

# CENG 421 PROJECT

AHMET KEREM ŞAYLI  
290201035

## 1. Introduction & Project Overview

This project implements a **client-server file sharing system** in C, utilizing **TCP sockets** for reliable data transfer. The key objectives are:

- Demonstrate **basic network communication** in C using standard system calls like `socket()`, `bind()`, `listen()`, `accept()`, etc.
- Provide a “**useful**” **service** where multiple clients can connect to a server to **upload, download, list, rename, or delete files**.
- Implement **privileged commands** for the first connected client (admin) to restrict operations like rename and delete to the admin only.
- Ensure the program **compiles under GCC** without external third-party libraries, in alignment with class requirements.

### Libraries Used

`stdio.h`, `stdlib.h`, `string.h`, `unistd.h`, `arpa/inet.h`, `sys/socket.h`, `fcntl.h`, `errno.h`, `dirent.h`, `sys/stat.h`

When the server starts, it listens on **port 8080**, awaiting client connections. The **first** client who connects is automatically designated as the “admin,” while subsequent clients have “regular user” privileges. The system allows:

1. **LIST** – Show server’s file directory contents.
2. **UPLOAD** – Send a file from the client’s local machine to the server.
3. **DOWNLOAD** – Retrieve a file from the server.
4. **DELETE** – Admin-only command to delete a file on the server.
5. **RENAME** – Admin-only command to rename a file on the server.
6. **EXIT** – Disconnect from the server gracefully.

This structure showcases concurrency (the server forks a child process for each client) and basic file operations in C. All data is stored on the server in a folder named `./server_files`. No advanced or “off-limits” libraries are used—only standard C, standard system calls, and the Makefile tool.

## 2. How to Compile and Run

### 2.1. Compilation

#### 1. Prepare the Source

Place the following files into a single directory:

- server.c
- client.c
- Makefile

#### 2. Run make file

This will produce **two** executable files:

- server (compiled from server.c)
- client (compiled from client.c)

### 2.2. Running the Server

- **Start the Server** by typing:

```
./server
```

The server will bind to **port 8080** and create (or check for) a directory called `server_files` to store incoming files.

### 2.3. Running the Client

- **Start a Client** in a separate terminal (or on a different machine, if accessible) by typing:

```
./client <username>
```

Replace `<username>` with any desired username, for example:

```
./client kerem
```

This username is sent to the server so it knows who is connecting.

- The **first** client that connects becomes **admin**; additional clients become regular users.

### 3. What the Program Does & Example Workflow

#### 1. First Client (Admin) Connection

- The first client to connect is greeted with:  
    *"Welcome <username>! You are the admin."*
- Admin privileges: Can DELETE or RENAME files on the server.

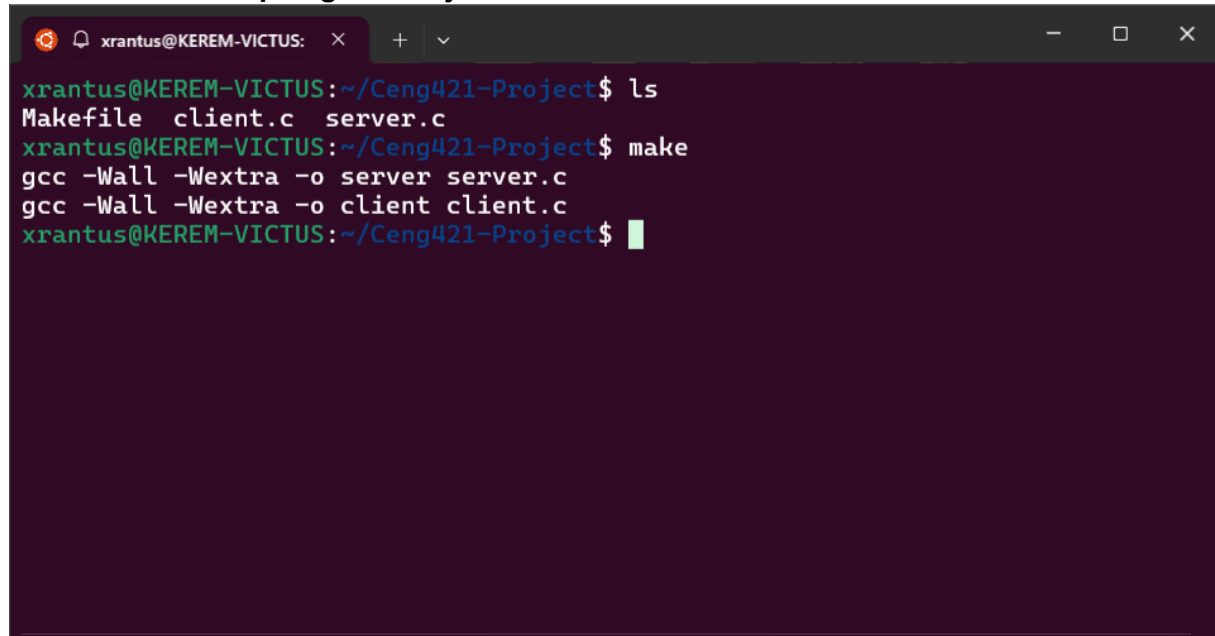
#### 2. Subsequent Client Connections

- Any further clients see a message:  
    *"Welcome <username>! You are a regular user."*
- Regular users can LIST, UPLOAD, DOWNLOAD, and EXIT but **cannot** DELETE or RENAME.

