## horizontal line

Retro-Cloud

An emulation game streaming solution

Student Name: Tomer Wiesel

ID Number: 216156166

Teacher: Yoram

Submission Date: 01.06.2025

# Table of Contents

**Overview2**

Emulation 101 2

Emulation vs Virtualization 3

A little bit about my project 3

Current state of the market 4

Target demographic 4

Future Updates 4

Project Roadmap 5

**Project Environment6**

IDEs and Languages 6

Installation Guide 7

**System Requirements and Architecture 8**

Use Cases 8

Server Requirements 8

Ecosystem 8

Core Modules 9

Protocol 17

Encryption 18

Threats/Vulnerabilities 19

**GUI 20**

**Functions/Classes and their purpose 26**

**Special Code Segments 26**

**Reflection 33**

**Bibliography 35**

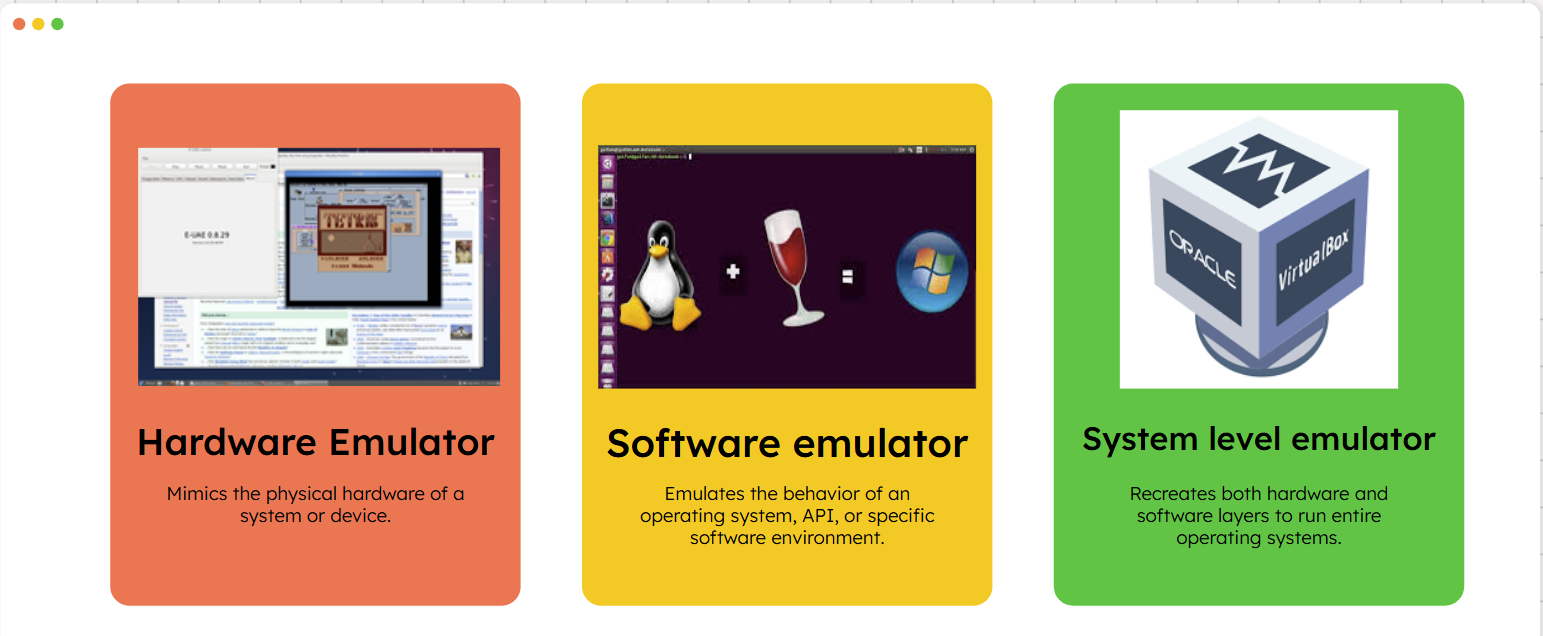
**Overview:**

Retro cloud, is a client-server based application, that allows the user to quickly and conveniently play their favorite retro games, without having to download any of them. This project primarily deals in emulation. Before we can talk about my project and how it works, we must first understand this field in computer science.

Emulation 101:

Emulation is the process of enabling one system, often called a host, to mimic another, often called a guest. This allows the host system to run software and or perform tasks meant for the guest system. This is done through pieces of software or hardware called Emulators, which, through knowledge gained by reverse engineering, simulate the functionality and operating conditions of the guest system. There are several different methods of emulating and as a result, many different kinds of emulators.

Typically, there are three main kinds of emulators:



* **Hardware emulators, which focus on mimicking the physical components of a system. This is done through the simulation or even the recreation of the circuitry.**
* **Software emulators, which purely implement other kinds of software in a way that is understandable to the host system.**
* **System level emulators, which are a kind of collaboration of the former two types.**

Most emulators usually have some form of compatibility layer, that comes in between the host and guest system, allowing one to interact with the other. This is called the Hardware abstraction layer, or HAL for short. The HAL, allows the user to, for example, provide input to the emulator through peripherals such as keyboards, mice and or controllers, while in turn, the giving the emulator the ability to display graphics and output audio.

Game data: compiled/assembled for Gameboy hardware

Emulator: Translates data to be processable by modern architecture

HAL

PC resources and hardware

PC IO: keyboard, graphics, speakers

**Before we continue I would like to bring up a subject that can be the cause of confusion in the discussion on emulation, and that is the topic of Virtualization.**

Emulation vs. Virtualization:

Emulation refers to **any system that recreates the functionality of one platform on a different platform**, typically by **fully imitating hardware, software, or both**, often at the instruction level. Emulation often "fakes" both hardware and software, and does not rely on the host CPU to match the guest CPU architecture. Virtualization is **a specialized form of emulation** where the guest and host systems share the **same CPU architecture**, and certain hardware features are passed through to the guest OS for better performance.

Instead of fully "pretending" to be another system, virtualization sets up a virtual machine that runs alongside the host OS.

A little bit about my project:

**Retro cloud**, as I touched on earlier, is a client-server based app, which lets you emulate retro games, specifically original Gameboy games. The client, runs a gameboy emulator which I wrote, while the server provides the games. This way, the client doesn’t have to go searching through vast internet archives for the digitalized copies of the games they want to play. The twist however, is that the clients don’t actually have to download the games. When the user plays a game, the game data is sent by the server to the client, which stores it not on the drive but in memory for the duration of the playthrough (this only works while the client is connected to the server). Essentially, my app allows the user to “stream” gameboy games.

I decided to make this project as I eagerly desired to learn more about low level programming and because the gameboy has always held a special pace in my heart. When I was younger my mom gave me her original gameboy, and I’ve been playing Tetris and Mario on it ever since. I also just love emulators in general, and often enjoy playing nintedo classics on one of my raspberry pi computers, set up to run retropie(a raspberry pi image/os built for videogame console emulation)

Current state of the market:

Most emulators (both downloaded and online) involve the manual setup of libraries, which require you to find the game files, or ROMS(R.O.M is short for Read Only Memory) by yourself on archives or less than trustworthy and often suspicious websites.

I often found the task of setting up a game library to be quite tedious, and wanted to simplify the process. This is why I chose the concept of “game streaming” to be the focus of the networking side of the project.

Target Demographic:

This project is intended for people, like myself, who just generally enjoy retro games, and don’t require very specific setups of certain emulators in specific configurations with a game library made for their preference. This app is meant to streamline the setup process and make it so that you can be ready to play pretty much out of the box.

Future Updates:

I think that if I were to continue expanding this project, I would add additional emulators, and improve the already existing Gameboy emulator, allowing it to support a larger catalogue of games: See MBCs section of the [Gameboy-Pandocs](https://gbdev.io/pandocs/MBCs.html) .

