CinsApartment Final Homework Report

Final Homework Report

Akıncan KILIÇ
Department of Computer Engineering
Computer Network Programming
Manisa, Turkey
akincan.kilic.se@gmail.com

***Abstract*—In this paper, I will be providing a detailed explanation of the design and implementation of the client-server system. I will be discussing the different components of the system and their functionalities, as well as the challenges I faced and the solutions I came up with. Also, I will be walking through some of the code I wrote, which includes how I implemented the GUI using Flet [1], how I handled the network communication using threading, and how I handled the different client operations. Lastly, I will provide a summary of my overall experience with the project and any suggestions for future work and will also serve as a reference for those interested in implementing similar systems using similar tools and techniques.**

***Index Terms*—Client-Server Architecture, Python, Flet, GUI, Threading, Multiprocess Communication, Network Communication**

I. Introduction

This project is related to the Computer Network Programming lecture, which I am taking in my 7th semester. In this project, I set out to design and implement a simple Client-Server architecture using Python.

The instructions were to use Asynchronous code and the C# language to write both the server and the client codes to utilize the Windows Forms feature; however, I only had a macOS machine with me. After days of attempting to install a Virtual Machine to run the Windows operating system, I encountered many issues, primarily due to the new M1 chip on my macOS machine; I could not get a version of Windows to run on my machine. Therefore to make my GUI, I decided to use the new Python module called Flet [1] instead of ”Windows Forms”, and I decided to switch to Python instead of C#.

The second problem I encountered was when I tried to implement the Asynchronous logic in Python. In Python, async programming is much more complicated than C#, where you have to create event loops, assign tasks, gather and await their results, etc. And this heavily complicated the code and made it nearly unreadable, making it hard to maintain, understand and refactor. And since I already had to make many threads in my very own Client and Server architecture that I was designing, I decided to let go of trying to over-complicate my code with async, and I decided to use threads instead.

II. Fetching Data From the Internet

A. Fetching Weather Data

To fetch the Weather data, I used the given website in the assignment, which was weather.com [2]. I used the Python requests library to fetch weather data from a specific URL that was targeting the location of Manisa. I achieved this by using the `requests.get()` method. I then used the `BeautifulSoup` library to parse the HTML content of the page, which I obtained from the `requests.get().content` method.

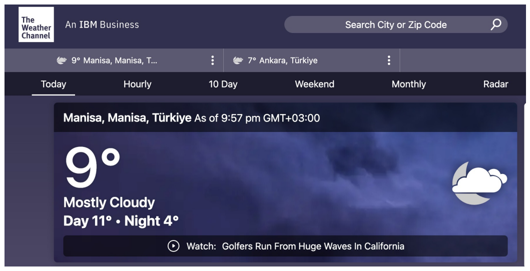


Fig. 1: Sample image from weather.com

To get the required weather information, such as the weather description, temperature, and day & night temperature, I used the select_one method provided by `BeautifulSoup` and passed the required CSS Selector to select the specific elements from the HTML.

The temperature was in Fahrenheit, which I then converted to Celsius using a helper method `fahrenheit_to_celcius` Once I had collected all the data, I returned them as a dictionary.

The class `WeatherDataFetcher` also provides constant fields of empty weather data and CSS Selectors to provide a template to the Server cache before startup.
Here is the weather data fetched and how it looks for the Client:

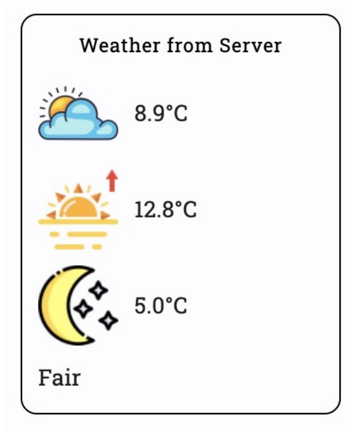


Fig. 2: Weather Data on GUI

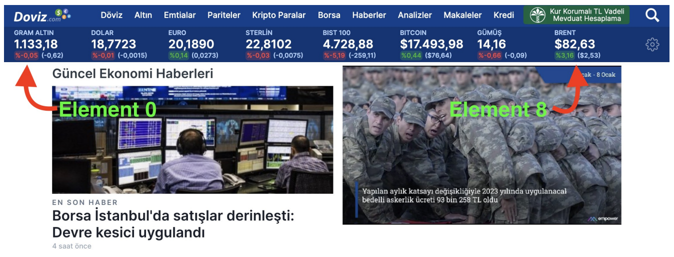
B. Fetching Currency Data

To fetch the currency rates data, I used the Python requests library to make a `GET` request to doviz.com [3]. I decided to go with this website because it provided all the currency information I needed in real-time, and it was relatively simple to scrape. Then similar to the weather fetching, I used the `BeautifulSoup` library to parse the HTML content of the page obtained from the request.
To get the required currency information, such as the USD, EUR, GOLD, GBP, and BTC, I used the select_one method provided by `BeautifulSoup` and passed the CSS selectors that I had defined in the constructor. These selectors were generated using the helper method `get_nth_selector(n: int)`.
This method accepts an integer as a parameter, and it returns a string of the specific CSS selector for that element; on the website, the exchange rates are shown at the top of the page in a specific order, and this method returns the CSS selector of that element based on the order in which it appears.

Here is a table that demonstrates the order of the child elements on the doviz.com website:

Code	Type
1	Gold
2	USD
3	EUR
4	Sterling
5	BIST
6	Bitcoin
7	Silver
8	Brent

Fig. 3: Table of Currencies from doviz.com



GRAM ALTIN	DOLAR	EURO	STERLİN	BIST 100	BITCOIN	GÜMÜŞ	BRENT
1.133,18	18,7723	20,1890	22,8102	4.728,88	$17.493,98	14,16	$82,63
%-0,05 (-0,62)	%-0,01 (-0,0015)	%0,14 (0,0273)	%-0,03 (-0,0075)	%-5,19 (-259,11)	%0,44 ($76,64)	%-0,66 (-0,09)	%3,16 ($2,53)

Güncel Ekonomi Haberleri

EN SON HABER
Borsa İstanbul'da satışlar derinleşti: Devre kesici uygulandı
4 saat önce

ak - 8 Ocak
Yapılan aylık katsayı değişikliğiyle 2023 yılında uygulanacal bedelli askerlik ücreti 93 bin 258 TL oldu

Fig. 4: Sample image from doviz.com

I have also implemented `fetch_exchange_rates` in the class `CurrencyDataFetcher`, which uses these selectors to obtain the data and prepares it as a dictionary for return. The class `CurrencyDataFetcher` also provides a constant field of empty currency data. Additionally, I have used the `convert_string_number_to_float` method to convert the data from string to float and rounded it to 4 decimal places. Also, although the website offered the currencies in Turkish Lira, Bitcoin was offered in $. Which I also converted by implementing some checks for it.

Here is the end result of what the Client GUI sees:

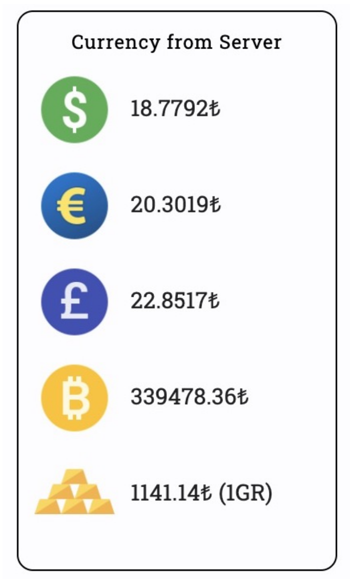


Fig. 5: Currency Data on GUI

III. The MVC Approach for the GUI

The Model-View-Controller (MVC) architectural pattern is a commonly used approach to organize the code for a software project, especially for the ones that have a graphical user interface (GUI). It is designed to separate the concerns of the application's data and logic (the model), the way the data is presented to the user (the view), and the way the user interacts with the data (the controller).

In this project, since I had to write a GUI for both the Server and the Client, and given that their code were already pretty complicated, I decided to separate the GUI code from the logic. Therefore I used the MVC architectural pattern, therefore I split the GUI, the controller code, and the server/client code into different files by following the MVC approach.