#### 3. Commands

- **LIST**
  - Server enumerates the files in ./server\_files and their sizes, sends that list back to the client.
- **UPLOAD <filename>**
  - The client reads a local file in chunks and sends it to the server, which writes it into ./server\_files.
- **DOWNLOAD <filename>**
  - The client requests a file from ./server\_files. The server sends it in chunks until complete.
- **DELETE <filename> (Admin-only)**
  - Removes the specified file from ./server\_files. Only the admin can do this.
- **RENAME <old> <new> (Admin-only)**
  - Renames <old> to <new> in ./server\_files. Only the admin can do this.
- **EXIT**
  - Closes the client's connection gracefully. The server logs the user's disconnection and, if that user was admin, the server promotes the next connected user to admin (if any remain).

## Makefile and Compiling the Project



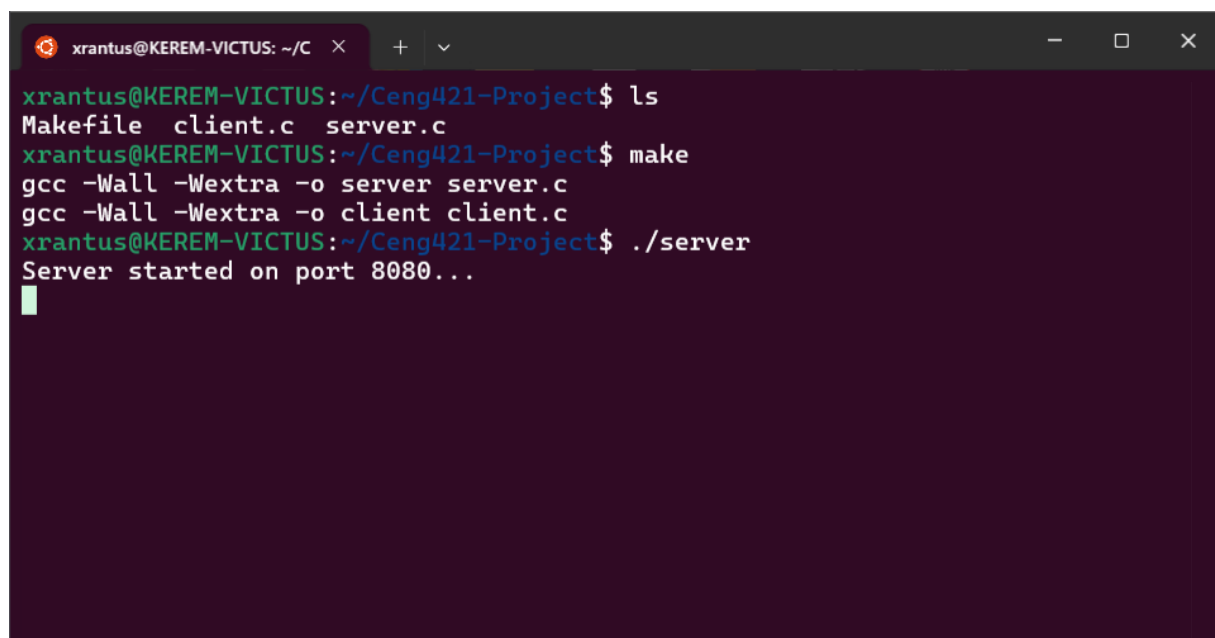
```
xrantus@KEREM-VICTUS: ~/Ceng421-Project$ ls
Makefile  client.c  server.c
xrantus@KEREM-VICTUS: ~/Ceng421-Project$ make
gcc -Wall -Wextra -o server server.c
gcc -Wall -Wextra -o client client.c
xrantus@KEREM-VICTUS: ~/Ceng421-Project$
```

Running make compiles the two executables:

- **server** (from server.c)
- **client** (from client.c)

This confirms the project *compiles under GCC* and produces clean executables on a 64-bit Linux system, as required.

## Starting the Server



```
xrantus@KEREM-VICTUS: ~/C$ ls
Makefile  client.c  server.c
xrantus@KEREM-VICTUS: ~/Ceng421-Project$ make
gcc -Wall -Wextra -o server server.c
gcc -Wall -Wextra -o client client.c
xrantus@KEREM-VICTUS: ~/Ceng421-Project$ ./server
Server started on port 8080...
```

Here we see the newly compiled **server** executable being run.

## First Client Connecting as Admin

```
xrantus@KEREM-VICTUS: ~/C x + v
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ./client kerem
Connecting as: kerem
Welcome kerem! You are the admin.

Available commands:
LIST                - List all files in server
UPLOAD <filename>   - Upload a file to server
DOWNLOAD <filename> - Download a file from server
DELETE <filename>   - Delete a file (admin only)
RENAME <old> <new>  - Rename a file (admin only)
EXIT                - Disconnect from server

kerem> █
```

- The client is started with `./client kerem`. This passes the username "kerem" to the client.
- The server designates this first user as admin, printing "Welcome kerem! You are the admin."
- Admin privileges let this user perform DELETE and RENAME commands.

