
its the <file> at the same time. <string> must be enclosed in double quotes.
```

8.9 Writing to a Non-Integer Line

```
>> Enter command: writeT deneme.txt nonint "deneme"

Invalid argument. Line number should be a positive number.
```

8.10 Writing without Double Quotes Around the String

```
>> Enter command: writeT deneme.txt deneme

Invalid number of arguments for the command. Here are the possible options for the command:

writeT <file> <line #> <string>
  Writes the content of <string> to the #th line of the <file>, if no line number is given writes to the end of <file>. If the <file> does not exists in the server's directory c
reates and edits the <file> at the same time. <string> must be enclosed in double quotes.
```

8.11 Writing without a Line

```
>> Enter command: writeT deneme.txt "deneme"

File write request received. Beginning file write...
6 bytes written
```

8.12 Writing to a Non-Existing Line

```
>> Enter command: writeT deneme.txt 123456 "\ndeneme2"

File write request received. Beginning file write...
8 bytes written
```

8.13 Reading a Non-Existing File

```
>> Enter command: readF nonexist.txt

File not found
```

8.14 Reading an Existing File

```
>> Enter command: readF deneme.txt

deneme
deneme2

Bytes read: 14
```

8.15 Reading an Existing Line

```
>> Enter command: readF deneme.txt 2

deneme2

Bytes read: 7
```

8.16 Reading a Non-Existing Line

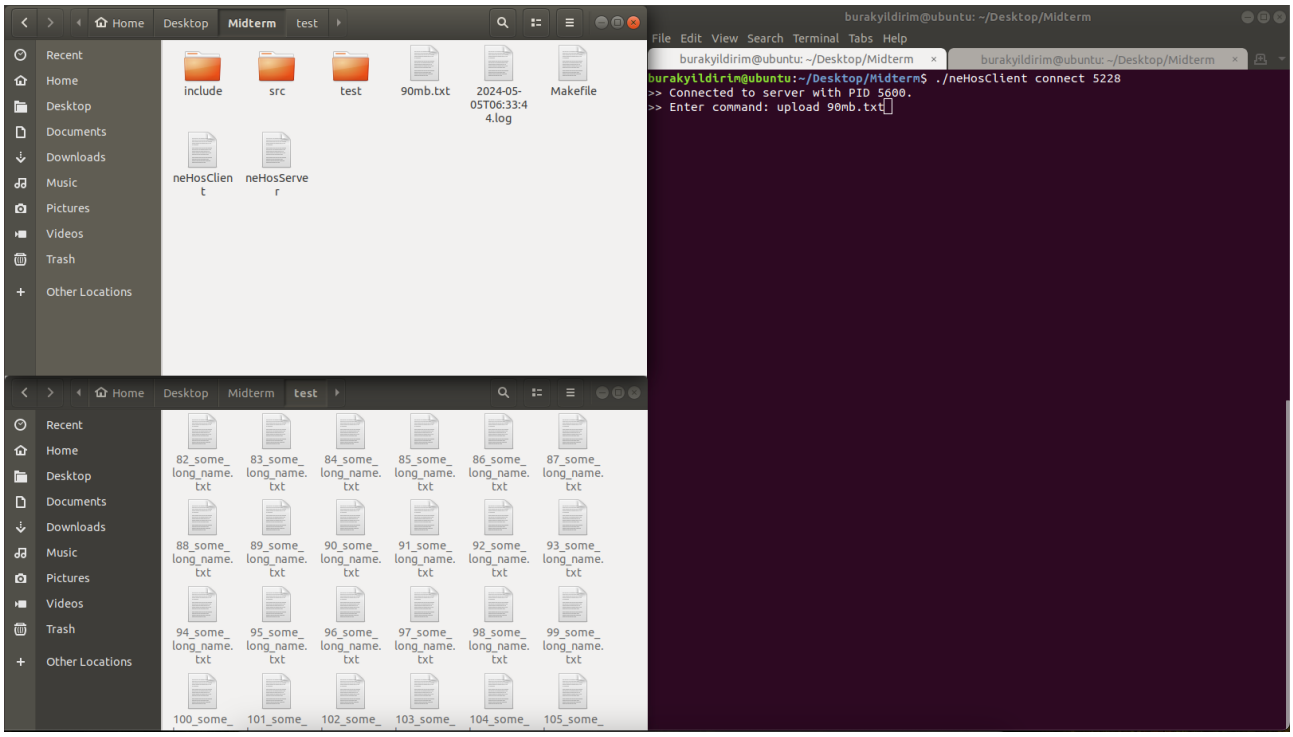
```
>> Enter command: readF deneme.txt 1234