By following this approach, the GUI code (ClientGUI.py and ServerGUI.py) is only responsible for displaying the data to the user and capturing the user's input. It doesn't contain any business logic or any knowledge of the data.

The controller code (ClientController.py and ServerController.py) acts as an intermediary between the GUI and the actual server/client logic, it handles the user's input, update the model and updates the GUI accordingly.

The server/client code (Server.py and Client.py) contain the actual logic, such as handling the network communication and the data model.

This separation of concerns makes the code more modular, easier to understand, and maintain, as the changes in one component don't affect the others, also it makes it easy to test, change and debug the code independently.

In this project you will see the following file structure:

- Client.py
- ClientController.py
- ClientGUI.py
- Server.py
- ServerController.py
- ServerGUI.py

IV. Helper Files

I created some helper python files to make the code cleaner and more manageable.

A. The Utility File

Since there were many functionalities common across both the ServerGUI and the ClientGUI, I decided to create a Utility function where I could store the common methods that were required for both of them to keep the code more maintainable and clean.
Some of these include:

- The exit button
- Theme switch button
- App title
- Initialize the main flet page
- Getting current time as a string
- Getting random client names and apartment numbers
- Creating snackbars in the GUI for error messages
- Getting weather/currency information display containers

This file also includes button stylings to make the GUI look prettier.

B. Custom Exceptions File

I also had a need to develop my very own custom exceptions since there were many errors that needed to be caught in a try, catch block. And I needed specific details about why that exception had occurred. Therefore I created the `custom_exceptions.py` file which stores many kinds of specific exceptions to be caught that are defined by me, such as:

- ServerAlreadyRunningError
- InvalidPortError
- ApartmentNoShouldBeIntegerError
- NoServersFoundOnThisHostAndPortError
- ClientAlreadyConnectedError

There are more of these kinds of custom exceptions that really allowed me to control and handle the complicated nuances of trying to manage a massive multi-threaded server like this.

V. The AkinProtocol

The AkinProtocol is a custom protocol that I have created to handle the communication between the client and the server. The main purpose of it is to define a standardized way for the client and server to communicate with each other, to ensure that the client and the server are speaking the same language and understand the same commands.

By using this protocol, I wanted to have a clear separation of concerns between the client, the server, and the protocol, which makes it much easier to understand the communication between the two parties, and also easier to debug, maintain and add new features.

I provided a set of predefined commands, as well as predefined responses such as OK and ERROR, which the client and server use to exchange information.

A. Predefined Commands

The client and server use the following predefined commands to exchange information:

- `WEATHER`
- `CURRENCY`
- `SUBSCRIBE`
- `UNSUBSCRIBE`
- `CHAT_MESSAGE`
- `REGISTER_USER`

B. Predefined Responses

The client and server use the following predefined responses:

- `OK`
- `ERROR`

C. Predefined Delimiter

These predefined commands have a special delimiter attached to them that allows me to separate what exactly the client wants from the server, and what is the actual user input, here is that delimiter:

DELIMITER = @@|<<!?!>>|@@

This delimiter is very unique that it would take 440+ years for a brute forcing approach to even find it, let alone an user accidentally or purposefully enter this message themselves. So it could also be said that this makes the messaging more secure since the Server doesn't accept any message that does not exist in the AkinProtocol, where each of the commands have this special delimiter.

Overall, the AkinProtocol is an essential part of the project, it ensures that the client and server can communicate with each other in a clear, consistent, and predictable manner. It makes the client and server code simpler, more maintainable, and more readable and it is the foundation of the client-server communication which allows the clients to communicate and interact with the server.

VI. The Server and how it Works

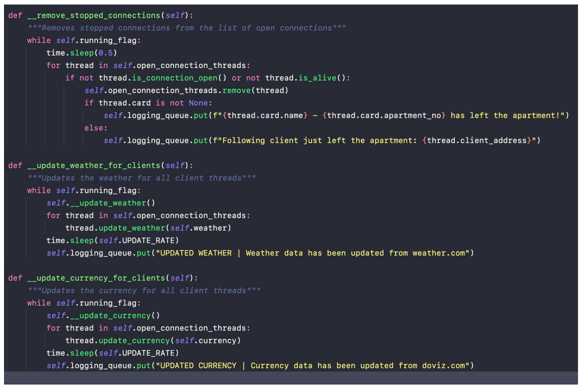
The server class itself is a thread, first it binds and listens for new connections with the host and port that it was given when it was constructed, then it starts accepting new clients and creates an entirely new thread to handle them.