## Second Client Connecting as Regular User

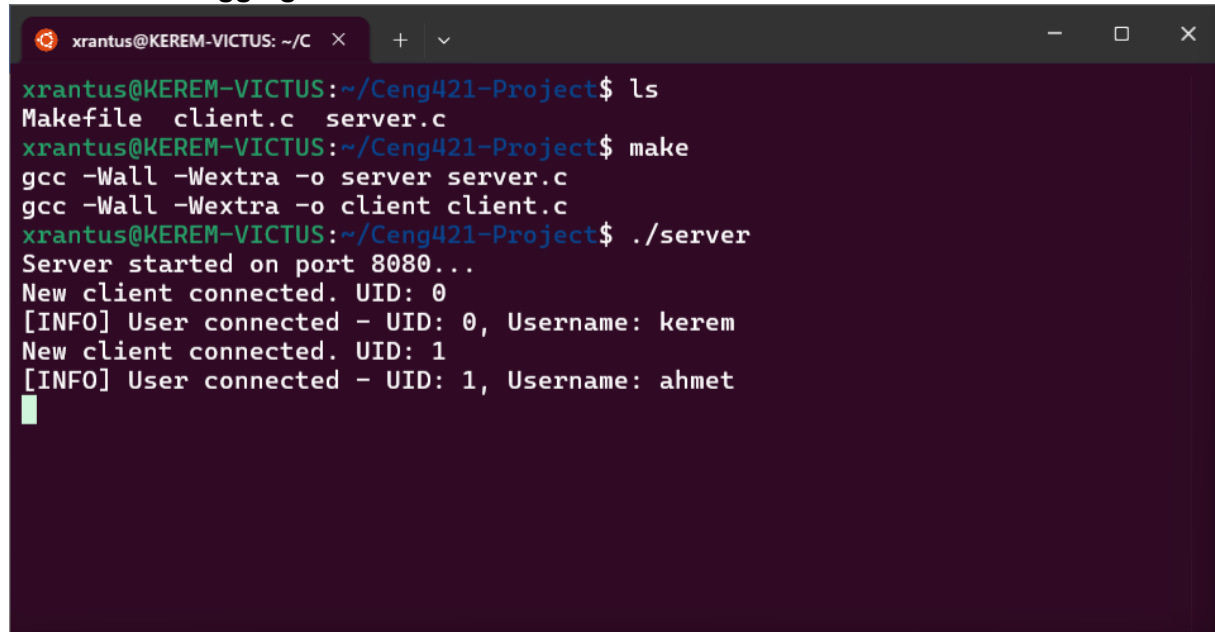
```
xrantus@KEREM-VICTUS: ~/C x + v
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ./client ahmet
Connecting as: ahmet
Welcome ahmet! You are a regular user.

Available commands:
LIST                - List all files in server
UPLOAD <filename>   - Upload a file to server
DOWNLOAD <filename> - Download a file from server
DELETE <filename>   - Delete a file (admin only)
RENAME <old> <new>  - Rename a file (admin only)
EXIT                - Disconnect from server

ahmet> █
```

- A second terminal is opened with `./client ahmet`.
- The server designates this other users as regular, printing "Welcome ahmet! You are a regular user."
- Since there is already one admin (the first client, kerem), new connections are not admins.
- The server sets `is_admin = 0` for any subsequent client after the first.

## Server-Side Logging of Users

A terminal window titled 'xrantus@KEREM-VICTUS: ~/C' with standard window controls. The terminal shows the execution of a C program. The user runs 'ls' showing 'Makefile client.c server.c'. Then 'make' is run, compiling both files with 'gcc -Wall -Wextra'. Finally, './server' is run, starting the server on port 8080. It logs two connections: one for 'kerem' (UID: 0) and one for 'ahmet' (UID: 1).

```
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ls
Makefile  client.c  server.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ make
gcc -Wall -Wextra -o server server.c
gcc -Wall -Wextra -o client client.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ./server
Server started on port 8080...
New client connected. UID: 0
[INFO] User connected - UID: 0, Username: kerem
New client connected. UID: 1
[INFO] User connected - UID: 1, Username: ahmet
```

On the server terminal, we see log messages each time a client connects:

- "UID: 0, Username: kerem"
- "UID: 1, Username: ahmet"

## LIST Command (Empty Directory)

```
client0 x + v
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ./client kerem
Connecting as: kerem
Welcome kerem! You are the admin.

Available commands:
LIST                - List all files in server
UPLOAD <filename>   - Upload a file to server
DOWNLOAD <filename> - Download a file from server
DELETE <filename>   - Delete a file (admin only)
RENAME <old> <new>  - Rename a file (admin only)
EXIT                - Disconnect from server

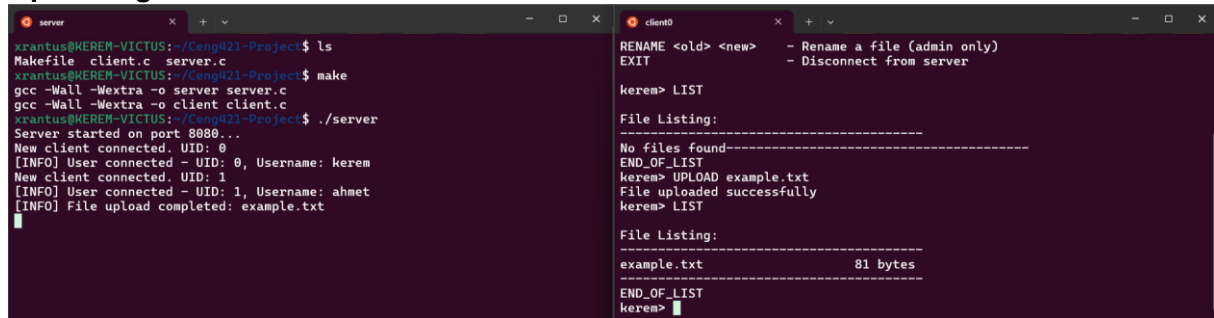
kerem> LIST

File Listing:
-----
No files found-----
END_OF_LIST
kerem> █
```

The admin client types LIST.