Line not found
```

8.17 Uploading a Non-Existing File

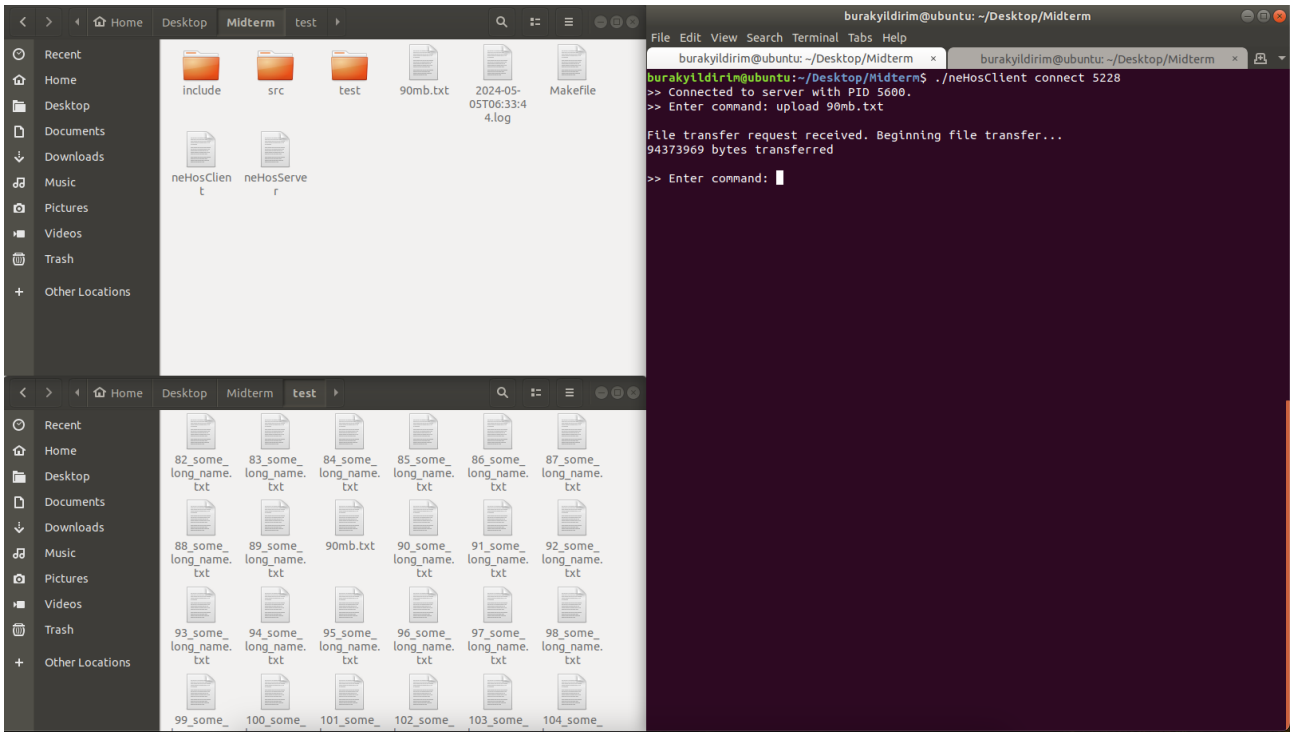
```
>> Enter command: upload nonexist