# 

**Project Roadmap:**

|  |  |
| --- | --- |
| **Version(W-windows/L-Linux)** | **Features** |
| 0L | Proof of Concept, basic emulator of the CHIP8 interpreted programming language. |
| 0.5L | Application Skeleton for graphics and menus |
| 0.75L | Gameboy Emulator |
| 1L | Single Client server and emulator with game streaming |
| 1.25L | Updating game list and game info pages |
| 1.5L | Account database with Sqlite3, login, registration |
| 2L | Synchronous Multiclient server with connected graphics, account database and game streaming |
| 2.25L | Port Forwarding compatibility with ngrok and SQL injection prevention |
| 2.5W | Windows port of the project |
| 2.75W | Encryption with OpenSSL(doesn’t work with port forwarding) |
| 3W | Gameboy Emulator V2 |
| 3.25W | Rate Limiting and Timeout handling |
| 3.5W | Conversion of the backend and client to be Asynchronous |
| 4W | Turned into Executable with Auto-Py-To-Exe |

**Project Environment:**

Languages:

The project was programmed in 3 languages. C was chosen for it’s speed and memory control as the language that the emulator would be written in. I chose Python for the frontend and backend as I was most familiar with it, for it’s simplicity in the development of online applications and for it’s diverse and intuitive graphics libraries such as PyQt.

IDEs’ and libraries:

I wrote the project in 2 IDEs’, Visual studio code for the emulator, and PyCharm for the app and server. I initially made the project in Linux due to the simplicity of using package managers to connect all the necessary libraries, but eventually ported it to Windows for the user’s convenience.

I utilized several tools and libraries in the process of making the project, each of which helped in it’s development:

* Qt designer: An app which lets you design app graphics for the PyQt library and converts them to code compatible files that integrate straight into the IDE.
* Cygwin: A CLI(command line interface) which holds a collection of open source packages and helps with compiling Linux projects for Windows.
* SDL: Hardware abstraction library. Allows the emulator to display graphics, output audio, take keyboard input
* PyQT: Graphics library for the client
* OpenSSL: Provided a layer of encryption to secure communication between the client and server
* Asyncio: Facilitates asynchronous code(helped my graphics run more smoothly)
* Auto-Py-To-EXE: Converted the client files and folder into one easy to use executable
* DB browser and Sqlite (App and library): Provides a simple and interactive way to integrate databases into the project

**Install Guide:(Windows Only)**

Server Host:

1. Visit (Itch io page for the project)
2. Click install server
3. Populate ServerData/roms folder with roms(the server owner does need their own roms)
4. Take the shortcut to your desktop
5. Launch server

Client:

1. Visit (Itch io page for the project)
2. Click install client
3. Take the shortcut to your desktop
4. Enjoy

# 

**System requirements and Project Architecture:**

Use cases:

The main use case for this project, is to streamline the setup of emulation game libraries. The server handles the game library, that way it only has to be setup once. This allows all clients to enjoy the content without having to deal with the tedious setup of the ROM library.

Server Requirements:

**Concurrent gameplay:**

Retro cloud’s server is capable of handling multiple users playing different games, at the same time. This was achieved through the following means:

1. The server was built to run asynchronously (using libraries like asyncio, aiosqulite, aiofiles and qasync), to improve responsiveness, parallelism and resource utilization.

2. Emulation is done on the client’s side, so all the handling of gameplay is done without any expense to the server.

3. Retro game files are small. The largest Gameboy cartridge/file is at most 2-2.5MB, with most games being between 32KB-100KB. This means the server doesn’t have to send mass amounts of data, every time a user wants to play a game.

**Streamed Games:**

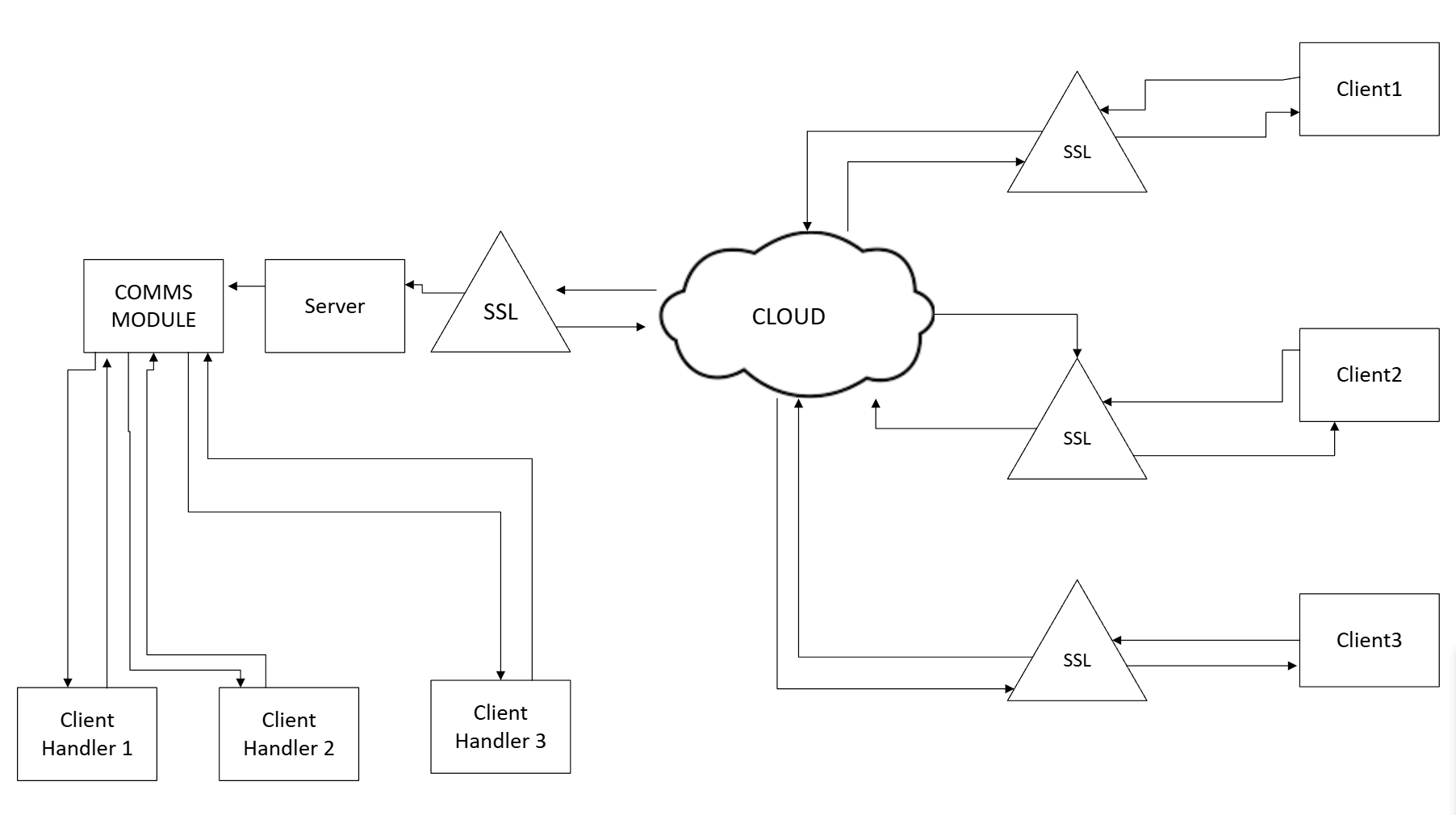
The server’s purpose is to host the games that its clients want to play. There’s a catch though, the clients don’t actually get to “keep” the games. Had it been this way, the server would run out of users quite quickly as they would simply download the games and leave. To prevent this, the server employs two tactics:

1. The games are sent to the client, but rather than be saved into a file, they are injected into ram, directly following decryption. This is made possible because the app doesn’t use a traditional emulator but rather a custom one I wrote, which supports running games from stdin during execution.

2. The server encrypts the incoming data, so that the client can’t peek at it at any point(through packet sniffing for example) until the data is already copied into ram and discarded. Essentially, the clients have no way to interact with the game data, and are only able to request it and play it after it's processed.

Ecosystem:

The client and server use an SSL certificate in order to verify identity and encrypt the traffic being sent between them over the network (they use TCP). Additionally, the server utilizes an Sqlite3 database to store user info. The exchanges between server and client begin once the client inputs the server’s ip address into the server pin field in the main menu and hits the connect button. The client makes requests to the server (such as login or gameplay requests), while the server then reacts to these requests with the proper response as declared in the communication protocol between them.