- The server responds that no files are found in its ./server\_files directory.
- The LIST command calls list\_files(client\_socket) in **server.c**.
- Inside list\_files(), the server:
  1. Opens the directory FILE\_DIRECTORY (by default ./server\_files).
  2. Reads each file in that directory.
  3. Sends back filenames and sizes to the client.
  4. If the folder is empty, it sends "No files found".

## Uploading a File



```
server
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ls
Makefile client.c server.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ make
gcc -Wall -Wextra -o server server.c
gcc -Wall -Wextra -o client client.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ./server
Server started on port 8888...
New client connected. UID: 0
[INFO] User connected - UID: 0, Username: kerem
New client connected. UID: 1
[INFO] User connected - UID: 1, Username: ahmet
[INFO] File upload completed: example.txt

client0
RENAME <old> <new> - Rename a file (admin only)
EXIT - Disconnect from server

kerem> LIST

File Listing:
-----
No files found-----
END_OF_LIST
kerem> UPLOAD example.txt
File uploaded successfully
kerem> LIST

File Listing:
-----
example.txt 81 bytes
-----
END_OF_LIST
kerem>
```

The admin client uploads a file

### 1. Client side (client.c):

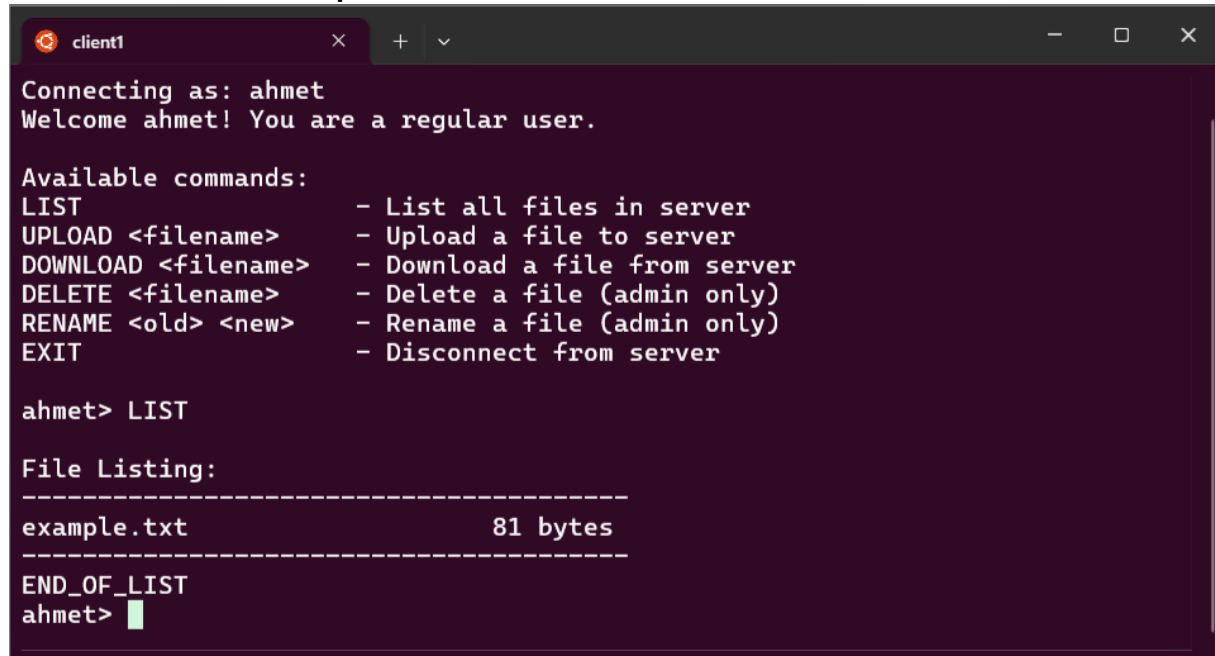
- When user enters UPLOAD example.txt, the client sends "UPLOAD example.txt" to the server.
- Then the client calls handle\_upload(sock, filename):
  - Opens local file example.txt.
  - Sends file data in chunks to the server.
  - Sends END\_OF\_UPLOAD when done.

### 2. Server side (server.c, handle\_upload()):

- Receives "UPLOAD example.txt".
- Creates/opens a file in ./server\_files with the same name.
- Reads incoming chunks into that file until it sees END\_OF\_UPLOAD.
- Closes the file, sends "File uploaded successfully" to client.
- Logs [INFO] File upload completed: example.txt.
- Because files are stored server-side, any client can LIST and see that file.