File not found
```

8.18 Uploading an Existing File

8.18.1 Before

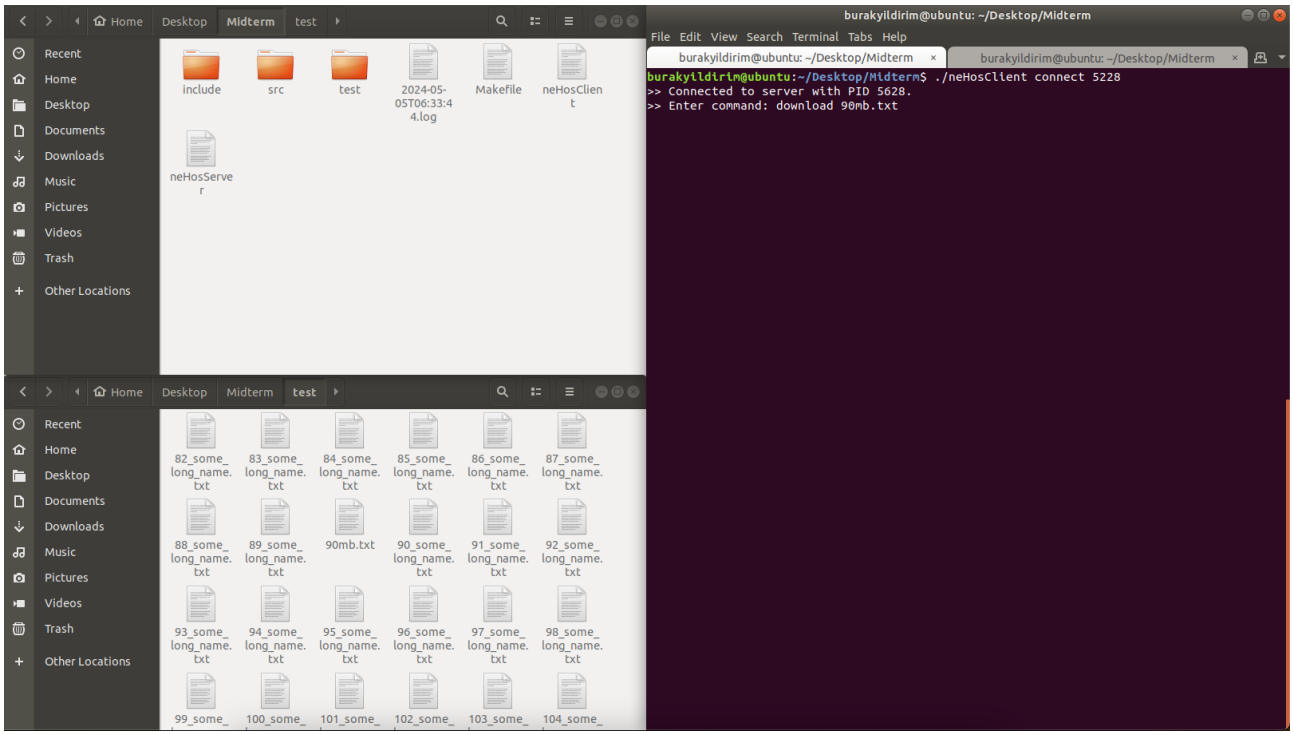


8.18.2 After

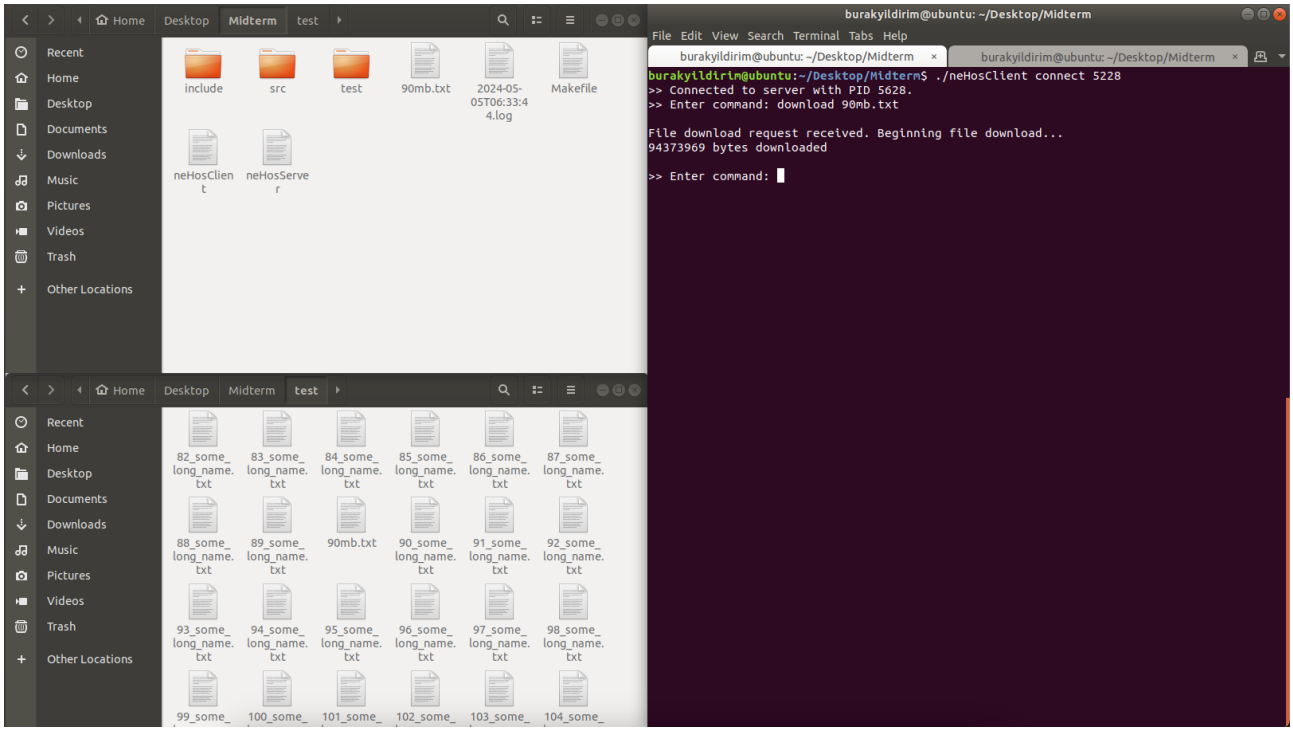