Core modules in the project:

The code for the project is spread out through multiple files, classes, and languages. At the time of execution, it is the cooperation between these various regions or “modules” of the project, being handled through a series of exchanges, done according to a protocol, that allows it to function.

**Server Modules:**

The server is comprised of 4 core modules: Comms(Handler), Assigner, Game resources and finally Auth(DB and Encryption).

A diagram of a software system

AI-generated content may be incorrect.

The Auth module: The Auth module is comprised of two parts. The first is the Encryption layer which is established between the client and server upon a successful exchange of Asymmetric keys using an SSL certificate(See Encryption section later in the chapter). The second is the User Account database. When a client wants to login, logout, or register, the server utilizes the account database to either create new data, update existing data or verify it.

A diagram of a software

AI-generated content may be incorrect.

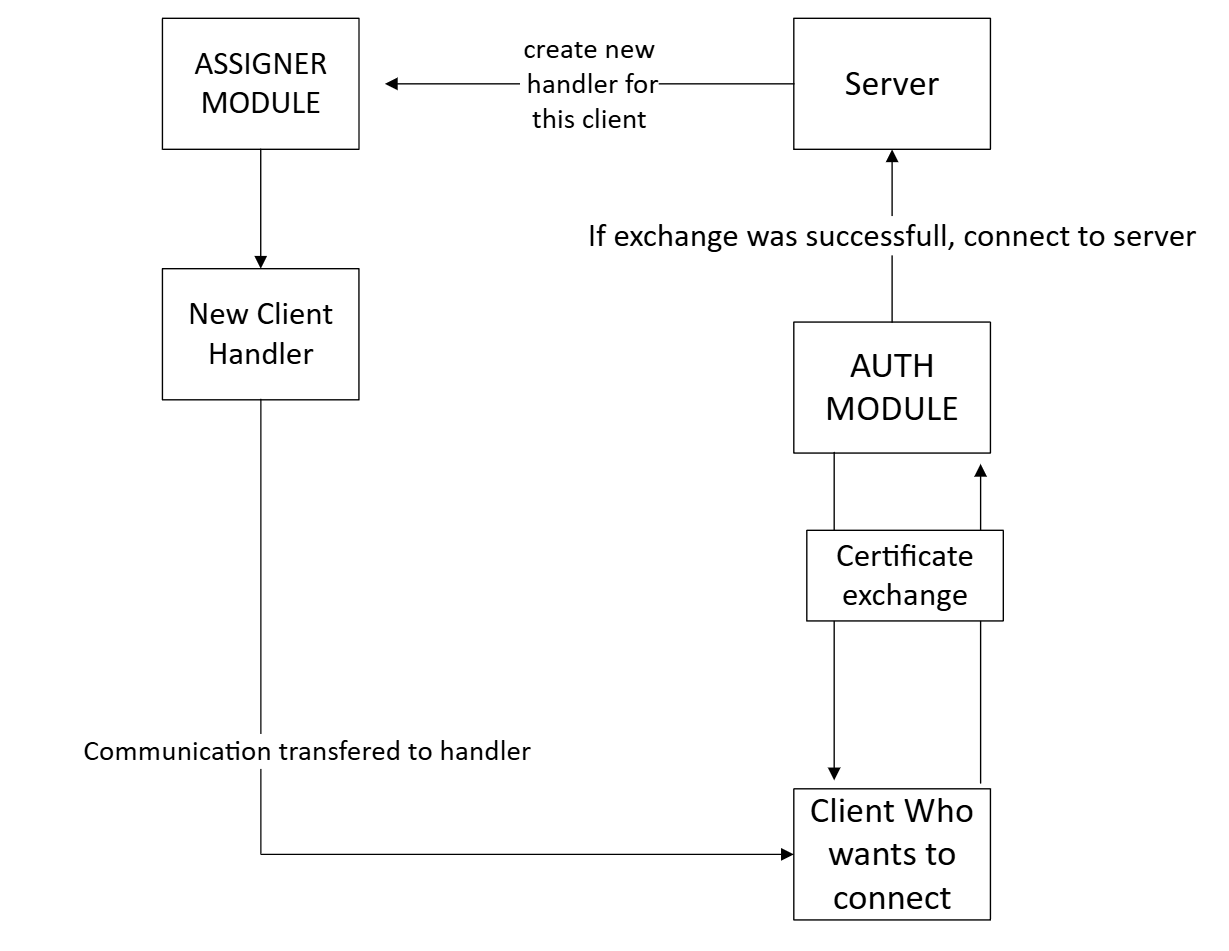
**\*Flowchart of the Encryption section of the AUTH module**

A diagram of a software system

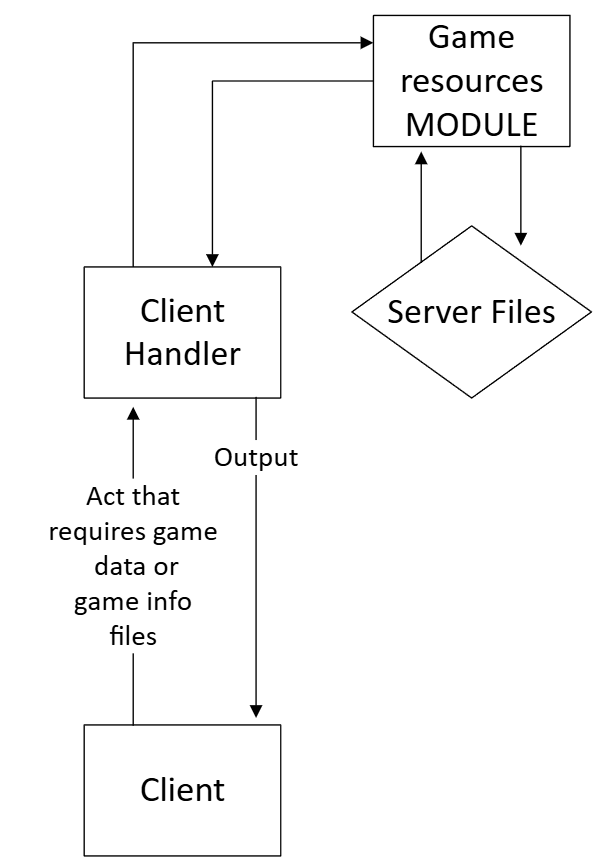
AI-generated content may be incorrect.