## Other Client Sees the Uploaded File



A terminal window titled 'client1' with standard window controls. The text inside shows a user 'ahmet' connecting and being welcomed as a regular user. A list of available commands is shown: LIST, UPLOAD, DOWNLOAD, DELETE, RENAME, and EXIT, each with a brief description. The user then enters the 'LIST' command, and the terminal displays a file listing for 'example.txt' which is 81 bytes. The listing is enclosed in dashed lines. The terminal ends with 'END\_OF\_LIST' and a prompt for the user.

```
client1
Connecting as: ahmet
Welcome ahmet! You are a regular user.

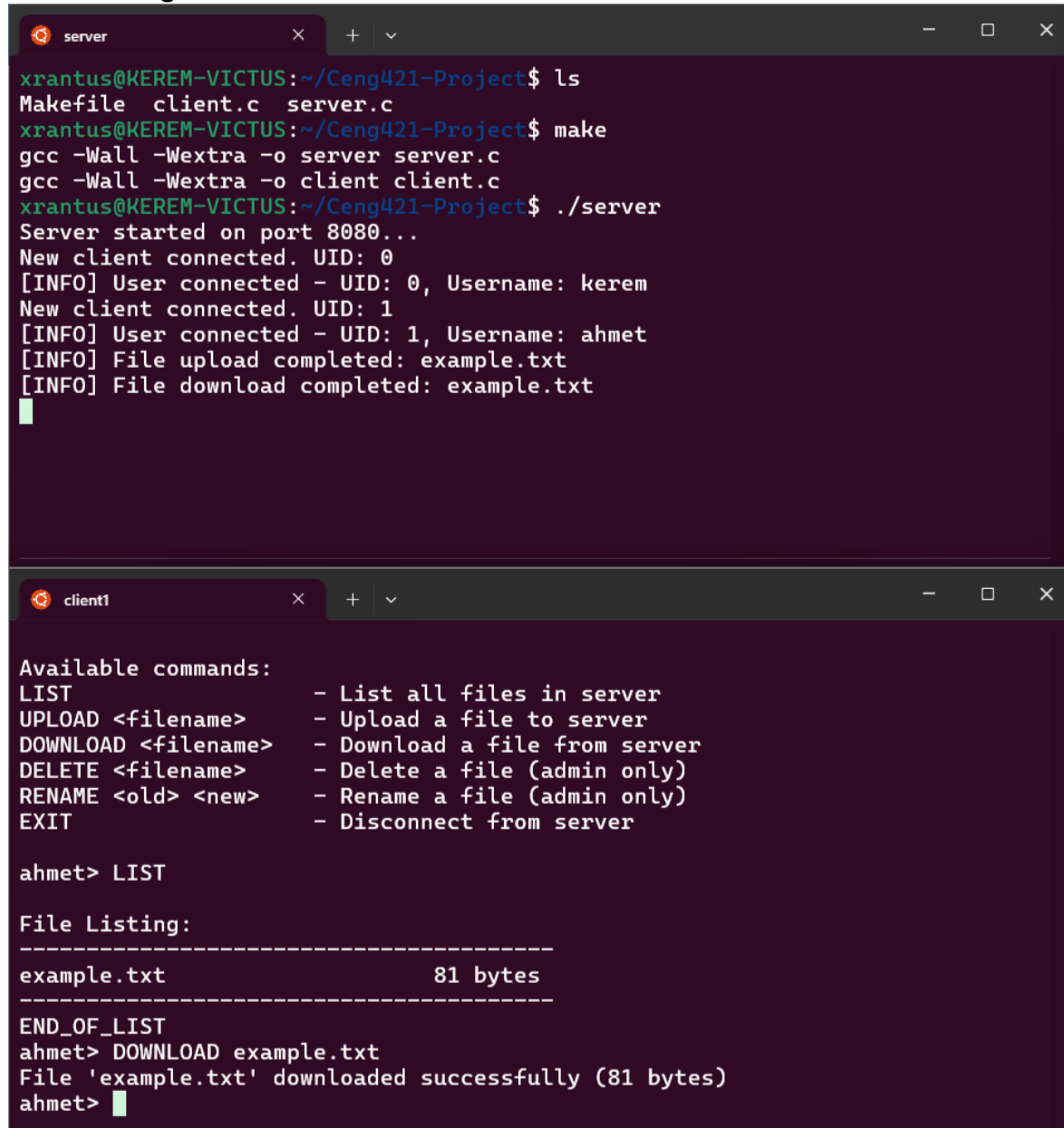
Available commands:
LIST                - List all files in server
UPLOAD <filename>   - Upload a file to server
DOWNLOAD <filename> - Download a file from server
DELETE <filename>   - Delete a file (admin only)
RENAME <old> <new>  - Rename a file (admin only)
EXIT               - Disconnect from server

ahmet> LIST

File Listing:
-----
example.txt                81 bytes
-----
END_OF_LIST
ahmet> █
```

- The regular user is able to see that example.txt was uploaded by the admin.
- The server stores the file in the same shared directory, so *all* connected clients see the same contents when they run LIST.

## Downloading a File



```
server
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ls
Makefile  client.c  server.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ make
gcc -Wall -Wextra -o server server.c
gcc -Wall -Wextra -o client client.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ./server
Server started on port 8080...
New client connected. UID: 0
[INFO] User connected - UID: 0, Username: kerem
New client connected. UID: 1
[INFO] User connected - UID: 1, Username: ahmet
[INFO] File upload completed: example.txt
[INFO] File download completed: example.txt

client1
Available commands:
LIST                - List all files in server
UPLOAD <filename>   - Upload a file to server
DOWNLOAD <filename> - Download a file from server
DELETE <filename>   - Delete a file (admin only)
RENAME <old> <new>  - Rename a file (admin only)
EXIT                - Disconnect from server

ahmet> LIST

File Listing:
-----
example.txt                81 bytes
-----
END_OF_LIST
ahmet> DOWNLOAD example.txt
File 'example.txt' downloaded successfully (81 bytes)
ahmet>
```

The second client downloads the file from the server.