8.19 Downloading an Existing File

8.19.1 Before



8.19.2 After



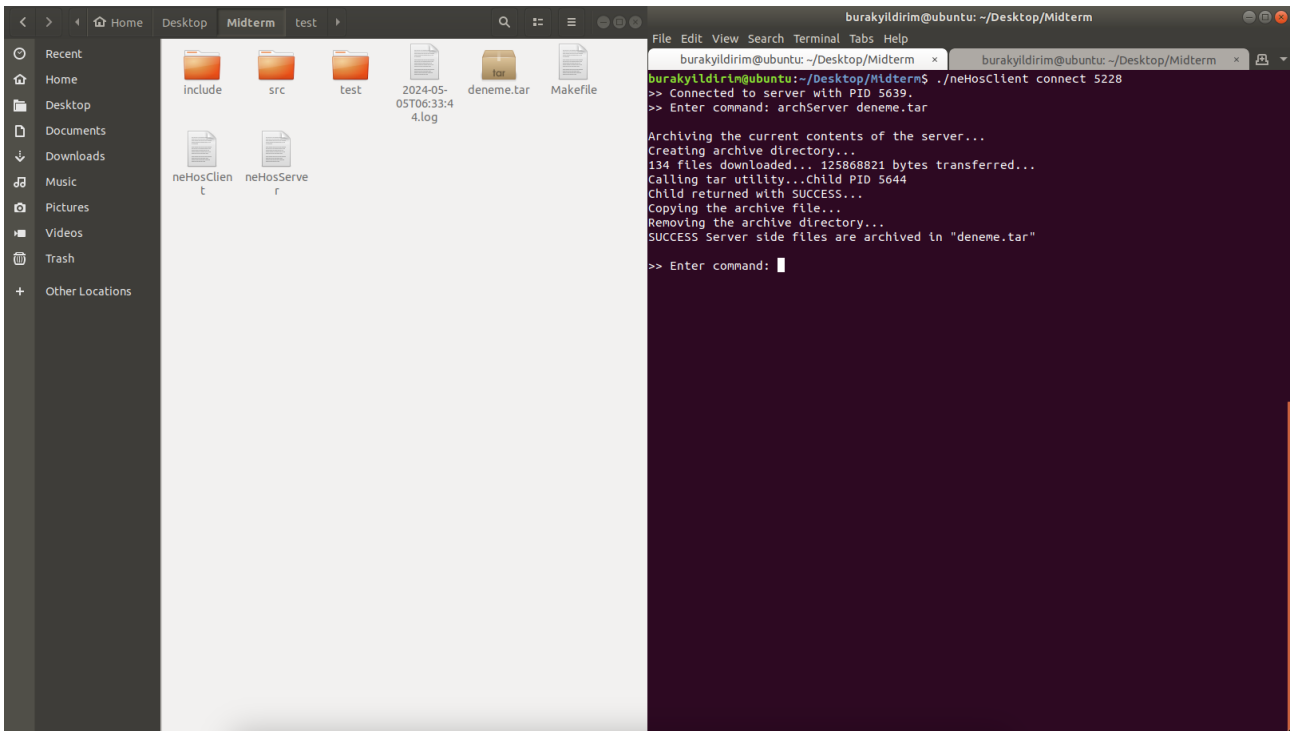
8.20 Downloading a Non-Existing File