Then I wrote some helper functions which are essential to the server code to help manage it's many clients that are spread across many threads. The server has a `multiprocessing.Queue()` object which allows for communication between the processes. I use this type of object to communicate between my threads.
The server has a `message_queue` and a `logging_queue`, these are both `multiprocessing.Queue()` objects. I store all the messages the clients send on the `message_queue`, and I use the `logging_queue` to display any server management related updates back to the GUI. Since the GUI is a separate process, it has to communicate in this way.
The server keeps all the client threads that it has opened this far in a variable called `open_connection_threads`.

A. Helper Threads

The server code uses the following helper threads to handle various tasks in the background. Here are all the helper threads that are running on the Server code and their explanations:

- **group_chat_updater_thread -** This thread continuously monitors the `messsage_queue` of the server in the background and if there is any message present that was sent from the clients that is a command to send a message to the group chat, this thread sends that message to all the active client threads of the server.
- **open_connection_checker_thread -** This thread continuously monitors the active client threads of the server to check if any has disconnected, or lost connection somehow, and then logs these messages to the GUI to keep the server status accurate.
- **weather_updater_thread -** This thread calls the Weather class that I explained above and fetches new weather data from it and stores it in the local cache of the main server thread. The client threads then pull this local cache from the main server if the client ever makes a request for weather data.
- **currency_updater_thread -** This thread works very similar to the weather updater thread, but instead does the same things for the currency. Both the weather and the currency thread is controlled by a variable called `UPDATE_RATE` which controls how long these threads sleep between each scrape of new data, it is important to keep this rate at a reasonable level such as scrape once every 10 seconds minimum to avoid getting banned from the websites.

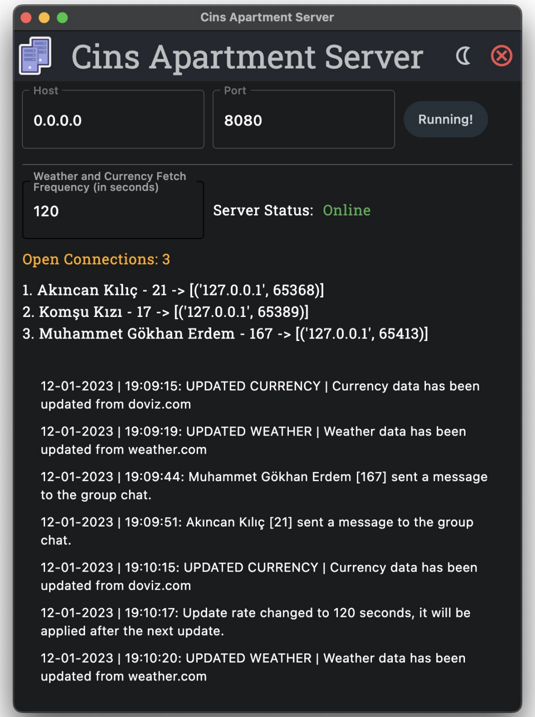


```python
def __remove_stopped_connections(self):
    """Removes stopped connections from the list of open connections"""
    while self.running_flag:
        time.sleep(0.5)
        for thread in self.open_connection_threads:
            if not thread.is_connection_open() or not thread.is_alive():
                self.open_connection_threads.remove(thread)
                if thread.card is not None:
                    self.logging_queue.put(f"{thread.card.name} - {thread.card.apartment_no} has left the apartment!")
                else:
                    self.logging_queue.put(f"Following client just left the apartment: {thread.client_address}")

def __update_weather_for_clients(self):
    """Updates the weather for all client threads"""
    while self.running_flag:
        self.__update_weather()
        for thread in self.open_connection_threads:
            thread.update_weather(self.weather)
        time.sleep(self.UPDATE_RATE)
        self.logging_queue.put("UPDATED WEATHER | Weather data has been updated from weather.com")

def __update_currency_for_clients(self):
    """Updates the currency for all client threads"""
    while self.running_flag:
        self.__update_currency()
        for thread in self.open_connection_threads:
            thread.update_currency(self.currency)
        time.sleep(self.UPDATE_RATE)
        self.logging_queue.put("UPDATED CURRENCY | Currency data has been updated from doviz.com")
```