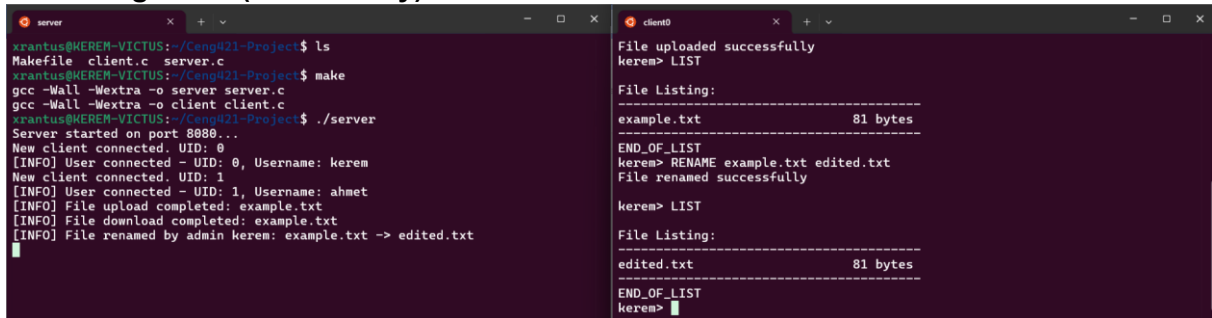
**Client** (client.c, handle\_download()):

- Sends "DOWNLOAD example.txt" to the server.
- Creates a local file example.txt .
- Reads incoming data in chunks until it sees END\_OF\_FILE from the server or an error message.
- Prints success message.

**Server** (server.c, handle\_download()):

- Opens ./server\_files/example.txt.
- Reads the file and sends it in chunks to the client.
- Ends with sending END\_OF\_FILE.

## Renaming a File (Admin Only)



```
server
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ls
Makefile client.c server.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ make
gcc -Wall -Wextra -o server server.c
gcc -Wall -Wextra -o client client.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ./server
Server started on port 8888...
New client connected. UID: 0
[INFO] User connected - UID: 0, Username: kerem
New client connected. UID: 1
[INFO] User connected - UID: 1, Username: ahmet
[INFO] File upload completed: example.txt
[INFO] File download completed: example.txt
[INFO] File renamed by admin kerem: example.txt -> edited.txt

client0
File uploaded successfully
kerem> LIST
File Listing:
-----
example.txt                        81 bytes
-----
END_OF_LIST
kerem> RENAME example.txt edited.txt
File renamed successfully
kerem> LIST
File Listing:
-----
edited.txt                        81 bytes
-----
END_OF_LIST
kerem>
```

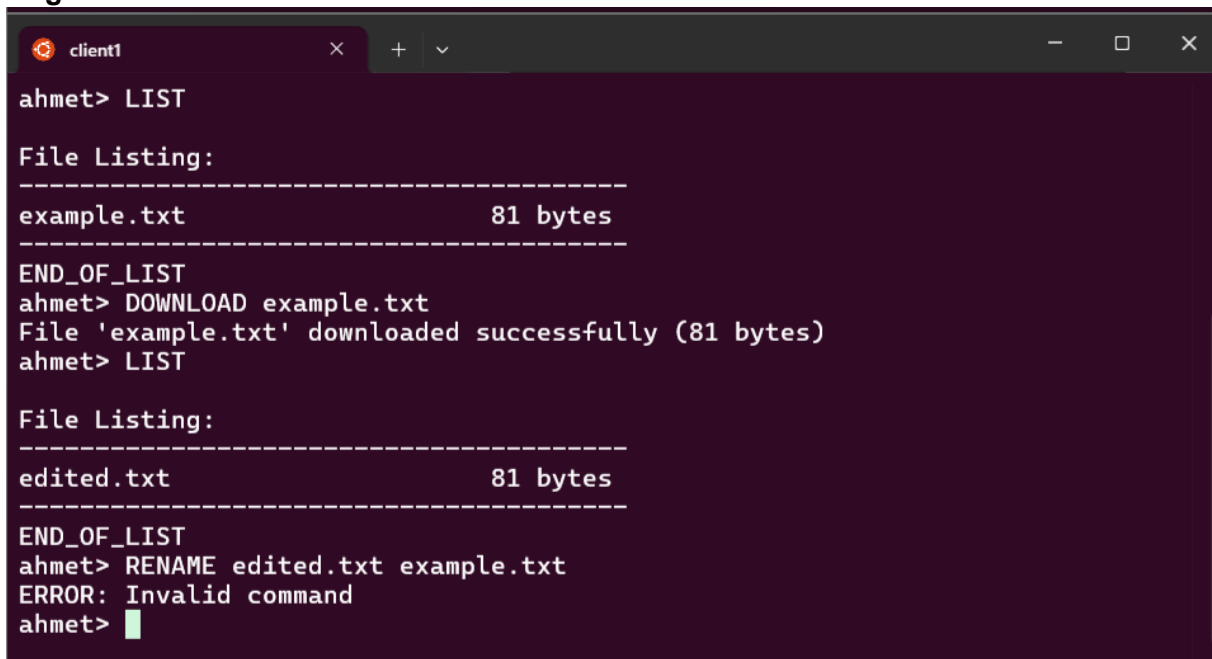
The admin client can rename the file from example.txt to edited.txt.

- The server logs that the file has been renamed by admin.

## Relation to the Code

- In **server.c**, only `is_admin = 1` clients can issue DELETE or RENAME.
- For RENAME, the code:
  - Uses `strtok()` to parse the old name and the new name.
  - Calls the C library function `rename(old_path, new_path)`.
  - Prints the success or error message back to the client.