**\*Flowchart of the Database section of the AUTH module**

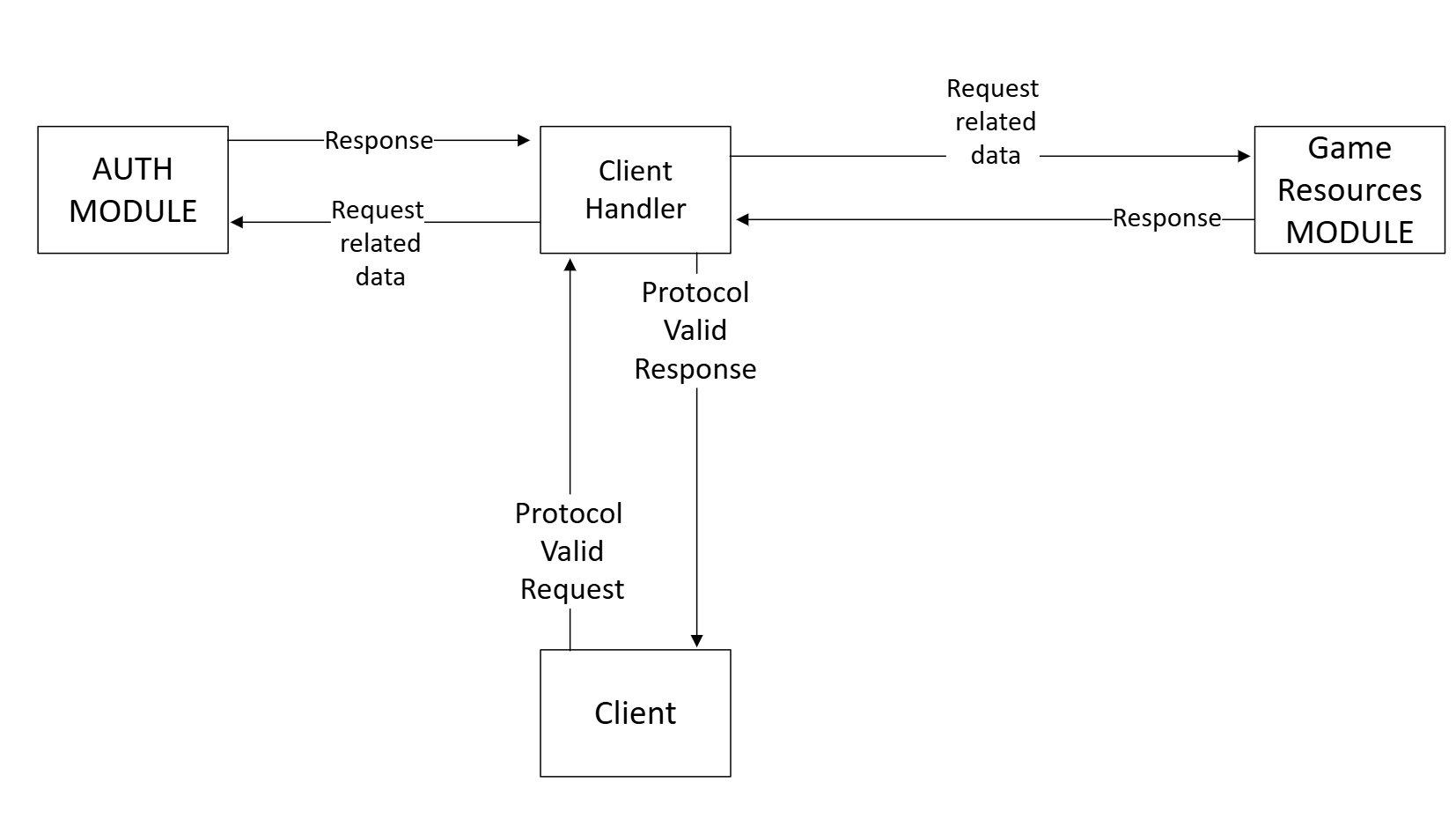
The Assigner module: The assigner module is very minimal yet plays an important role in the server’s operation. When a new client connects to the server, it is the job of the assigner to establish a proper connection between the two, and assig the client a handler.

****

The Game Resources module: This module is responsible for managing the server’s ROMS and game info files. When a client plays a game, or views a game’s info page, it is the data from this module that is sent to the client.

****

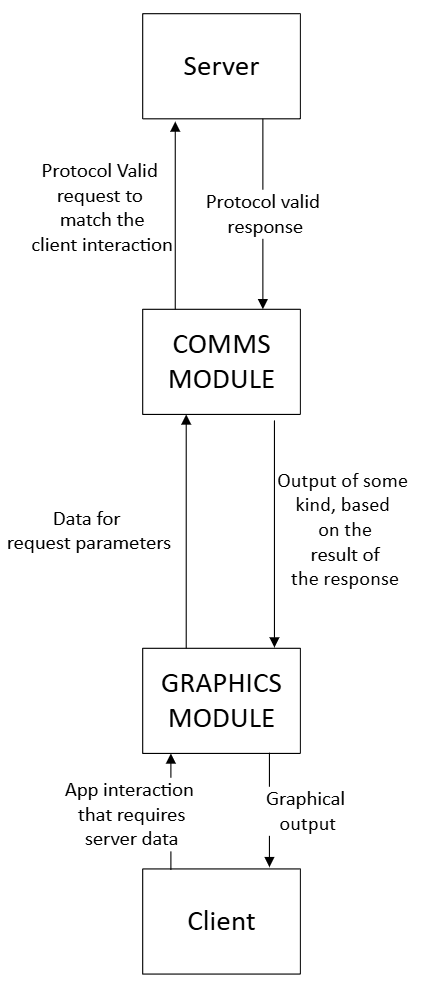
The Comms module: The comms module is responsible for handling the individual client. After the client is assigned a handler, communication with a client is managed by a thread of this module. Exchanges between client and server are done according to a protocol, which instructs both client and server how to request or react in different ways based on the operation that is attempted.

****

**Client Modules:**

The client is comprised of 3 core modules: Comms, Graphics, and Emulation.

The Comms module: The “comms”, or communication module, is responsible for handling and guiding any interaction with the server and forwarding the output/result to the app via the other modules. All packets sent to the server go through the communication module, to make sure that they are protocol valid.



The Graphics module: This is the module that handles all the app’s GUI. The module uses the library PyQt5, and is elaborated upon in the GUI chapter of the book.

The Emulation module: The emulation module is made up of the emulator itself, and the pipeline through which the game data is passed to it. This module runs the games.

A diagram of a computer program

AI-generated content may be incorrect.

Protocol:

Any data sent between the client and server, is done according to a protocol. All requests are initiated by the client, and all responses are provided by the server.

|  |  |
| --- | --- |
| **Client Request** | **Server Operation and output** |
| REG|username|password | Attempts to register new user. If successful, logs the new user in and outputs, “User created successfully” |
| LOG|username|password | Attempts to register new user. If successful, logs the new user in and outputs, “User Logged in successfully” |
| QUITGAME | If the user was logged in, logs them out. The server removes them from the current user list. |
| GAME|game\_name | The server sends the game data of the game titled game\_name |
| INFO|game\_name | The server sends the game info files of the game titled game\_name |
| BACK|window\_name | The server acknowledges that the client is moving back to a previous window, and based on the window, sends any relevant data. |
| MAN | Sends the game library manifest |
| check | Just a ping, server does not send anything back. This is meant to check that the client is still connected while playing a game |

|  |  |
| --- | --- |
| **Server Message** | **Client Response** |
| User created Successfully | Updates username on all windows and sends the manifest request |
| User Logged in Successfully | Updates username on all windows and sends the manifest request |
| Online game list | Updates the game selection screen based on manifest data |
| BAN | Disconnects and goes to main menu |
| INFO | Updates game info window with game info data |
| GAMEDATA | Starts the emulator and feeds it the game data over stdin |

Encryption:

In order to protect both user data, and game data, the server encrypts both incoming and outgoing traffic. The app uses an SSL certificate and the ssl library in Python.

An SSL certificate enables encrypted communication between a client and a server using the TLS (Transport Layer Security) protocol. This encryption begins with an asymmetric process in which the client receives the server's public key through the SSL certificate. The client then uses this public key to securely send a randomly generated symmetric session key to the server. Once both parties have the symmetric key, they switch to fast and efficient symmetric encryption for the remainder of the session. This hybrid approach combines the security of asymmetric encryption with the speed of symmetric encryption. SSL certificates also help protect against man-in-the-middle (MITM, also shown below in the picture) attacks by providing a way for clients to verify the identity of the server. The server's SSL certificate is signed by a trusted Certificate Authority (CA), and the client checks this signature to ensure it is connecting to the legitimate server. If the certificate is invalid or tampered with, the connection is blocked or flagged, preventing attackers from impersonating the server. Overall, SSL certificates are essential for securing data in transit, ensuring authenticity, and safely establishing encrypted channels using a combination of asymmetric and symmetric cryptography.

for more information on SSL: [Explanation video](https://www.google.com/search?q=digital+certificate+explanation+bind+onwner+to+their+public+key&sca_esv=3f7494e869ebe89e&rlz=1C1GCEU_enIL1161IL1161&udm=7&biw=1258&bih=702&sxsrf=AHTn8zo_8X7avMU2ljv7P7KG4Vu8LVm58A%3A1747912218718&ei=GgYvaLfHK7aVxc8PmIyriQg&oq=digital+certificate+explanation+bind+onwner+to+their+public&gs_lp=EhZnd3Mtd2l6LW1vZGVsZXNzLXZpZGVvIjtkaWdpdGFsIGNlcnRpZmljYXRlIGV4cGxhbmF0aW9uIGJpbmQgb253bmVyIHRvIHRoZWlyIHB1YmxpYyoCCAEyBxAhGKABGAoyBxAhGKABGAoyBxAhGKABGApI0J4BUL0IWNCOAXADeAGQAQCYAfQBoAGoKqoBBjAuMjcuM7gBA8gBAPgBAZgCIaACwCvCAgoQABiwAxjWBBhHwgIGEAAYFhgewgILEAAYgAQYhgMYigXCAggQABiABBiiBMICBRAAGO8FwgIFECEYoAHCAgUQIRifBcICBBAhGBXCAgYQIRgVGAqYAwCIBgGQBgOSBwYzLjI1LjWgB_G9AbIHBjAuMjUuNbgHsivCBwcwLjEzLjIwyAdo&sclient=gws-wiz-modeless-video#fpstate=ive&vld=cid:0943d890,vid:5rT6fZUwhG8,st:0)(The graphic to the left is also from that video)

A diagram of a computer

AI-generated content may be incorrect.