```
>> Enter command: download nonexist

File not found
```

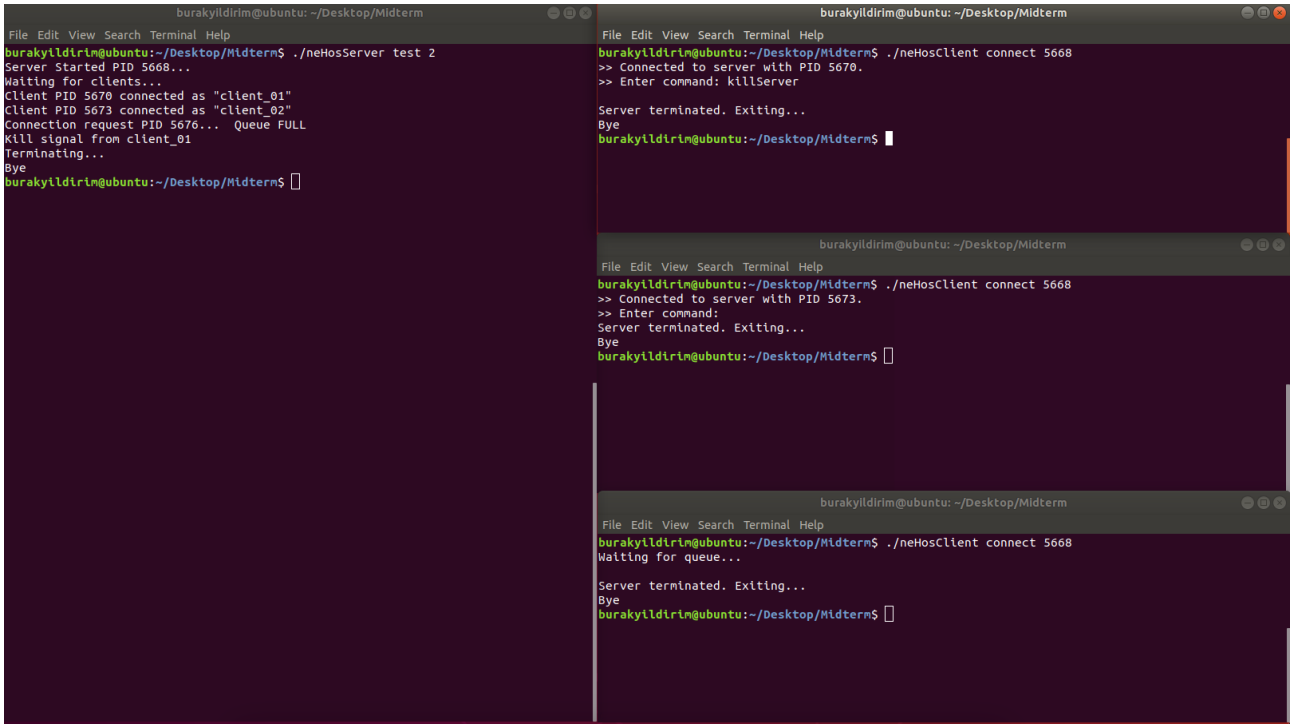
8.21 Archiving the Server Directory without .tar Extension