Fig. 6: Server Helper Thread Codes

A single client thread is the one that actually does all the message handling that is given to its socket, it uses the `AkinProtocol` file to check if the message sent is valid, and then uses `AkinProtocol` again to understand which command it has been requested, and splits what the user wants by the delimiter. It then also responds with a `AkinProtocol` command back to the client. This is why `AkinProtocol` was essential for this project. The client thread handles the following messages:

- **Register User -** Registers the user when the client scans their card to the server.
- **Subscribe Request -** Allows the user to view and send messages to the group chat of the server.
- **Unsubscribe Request -** Allows the user to decide to leave the group chat, but keep receiving weather and currency data, and just not participate in the group chat.
- **Chat Message -** This is a chat message that the client tried to send to the server, the client thread catches this and puts it to the `message_queue` of the main server thread for it to be handled and passed down across all other client threads.
- **Get Weather -** Returns the weather data that is locally cached on the main server back to the client.
- **Get Currency -** Returns the currency data that is locally cached on the main server back to the client.



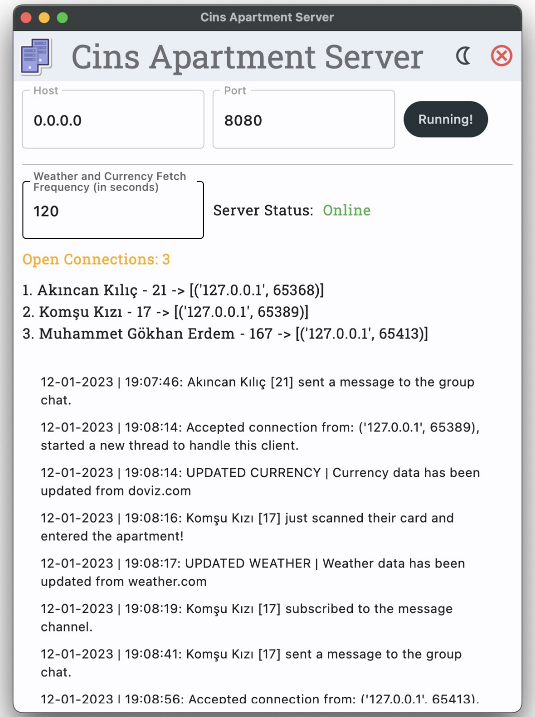
Host: 0.0.0.0
Port: 8080
Weather and Currency Fetch Frequency (in seconds): 120
Server Status: Online

Open Connections: 3

1. Akıncan Kılıç - 21 -> [('127.0.0.1', 65368)]
2. Komşu Kızı - 17 -> [('127.0.0.1', 65389)]
3. Muhammet Gökhan Erdem - 167 -> [('127.0.0.1', 65413)]

```
12-01-2023 | 19:09:15: UPDATED CURRENCY | Currency data has been updated from doviz.com
12-01-2023 | 19:09:19: UPDATED WEATHER | Weather data has been updated from weather.com
12-01-2023 | 19:09:44: Muhammet Gökhan Erdem [167] sent a message to the group chat.
12-01-2023 | 19:09:51: Akıncan Kılıç [21] sent a message to the group chat.
12-01-2023 | 19:10:15: UPDATED CURRENCY | Currency data has been updated from doviz.com
12-01-2023 | 19:10:17: Update rate changed to 120 seconds, it will be applied after the next update.
12-01-2023 | 19:10:20: UPDATED WEATHER | Weather data has been updated from weather.com
```

Fig. 7: Dark Mode Server GUI



Host: 0.0.0.0
Port: 8080
Weather and Currency Fetch Frequency (in seconds): 120
Server Status: Online