## Regular Users Cannot Rename



```
client1
ahmet> LIST
File Listing:
-----
example.txt                        81 bytes
-----
END_OF_LIST
ahmet> DOWNLOAD example.txt
File 'example.txt' downloaded successfully (81 bytes)
ahmet> LIST
File Listing:
-----
edited.txt                        81 bytes
-----
END_OF_LIST
ahmet> RENAME edited.txt example.txt
ERROR: Invalid command
ahmet>
```

A regular user tried to rename a file, but the server refused since only admin can rename. The server sends back “ERROR: Invalid command”.

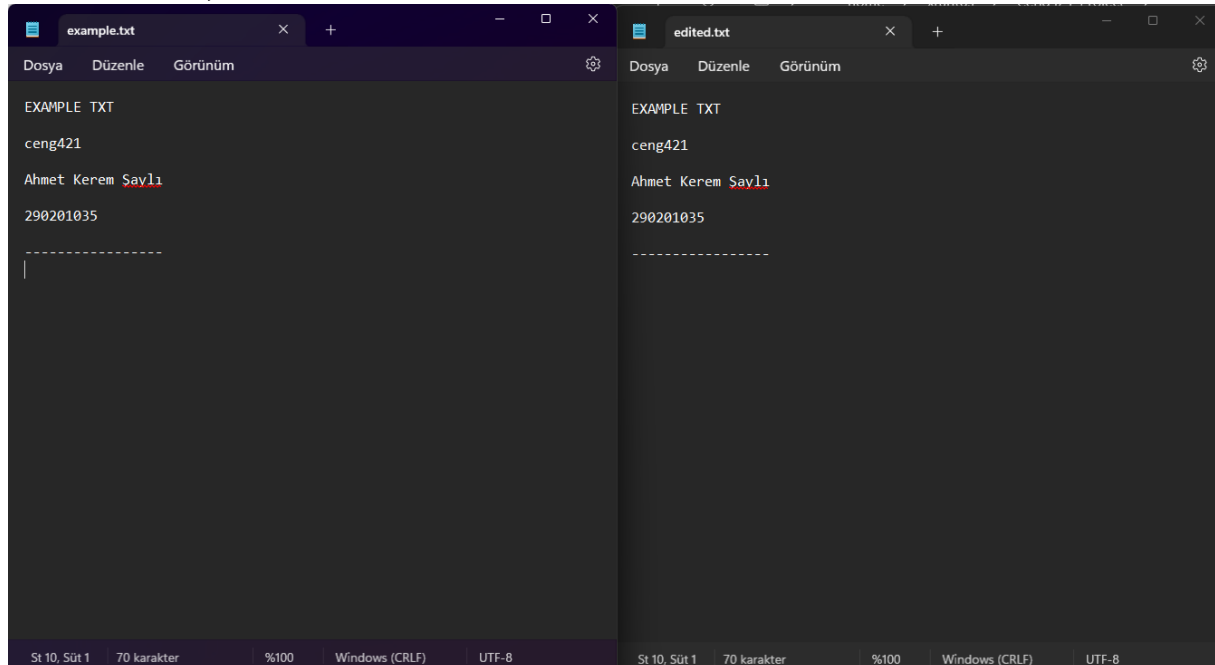
```
server
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ls
Makefile client.c server.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ make
gcc -Wall -Wextra -o server server.c
gcc -Wall -Wextra -o client client.c
xrantus@KEREM-VICTUS:~/Ceng421-Project$ ./server
Server started on port 8080...
New client connected. UID: 0
[INFO] User connected - UID: 0, Username: kerem
New client connected. UID: 1
[INFO] User connected - UID: 1, Username: ahmet
[INFO] File upload completed: example.txt
[INFO] File download completed: example.txt
[INFO] File renamed by admin kerem: example.txt -> edited.txt
[INFO] File download completed: edited.txt
█

client1
File Listing:
-----
example.txt                81 bytes
-----
END_OF_LIST
ahmet> DOWNLOAD example.txt
File 'example.txt' downloaded successfully (81 bytes)
ahmet> LIST

File Listing:
-----
edited.txt                 81 bytes
-----
END_OF_LIST
ahmet> RENAME edited.txt example.txt
ERROR: Invalid command
ahmet> DOWNLOAD edited.txt
File 'edited.txt' downloaded successfully (81 bytes)
ahmet> █
```

Even though the user cannot rename, they *can* still list and download the newly renamed file.