```
>> Enter command: archServer notarext

Invalid filename. Filename should be in the format <filename>.tar
```

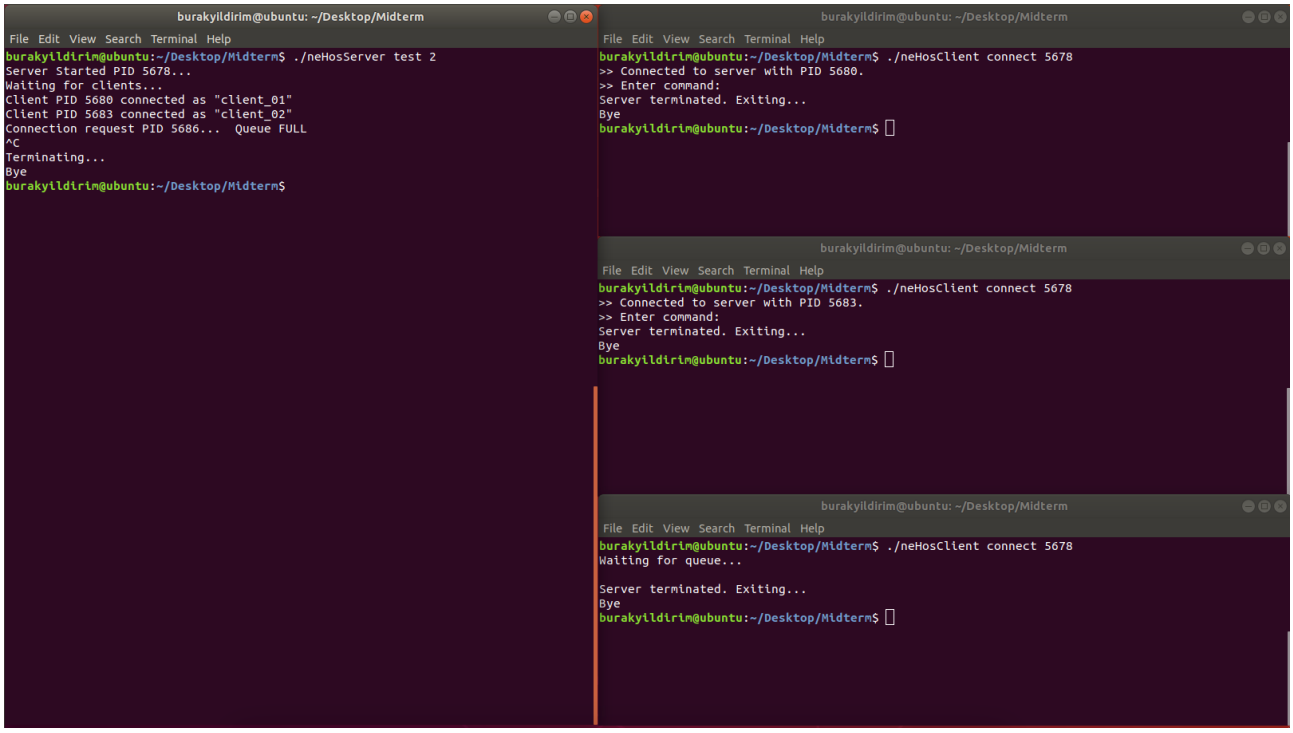
8.22 Archiving the Server Directory



8.23 Killing the Server with killServer Command



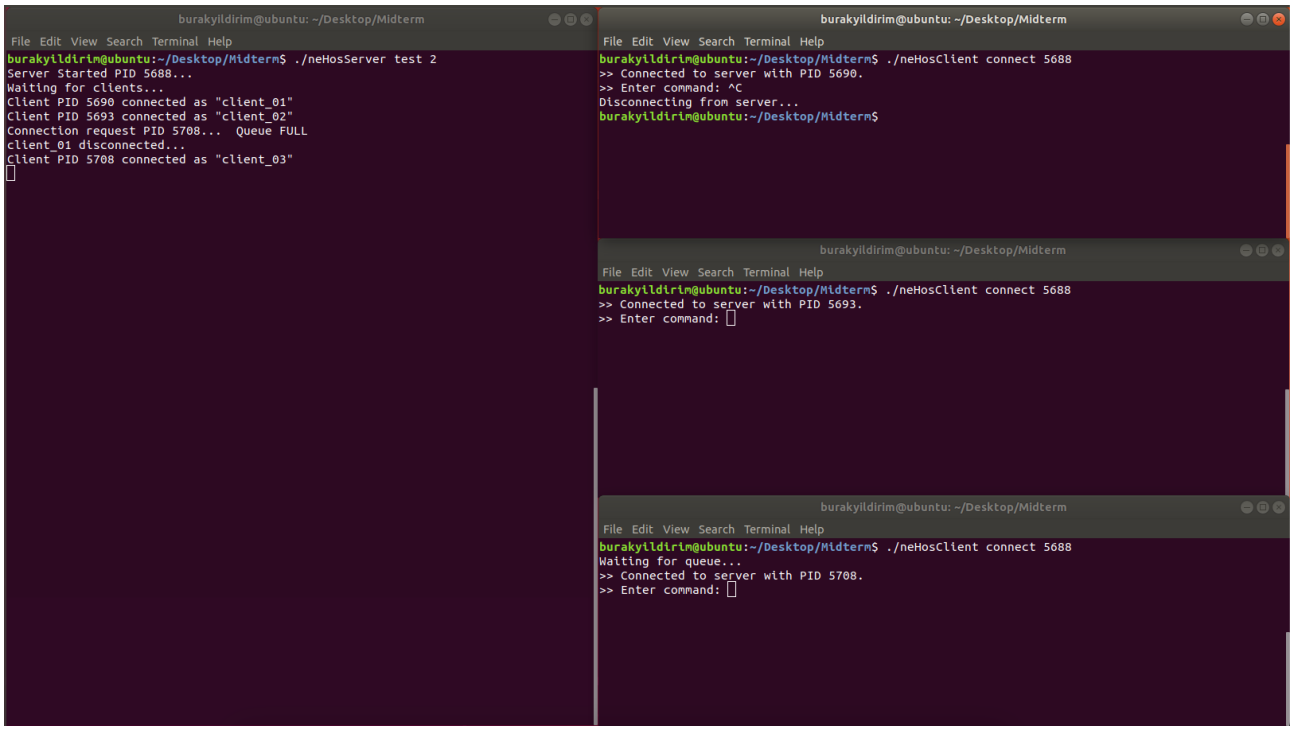
8.24 Killing the Server with SIGINT



8.25 Disconnecting from Server with Quit Command