Threats and vulnerabilities:

During operation, the server needs to be able to handle a variety of situations. Furthermore, it needs to be capable of protecting itself from data breaches, and information overflow. The following table describes the server’s use cases and how it handles each one.

|  |  |
| --- | --- |
| Threat | Handling process |
| Rate abuse | In order to prevent an overflow of data to the server, and to serve as very rudimentary DOS/DDOS protection, the server enforces a packet/request rate per time period of data per user. If a user surpasses this limit, then they will be temporarily banned from the server. |
| SQL Injection | In order to prevent data leaks, the server prevents SQL injections by using parameterized queries, and by storing the database info in a manner that is never presented to the client. |
| Man in the middle | Encrypted data and client/server identification using SSL Certificate |

Important files/classes/functions:

**Emulator(All Functions here belong to the Emulation Module)**

|  |  |
| --- | --- |
| cpu\_cycle | Reads Rom file at the address of the program counter and calls the read opcode’s matching emulated function, then updates cycle counters and the program counter. |
| update\_timers | Progresses timer registers and checks whether to enable timer interrupt |
| increment\_scan\_line | Essentially a graphics “cycle”. Loads graphics data from VRAM, renders it on screen (and triggers a certain graphics interrupt) if the proper conditions are met, otherwise it progresses the frame drawing. |
| check\_interrupts | Checks the interrupt registers and executes any set interrupts |
| load\_rom | Reads rom file from a given path and loads it into memory, then assigns a pointer to the memory block, which will be accessible to the emulator. This was used for testing purposes before the game streaming was implemented. |
| direct\_load\_rom | Loads rom file data from stdin, then assigns a pointer to the memory block, which will be accessible to the emulator. |
| init\_HAL | Initializes SDL and the necessary modules it needs in order to display video, take keyboard input and output audio. |
| shutdown\_emu | Closes the SDL modules and terminates execution |
| main | Calls all the startup functions (init\_HAL and others), runs the mainloop (the first 4 functions in this table) and handles shutdown by calling shutdown\_emu and freeing any memory blocks taken up during runtime. |

**App:**

**Client: asyncClient.py**

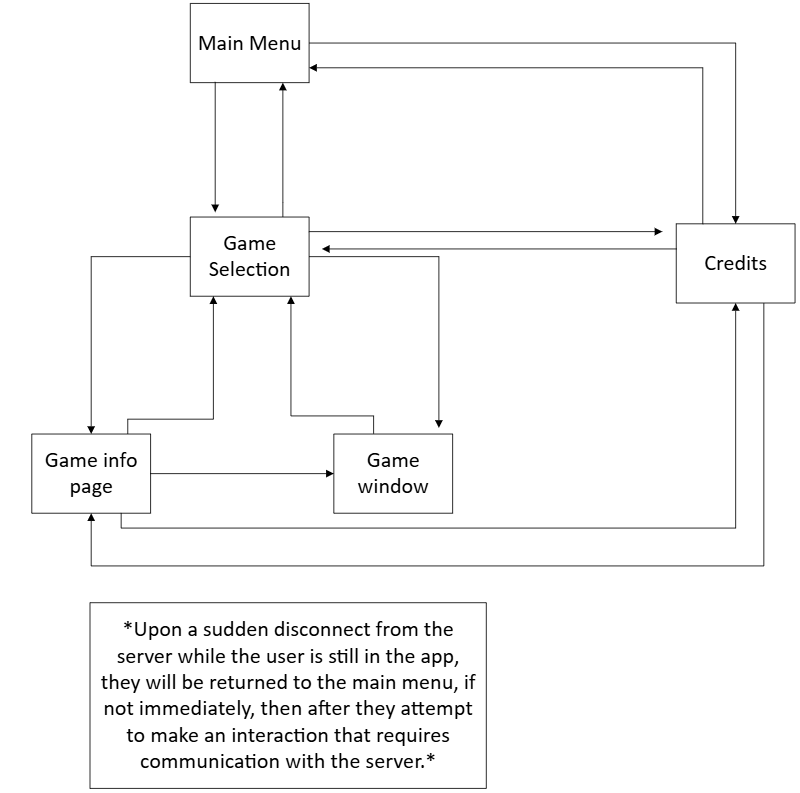
|  |  |  |  |
| --- | --- | --- | --- |
| Data type | Name | Purpose and notable functions | Module |
| Class | MenuWindow | Sets up the main menu window from the UI file and serves as it’s endpoint in the app. From this window, the user will connect to the server, register an account and/or login.  Some important functions in this class:  -**pin\_connect:** calls the start\_client function  -**register:** Prepares a register data packet to be sent to the server based on input in the username and password fields  -**login:** Prepares a login data packet to be sent to the server based on input in the username and password fields | GRAPHICS |
| Class | GameSelectWindow | Sets up and serves as the endpoint for the game selection window. The purpose of this window, is to display all the available games the server has at the moment, and allow the player to either play them or learn more about them.  Some noteworthy functions in this class include:  -**update\_game\_frames**:  Dynamically creates rectangular “frames” based on the games the server has, as is provided to the client by the server(in an external function). Each frame contains the game name, and two buttons, one for starting the game, and one for navigating to the game’s info page.  -**start\_game**:  Requests the game data of the game file matching the name in the “frame” from which the function was called, upon the pressing of the play button.  -**info\_game**:  Requests the info and art files of the game matching the name in the “frame” from which the function was called, upon the pressing of the info button. | GRAPHICS |
| Class | GameInfoWindow | Displays a more descriptive page for a particular game in the game select page. It contains the game name, release date, a short summary about the game and finally a cover art image of the game. One can also start playing the game from this page, by clicking the play now button(this triggers the start\_game function from the game selection screen)  The core functions of this class are:  -**update\_game\_info**:  This function receives all that which the window will display about a particular game, and load it onto the window. It then shows the game window. | GRAPHICS |
| Class | CreditsWindow | This window displays the sources I used to construct the project, later referenced here in the Bibliography section. The only core function it has is the back function, which returns you to the window from which you opened the credits window. | GRAPHICS |
| Function | async\_send\_message | Puts together a message packet and sends it to the server. Upon the occurrence of an exception in this function, the app can know if it was disconnected from the server. | COMMS |
| Function | async\_recvall | Used when receiving a message from the server. Constructs message packet to be decrypted and read. | COMMS |
| Function | start\_client | Attempts to connect to server and sets up the client’s reader and writer with asyncio.open\_connection. It then starts the “mainloop” function which is receive\_messages. | COMMS |
| Function | receive\_messages | Runs while the client is connected to the server. Parses the received messages from the server and reacts to them based on their content. | COMMS |
| Function | Main | Sets up all the windows with preliminary data and hides all of them but the main menu, which it displays. | GRAPHICS |
| Function | discon\_reset | Returns the app to default status upon a disconnect from the server. | COMMS |
| Function | In\_game | Runs while the client is playing a game.  Injects the game data into the emulator so that it can run games, and keeps the client in “gameloop”. Upon the game being closed. Returns to the standard mainloop.  Gameloop is a state where the client is running the emulator, all other windows are hidden and the client pings the server momentarily to verify that it's still connected. Gameloop prevents the client from receiving more messages from the server and reacting to them. | EMULATION |