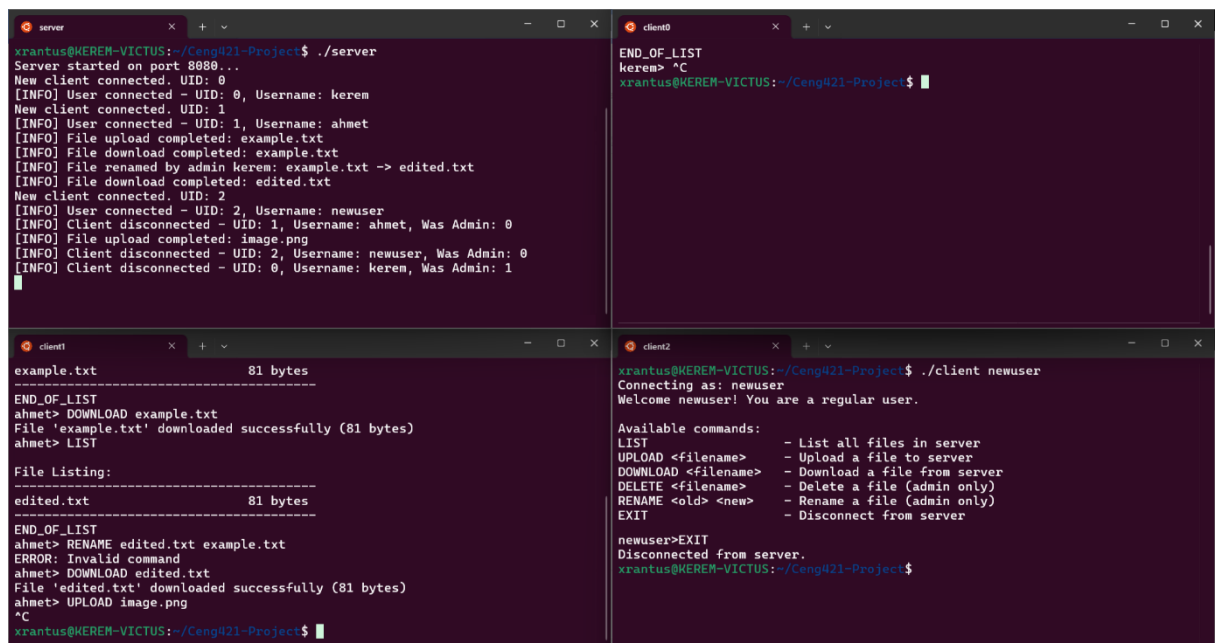
## Same Content, Different Name



example.txt and edited.txt Both contain the same text, just with a different filename.

Renaming does not change the file contents; it only changes the filename on the server.

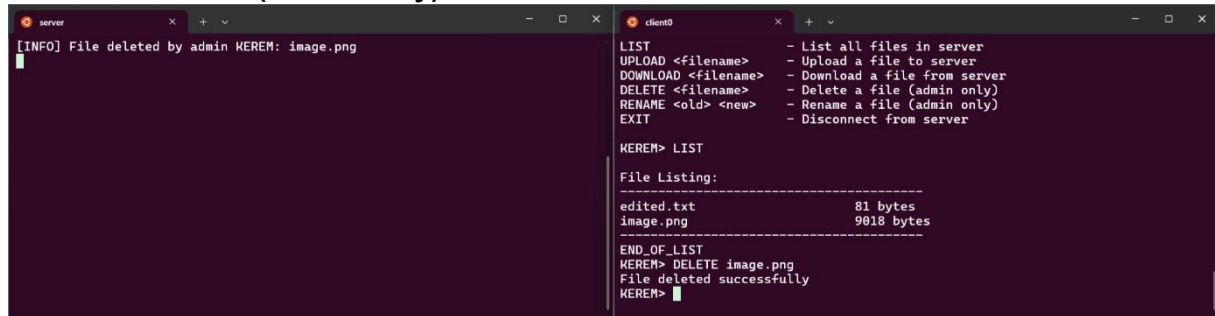
## Exit Command & CTRL+C Handling



A client can cleanly exit by typing EXIT. If they do a Ctrl+C in the client terminal, the connection also ends. The server logs that the user disconnected.

If the disconnecting user was admin, the server attempts to make the next available connected user the new admin.

## Delete Command (Admin Only)



```
server x + v - □ x client0 x + v - □ x
[INFO] File deleted by admin KEREM: image.png
█

LIST                                - List all files in server
UPLOAD <filename>                  - Upload a file to server
DOWNLOAD <filename>                 - Download a file from server
DELETE <filename>                   - Delete a file (admin only)
RENAME <old> <new>                  - Rename a file (admin only)
EXIT                                - Disconnect from server

KEREM> LIST

File Listing:
-----
edited.txt                        81 bytes
image.png                        9818 bytes
-----

END_OF_LIST
KEREM> DELETE image.png
File deleted successfully
KEREM> █
```

The admin can remove files from server\_files. Other clients see the file vanish from their subsequent LIST.

## 4. Challenges Encountered

### 1. Multi-Client Concurrency

- Using `fork()` for each client simplifies concurrency but requires careful resource management (e.g., file descriptors). We handled this by tracking clients in a global array and ensuring each child process calls `handle_client()` and then exits.

### 2. Ensuring Admin-Only Commands

- The project needed a way to **restrict** certain operations to the admin user only. We added a simple `is_admin` check in the server's command processing. If `is_admin` is 0, `DELETE` and `RENAME` are rejected.

### 3. File Transfer Reliability

- We made sure to send a special marker (`END_OF_UPLOAD` or `END_OF_FILE`) to know when uploads/downloads are complete. This ensures partial or truncated transfers are detected.

### 4. Graceful Disconnect

- Properly handling both `EXIT` commands and unexpected disconnects required us to implement `remove_client(uid)` on the server side, ensuring if the departing user was admin, admin rights transfer to another client.

### 5. Research and References

- Primarily, we referred to standard **manual pages** for sockets (`man socket`, `man bind`, `man listen`, etc.) and for system calls like `fork()`, `stat()`, `opendir()`, and `rename()`.
- No code was directly copied from external sources. All logic and structures are self-written.

## 5. Conclusion

In summary, this C-based client-server application illustrates **multi-client file sharing** over TCP sockets. The design ensures:

- **Core Features** like listing files, uploading, downloading, and admin-exclusive rename/delete commands.
- **Privilege Control** by assigning admin to the first connected client.
- **Concurrency** via `fork()`, allowing multiple clients to interact simultaneously.
- **Simplicity** and compliance with course guidelines: no external libraries, straightforward Makefile-based compilation, and only standard system calls.

By following the **How to Compile and Run** instructions and referencing the screenshots, you can see how each part of the system functions. This satisfies all project requirements while providing a clear demonstration of network programming, file I/O, and basic concurrency in C.