Open Connections: 3

1. Akıncan Kılıç - 21 -> [('127.0.0.1', 65368)]
2. Komşu Kızı - 17 -> [('127.0.0.1', 65389)]
3. Muhammet Gökhan Erdem - 167 -> [('127.0.0.1', 65413)]

```
12-01-2023 | 19:07:46: Akıncan Kılıç [21] sent a message to the group chat.
12-01-2023 | 19:08:14: Accepted connection from: ('127.0.0.1', 65389), started a new thread to handle this client.
12-01-2023 | 19:08:14: UPDATED CURRENCY | Currency data has been updated from doviz.com
12-01-2023 | 19:08:16: Komşu Kızı [17] just scanned their card and entered the apartment!
12-01-2023 | 19:08:17: UPDATED WEATHER | Weather data has been updated from weather.com
12-01-2023 | 19:08:19: Komşu Kızı [17] subscribed to the message channel.
12-01-2023 | 19:08:41: Komşu Kızı [17] sent a message to the group chat.
12-01-2023 | 19:08:56: Accepted connection from: ('127.0.0.1', 65413),
```

Fig. 8: Light Mode Server GUI

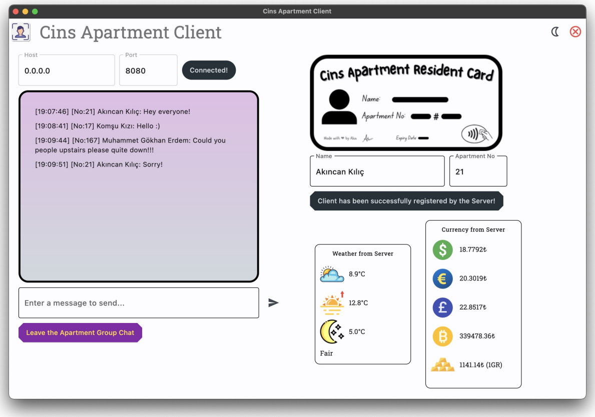
VII. The Client and how it Works

The `Client.py` is a lot simpler compared to the `Server.py`, it also takes in a host and a port to open a connection to the server, and then in a forever loop, it asks for currency and weather data from the server. This is in order to always keep it's Client local cache synced in with the local cache of the Server, this allows it to have all the recent information available from the server.

I then wrote all the possible `AkinProtocol` messages it could ask for from the server in different methods, these different methods are then called by the GUI, that passes its message down to the Controller, which then asks for the Client thread to send a message to the server with that specific protocol that you asked for from the GUI.

This Client code also has a helper thread that helps it to listen to all the messages the server is sending, and to pass them back to the original Client thread. I separated this logic into another thread since it allowed for faster response times and didn't keep the actual Client socket busy, so that the GUI could ask it to send requests while it didn't have to wait to receive messages back from the server, the helper thread is the one that is in a blocking state, waiting for messages from the server, meanwhile the main client thread just waits to be commanded to ask for requests to the server.

Since everything is maintained by the `AkinProtocol`, the client listener thread does not need to know which command was prompted to the server and what information was asked. Since the messages returned by the server are in the `AkinProtocol` format, the `ClientMessageListener` thread always updates and informs the main client thread the accurate information. This again shows the importance of writing my own protocol and why it was essential for an app like this.



Host: 0.0.0.0
Port: 8080

```
[19:07:46] [No:21] Akıncan Kılıç: Hey everyone!
[19:08:41] [No:17] Komşu Kızı: Hello :)
[19:09:44] [No:167] Muhammet Gökhan Erdem: Could you people upstairs please quite down!!!
[19:09:51] [No:21] Akıncan Kılıç: Sorry!
```

Enter a message to send...

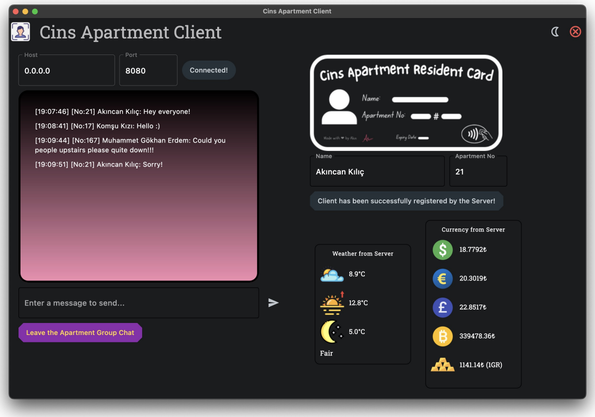
Name: Akıncan Kılıç
Apartment No: 21

Client has been successfully registered by the Server!