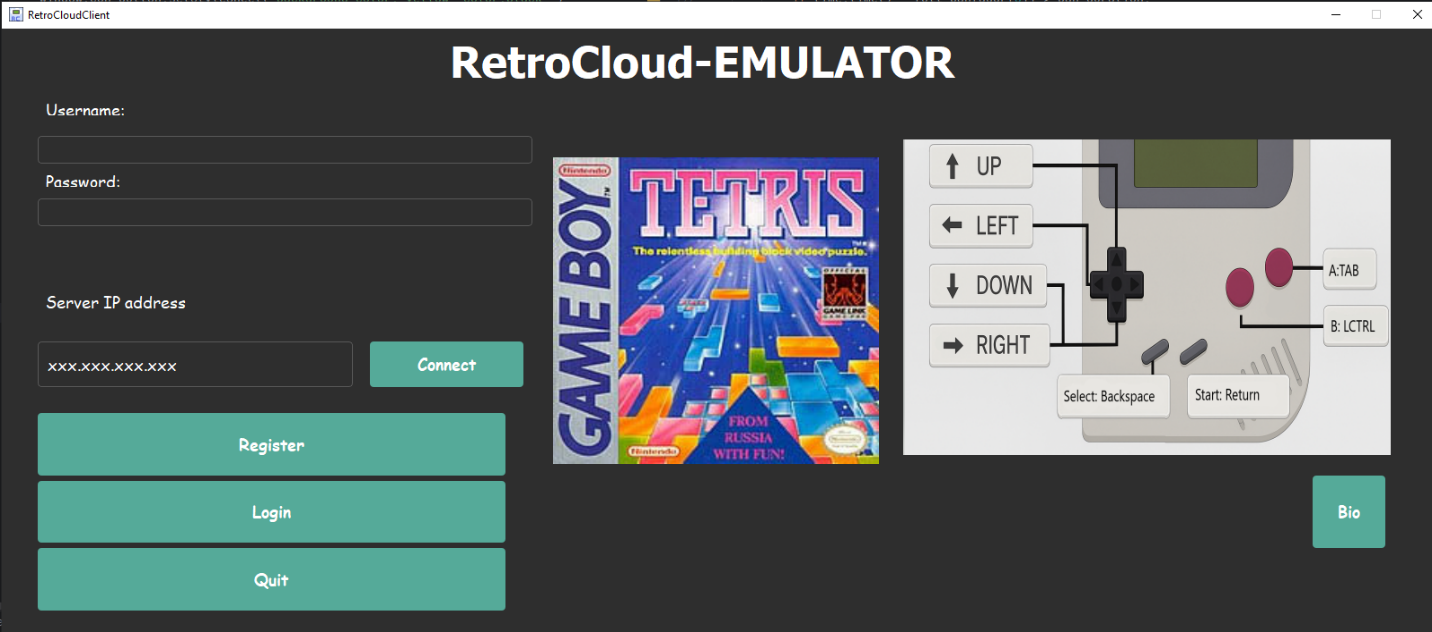
**Server: asyncServer.py**

**\*The server does not have any classes, Therefore all that will be mentioned in this section is functions**

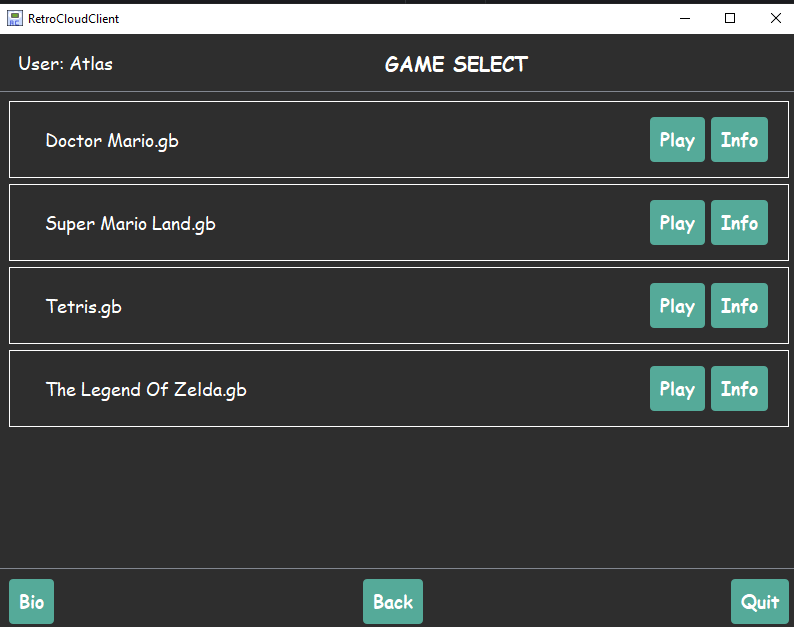
|  |  |  |
| --- | --- | --- |
| Function | Purpose | Module |
| send\_message | Puts together a message packet and sends it to the client. | COMMS |
| recvall | Used when receiving a message from a client. Constructs message packet to be decrypted and read. | COMMS |
| ensure\_db | Creates a database if it detects that the db file is missing | AUTH |
| register\_user | Checks the database if the user it was given exists, and if not, creates a new user with the username and password it was given | AUTH |
| login\_user | Checks the database if the user data it was given exists, and if so then if it’s valid. Should the information be valid, then it logs in the specific client as that user. | AUTH |
| logout\_user | Sets the logged in and playing fields of the database as false for a specific user. | AUTH |
| update\_manifest | Creates a list of the games the server has | GAME RESOURCES |
| handle\_client | Is responsible for managing all interactions with a particular client. Once a client is assigned a handler, it can then communicate with the server using the established protocol. | ASSIGNER |
| main | Sets up database, starts the server, awaits the joining of clients and assigns them handlers upon them connecting. | ASSIGNER + COMMS + AUTH |

