Crowd Counting System (CCS)

An NSAPDEV-NSEMBED Joint Project

Members:

Escalona, Jose Miguel A.

Estebal, Eidrene Glena C.

Fortiz, Patrick Ryan P.

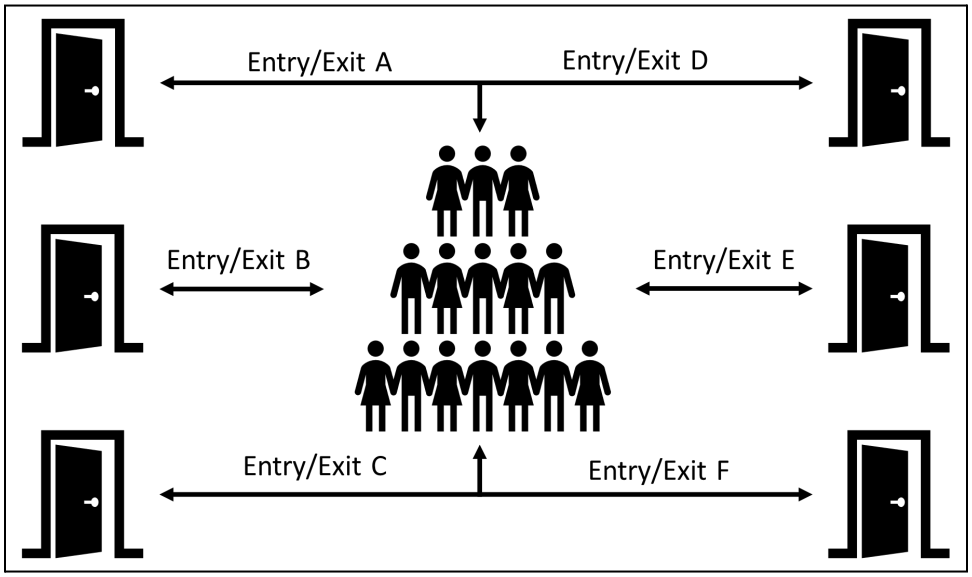
Table of Contents

Project Description....3
Project Capability and Scope....4
    1. Capabilities of the Project....4
    2. Scope of the Project....4
System Overview and Architecture....4
    Overall Logical Architecture....4
    Server Software Architecture....5
    Client Hardware and Software Architecture....10
Server Application....15
Client Application....16
Server Performance....17
    Challenges and Solution....17
        1. Server Application Freezing on Client Disconnection (Server)....17
        2. Updating DB File Blocks Network (Server)....17
    Performance Tests & Results....19
        1. System Specifications....19
        2. DB Test Files....19
        3. Test Environment....20
        4. Load Testing....20
        5. Latency Testing by DB Memory Size....20
        6. Latency Testing by Visitor Limit....21
        7. Latency Testing by Client Thread/Request Limit....23
        8. Noise Test....24
        9. Merge Test....24
Embedded System Results and Analysis....25
Conclusion....27
    Server Application....27
    Embedded System....27
References....28

Project Description

The project is called Crowd Counting System (CCS) which is a crowd logging system that makes use of an RFID reader to keep track of the number of people that are within a particular location by monitoring the number of people entering and exiting a particular premise.

The inspiration for this project is found in the situations during COVID-19 where the need for crowd population management of a particular place is necessary to maintain social distancing or to prevent conditions such as overcrowding in public spaces or events that can lead to devastating and deadly stampedes.



The applications of the project can be for Crowd Management Systems, Visitor Counters, or Physical User Access Control (UAC) Systems which can be applied in concert halls, stadiums, arenas, terminals, train stations (e.g., ticket gates), and other places where crowds converge whilst still requiring a controlled manner of access.

Project Capability and Scope

1. Capabilities of the Project

1.1. The project shall be capable of counting the entry and exit of people in a certain premises.

1.2. The project shall be capable of handling multiple entry and exit points at the same time while ensuring reliable counting and high throughput.