```
>> Enter command: quit

Sending write request to server log file
Waiting for logfile ...
Logfile write request granted
Bye
```

8.26 Disconnecting from Server with SIGINT



9 Logs

```
[2024-05-05 08:28:28] [INFO] Server (PID: 5924) started.
[2024-05-05 08:28:35] [INFO] Client (PID: 5925) connected to Child Server (PID 5926) as client_01.
[2024-05-05 08:28:41] [INFO] Client (PID: 5925) listed all the files in the directory test.
[2024-05-05 08:29:02] [INFO] Client (PID: 5925) created the file deneme.txt and wrote "deneme".
[2024-05-05 08:29:16] [INFO] Client (PID: 5925) wrote "\ndeneme2" to the end of the file deneme.txt.
[2024-05-05 08:29:43] [INFO] Client (PID: 5925) tried to read the file nonexist.txt but it was not found.
[2024-05-05 08:29:56] [INFO] Client (PID: 5925) read the file deneme.txt.
[2024-05-05 08:30:20] [INFO] Client (PID: 5925) read the line 2 from the file deneme.txt.
[2024-05-05 08:33:52] [INFO] Client (PID: 5925) uploaded the file 90mb.txt.
[2024-05-05 08:34:21] [INFO] Client (PID: 5925) downloaded the file 90mb.txt.
[2024-05-05 08:36:36] [INFO] Client (PID: 5925) tried to download the file nonexist but it was not found.
[2024-05-05 08:38:28] [INFO] Server (PID: 5924) terminated.
```