**GUI/Graphics:**

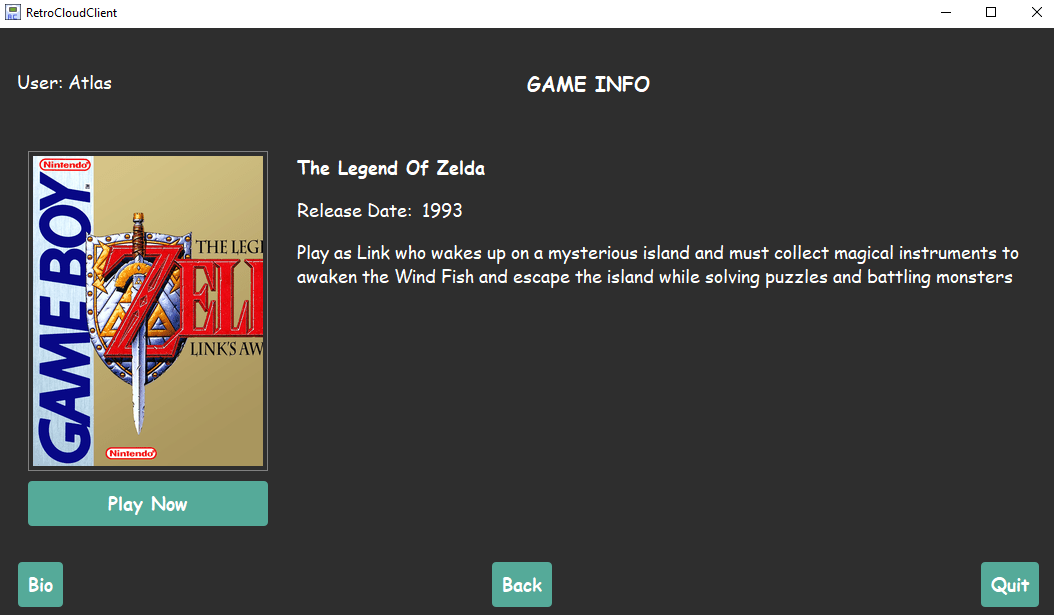
****

****

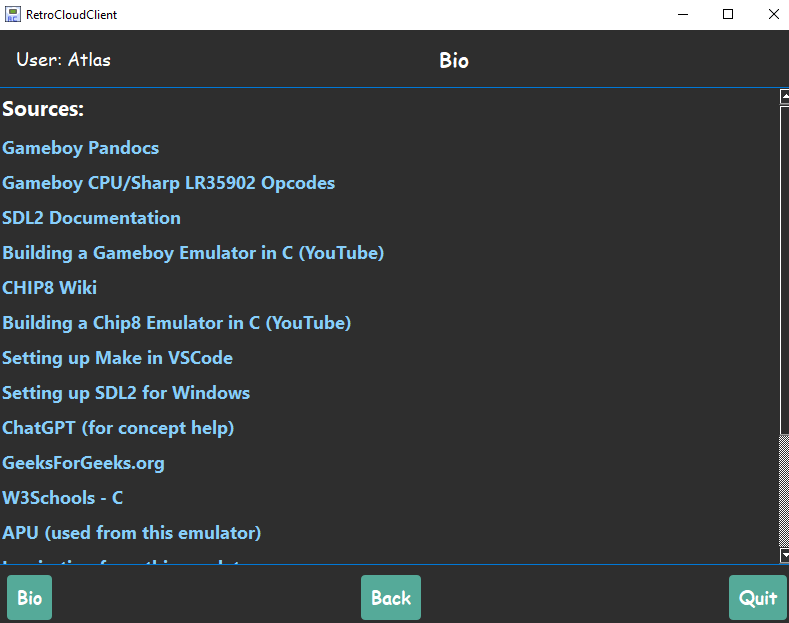
This is the main menu screen. Here, you can connect to the server, and either create an account or login to an existing one. You also get the option of viewing the credits page(bio button). From the main menu you can also quit the app.

****

This screen is the one that will be displayed following a successful Registration or Login. From this screen, one can view all the available games the server has to offer, in the form of game slots/rectangles. In each “slot”, are two buttons that allow the user to either directly launch a game, or view more information about them in the game info screen. From this screen, the user can also choose to go back to the main menu, go to the credits page or quit the app.

****

This screen is presented if the user clicks on the info button from the game selection screen. Inside it the user is met with the Game’s title, it’s release date, a short summary of the game’s content and a cover image. The option to play the game is also provided here. Additionally, the user can choose to go back to the game selection screen, go to the credits page or quit the app.

****

This is the credits screen. Any other screen has access to it, from the bio button. From this screen the user can visit any of the sources I used to make the project. Upon going back, they would be taken to the screen from which they entered credits. The user can also quit from here.

****

This is how the game screen would look like while playing Mario. From this screen the user can interact with the game using the controls shown in the main menu. Closing this screen would return you to the game selection screen.

**Reflection:**

This project has perhaps been one of the more challenging ones that I’ve done. That being said, it has also been one of the more fascinating projects I have taken up. If there is something that I enjoy doing when it comes to programming, it’s finding an interesting topic, and truly throwing myself into it, from all angles. Letting myself become immersed in the material and taking all the steps that I can to become not just familiar with it, but truly understanding it.

When I first started the final project around, I think it was September or October, I had, out of a lack of ideas, chosen to make a license plate tracking app (like the ones you see in police shows). It didn’t take me very long to make a working proof of concept, and to start working on the more complicated bits of the project. Progress was good, but I slowly found myself, not regretful of my decision, but more dissatisfied. For an opportunity to build a big project, where I had nearly full creative freedom (apart from the fixed parameters of the project), I believed I was not making the most of it. I could’ve chosen nearly any subject in which I wanted to learn, explore and build upon. That all changed, however, around December. One day in class I overheard a conversation between one of my classmates and my teacher. The student had been talking about his project which had to do with a field of programming that I had been learning about in my free time, which was low level programming. It’s a subject I’ve been interested in for quite a while and was slowly working to better my knowledge of, through small projects in the C programming language. I hadn’t even considered the possibility that we would be allowed to work on something like that, given that the project was mostly based around networking. Later that day I asked my teacher if theoretically I could change my project to something else. My first instinct was to suggest my end goal project for the field which was a very basic Operating system written in C and assembly (I know I know, crazy and impractical project but it’s interesting to me). While that idea was shut down, since the project was too focused on operating systems and not enough on networking (at least it would be until a more advanced stage), I was given the ok to change projects to something in the field, should it be capable of networking and cybersecurity to a certain degree. After a little bit of research, I came up with two candidates, either a chess engine, or an emulator. I ended up going with the emulator since I really enjoy retro games. Switching projects this late into the year would be a huge undertaking, not to mention quite risky should things not work out. Given that I had made enough progress with the car app to afford some free time, I decided to try an experiment to test the emulation idea. I gave myself Hannukah break, to make a proof of concept for the project (The CHIP8 Emulator, essentially the baby steps in the world of writing emulators), should I succeed, then I would switch projects, should I fail, I would stick to my first project. Seeing as you are now reading a project book on emulation and networking, I think it’s quite clear how that went.

This is where the main difficulties began. With only 5 months to learn, prototype and build a project from start to finish instead of the initial 8-9 months that we had, time was my main adversary. Second to it was my lack of experience with some of the more intricate details of the project that I had not foreseen when I first chose it, out of my unfamiliarity with the subject. Finding the right build tools, compiling for windows while dealing with the routing of external libraries, writing far larger projects in C, and the immense load of research I needed to do in order to understand how the Gameboy functioned, all while writing a functioning backend for it, made this project a challenge like I had never faced before. Failure was constant, and I had to rethink my approach to problems time and time again, in order to fix things, and progress.

I think my favorite example of this was when I was working on the graphics pipeline for the first Gameboy emulator. Somewhere in the code, I inverted two values that had to deal with the way scrolling worked when running a game. If a character reached the end of the screen, it’s supposed to move the background in the direction the character was going, so that they would “move” to the next area. Because I inverted the values, it would scroll to the opposite direction. This went by unnoticed by me, and made it’s way into several graphics related functions. I spent nearly 3 days trying to find where I went wrong. My code was so scrambled and all over the place, that I lost track of where certain things went, and I had rushed the research side of that feature because I was busy with tests. Eventually I realized that this brute force and rushed approach was not only what got me in that situation in the first place, but that it was inefficient and unnecessary. I went back and reread the section on scrolling and eventually came up with the idea to invert the input, so that it would cancel out the inversion done in the functions that made scrolling worked.

I learned from every mistake that I made throughout this project. Through this, I gained a better understanding of what I was working on, and an appreciation of the inner workings of the software and hardware which I was emulating. These mistakes and my efforts to fix them eventually motivated me to rewrite the emulator, making it much more compact, simpler to document, and easier to understand.