1.3. The project shall be capable of accurately and reliably keeping track of how many persons entered the set premises, and possibly, even who had entered the premises (i.e., ideal for security applications).

2. Scope of the Project

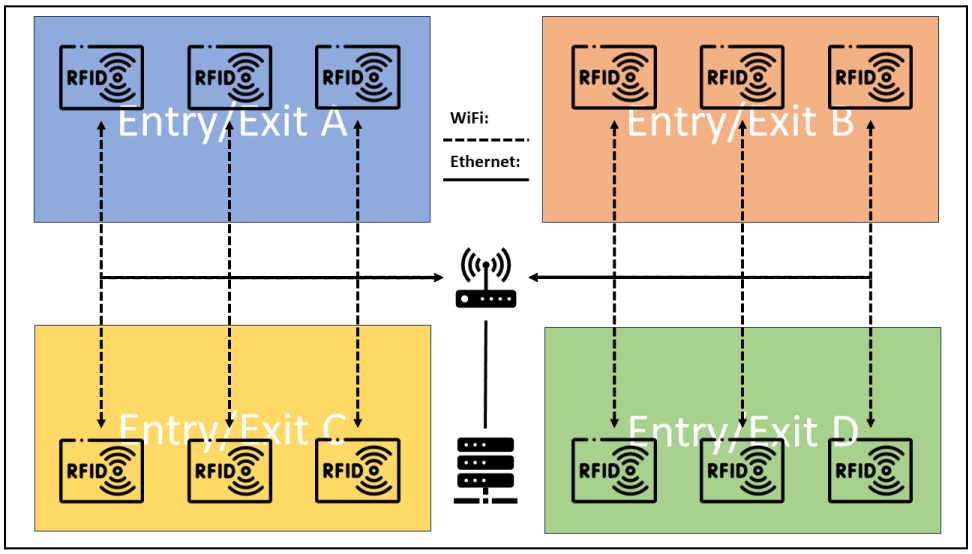
2.1. The scope for the server application side is limited to, at most, handling 200 entry/exit points at the same time, which is similar to the number of entry/exit points at Tokyo's Shinjuku Station.

2.2. The scope for communication protocols used between the embedded devices and the server is limited to IP-based communication.

2.3. The scope for the transaction recording mechanism will be software-defined to reduce the hardware quantity requirement.

System Overview and Architecture

Overall Logical Architecture



Overall System Architecture

The project's overall logical architecture is similar to the project's intention of acting as entry/exit monitors on a given premise. The overall architecture can be divided into the embedded system and the server. The overall architecture also follows a software-defined to limit the need for excessive hardware while still being able to achieve the required objectives.