Weather from Server: 8.9°C, 12.8°C, 5.0°C, Fair

Currency from Server: 18.7792₺, 20.3019₺, 22.8517₺, 339478.36₺, 1141.14₺ (1GR)

Fig. 9: Light Mode Client GUI



Host: 0.0.0.0
Port: 8080

[19:07:46] [No:21] Akıncan Kılıç: Hey everyone!
[19:08:41] [No:17] Komşu Kızı: Hello :)
[19:09:44] [No:167] Muhammet Gökhan Erdem: Could you people upstairs please quite down!!!
[19:09:51] [No:21] Akıncan Kılıç: Sorry!

Name: Akıncan Kılıç
Apartment No: 21

Client has been successfully registered by the Server!

Weather from Server: 8.9°C, 12.8°C, 5.0°C, Fair

Currency from Server: 18.7792₺, 20.3019₺, 22.8517₺, 339478.36₺, 1141.14₺ (1GR)

Fig. 10: Dark Mode Client GUI

VIII. The Card Reader

I also implemented a simple card reader feature, the card holds the name and the `apartment_number` of the client. The client then can use the Register button to send a request to the server to register themselves to the apartment. (In the future this could be connected to a database that will check whether if the resident is actually staying at that apartment, or whether the card is valid, however it was out of scope of this project). Before sending a message to the group chat, the client has to register by "scanning" their card to the card reader. The server will then respond with `AkinProtocol.OK` or `AkinProtocol.ERROR` to clarify whether it was a successful operation or not.
The clients card data is then used to identify the client on the ServerGUI and on the Group chat.

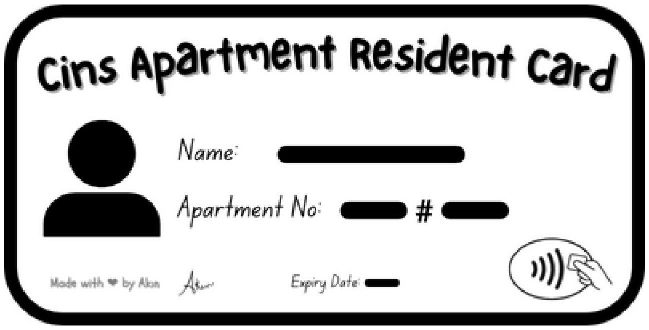


Fig. 11: The Client Card

IX. How to Run and Build This App

1) Use `pip install requirements.txt` on your terminal to install the required packages.
2) Run the `ServerGUI` by typing `python3 ServerGUI.py` in the terminal.
3) Run the `ClientGUI` by typing `python3 ClientGUI.py` in the terminal.

Use your own subnet mask and IP address to open an accurate host on a network that has no firewalls, so you can connect to the Server with a ClientGUI instance open on an entirely different machine under the same network. (Mobile hotspot sharing works really well for this)
The server host can be `0.0.0.0` which broadcasts the server.

X. Final Thoughts

This report was written in overleaf.com [4], which was a great aid in helping with the formatting and styling.

All the icons used in the project, such as the currency icons, the weather data icons were taken from icons8.com [5]

The card reader and some other GUI elements were hand drawn by me using vector drawing software.

This project took me a while to get right, however it has made me learn a lot of things along the way about networking, threading, multiprocess communucation and writing a GUI.

I have many more ideas and features to add in my mind, however those would be just fancy graphical features, along with animations which I had no time to make because of my busy schedule.

The source code of the project is available on GitHub at: https://github.com/akincan-kilic/Cins_Apartment_Management_System

Thanks

I want to thank my professor Muhammet Gökhan ERDEM for his immense dedication and passion to teaching. His lectures were always filled with enthusiasm and he made me fall in love with any subject he taught. His guidance and mentorship have been invaluable in my growth as a student and I am grateful for all the knowledge and skills he has imparted on me. Thank you, Professor ERDEM, for being an inspiration and a role model.

References

[1] flet.dev
[2] weather.com
[3] doviz.com
[4] overleaf.com
[5] icons8.com