## Looking back what would I do differently:

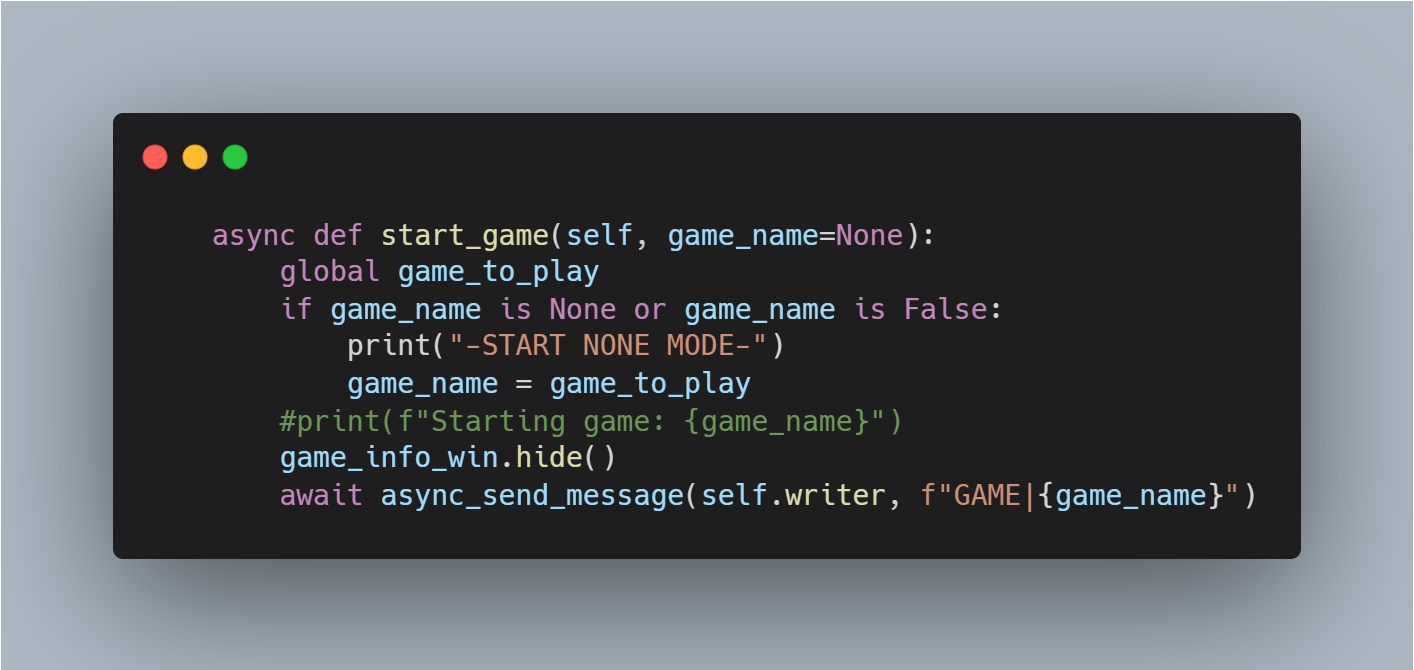
This project, albeit difficult, and filled with constant challenges, was an incredible learning opportunity. Had I been able to start the project anew, with the knowledge I have gained until now, I would have tried harder, and thought broader, to come up with a project idea that I truly felt passionate about, so that I would not end up switching mid-year and have to deal with the time constraints. With this added time, I would also have been able to research more to avoid hiccups down the line that stemmed from a lack of understanding of the material.

Nevertheless, I do not regret any minute I spent, frustrated at the bugs my code had, nor the many nights in which I stayed up late in order to fix them. This has certainly been one of my favorite projects that I’ve done.

**Special segments of code:**

**Backend Segments:**

This segment is the client side code that initiates a gamedata request

****

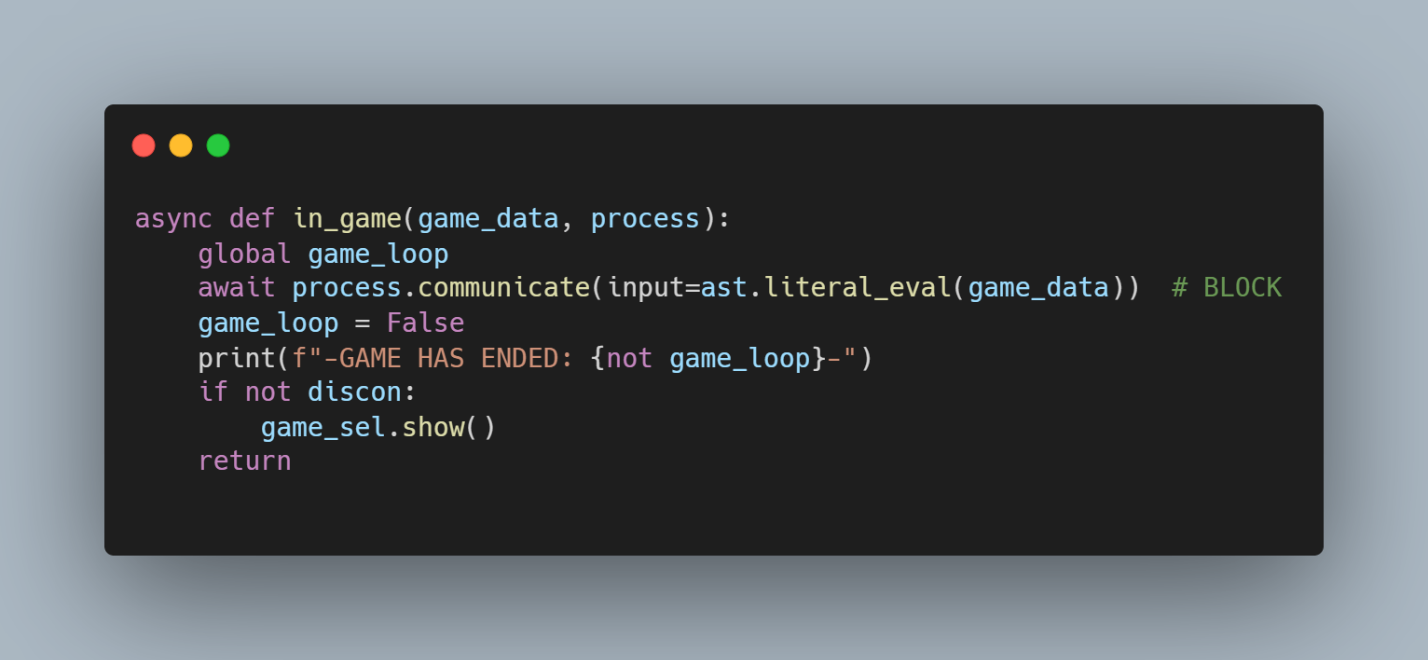
The following segment is the server side code which is run when the server receives a gamedata request:



The following segment is the client side code is run when the server properly responds to a game data request. The message is split apart and the game’s binary data segment is given to the ingame function, which transmits it to the emulator over stdin.

****

This is the ingame function, which feeds the game data to the emulator process and monitors it until it’s closed.

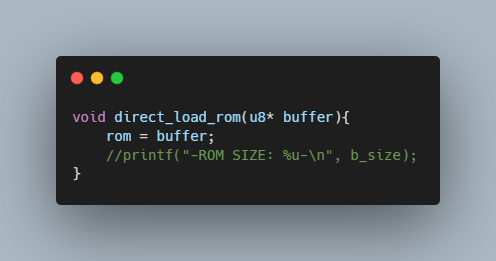


**Emulator Segments:**

This segment of code is the one that creates the memory block: buffer, that reads from stdin, and then assigns it to a pointer through the direct\_load\_rom() function. This is how the ROM is loaded on the emulator’s end.



This function assigns the buffer a pointer, to then be accessed later.



**Bibliography/Sources:**

1. [Gameboy Pandocs](https://gbdev.io/pandocs/About.html).
2. [Gameboy CPU/Sharp LR35902 Opcodes](https://meganesu.github.io/generate-gb-opcodes/)
3. [SDL2 Documentation](https://wiki.libsdl.org/SDL2/APIByCategory).
4. [Building a Gameboy Emulator in C, youtube guide](https://www.youtube.com/watch?v=e87qKixKFME&list=PLVxiWMqQvhg_yk4qy2cSC3457wZJga_e5)
5. [CHIP8 Wiki](https://en.wikipedia.org/wiki/CHIP-8)
6. [Building a Chip8 Emulator in C, youtube guide](https://www.youtube.com/watch?v=YvZ3LGaNiS0&list=PLT7NbkyNWaqbyBMzdySdqjnfUFxt8rnU_)
7. [Setting up Make in VScode](https://www.youtube.com/watch?v=06D1tBKeTB4)
8. [Setting up SDL2 for Windows](https://www.youtube.com/watch?v=H08t6gD1Y1E)
9. [ChatGPT-to ask about concepts and how to do certain operations](https://chatgpt.com/)
10. [GeeksForGeeks.org](https://www.geeksforgeeks.org/)
11. [W3Schools-C](https://www.w3schools.com/c/c_getstarted.php)
12. [APU(I used the APU implementation from this emulator)](https://github.com/sysprog21/gameboy-emu)
13. [Took inspiration from this emulator](https://github.com/rockytriton/LLD_gbemu/blob/main/README.md)
14. [CPP Gameboy Emulator I used for a function blueprint](https://github.com/Jormit/Gameboy-Emulator)
15. [PYQT5 Documentation](https://doc.qt.io/qtforpython-5/index.html)
16. [Setting up OpenSSL on Windows](https://tecadmin.net/install-openssl-on-windows/)
17. [Asyncio Documentation](https://docs.python.org/3/library/asyncio.html)