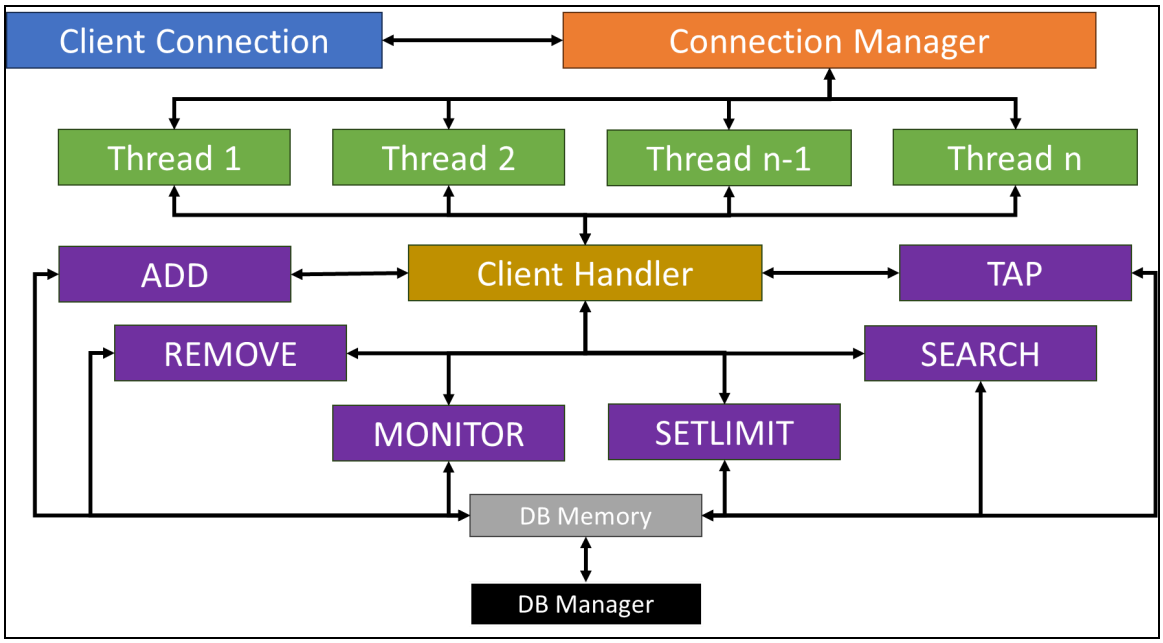
The embedded system will primarily consist of a microcontroller and an RFID reader where each of these is placed on different entry/exit points of a certain premises, to which such embedded systems are connected to the server through WiFi which then uses IP-based communication to transmit/receive data to and from the server.

The server will handle the recording and display of all registered entries and exits on the premises. As mentioned in the scope of this project, the server application is expected to handle multiple simultaneous connections and transaction attempts reliably. However, to keep in line with the objectives of the project, the server also limits the number of people that can enter the premises by returning a "LIMIT" response whenever the attempted entry will result in exceeding entry limits.

Server Software Architecture

The server software architecture will be primarily written in Python and will include the use of libraries that will enable network communications, concurrency, and file handling. Specifically, socket, threading, os, time, and re. The socket module provides access to the properties that will allow communication across a network. The threading library will create the threads needed to accurately count the number of people entering and exiting places. The rest of the libraries—os, time, and re—are used for other functions of the server which are more related to data management.

The server software architecture is a multi-threaded client-server architecture that consists of five layers. Every time a client connects to the server (Client Connection), be it an embedded client or an admin, the system creates a dedicated thread for the client via the Connection Manager. This thread delegates the execution of the necessary instructions associated with the request being made on behalf of the client which is handled by the Client Handler of the server. The Client Handler then executes the appropriate modules to obtain a server response to deliver back to the client. It also holds most of the critical sections of the server application, hence some locks are acquired and released before and after those sections of the server application.



Server Application Architecture

The descriptions of the different command modules of the server application are as listed below. These commands/modules are also reflected in the JSON string that will be passed between the server and the client.

Module	**Description**
`ADD`	Adding a user to the DB Memory, and subsequently the DB text file.
`REMOVE`	Removing a user to the DB Memory, and subsequently the DB text file.
`MONITOR`	Returns a query of statistics regarding the current crowd capacity and the number of known IDs.
`SETLIMIT`	Sets the value of the crowd control capacity limiter.
`SEARCH`	Searches for a given ID and returns if it is in the DB Memory or not.
`TAP`	Returns a tap query whether a given ID has entered, exited, or is invalid.

The server relies on a multi-threaded design such that every client attempting to connect will be considered a thread by the server to which the method attached to the thread will attempt to complete the transaction based on the received command. The code below shows the server creating new 'transaction' threads for every connection the server accepts.

```
def start():
    setup_server()
    load_memory()
    print("FILE:", FILE)
    server.listen()
    print("")
    print(f"DB Updates every {UPDATE_CYCLE} seconds.")
    print(f"DB Memory: {len(dbMemory)} IDs")
    print(f"Server is LISTENING on {IP}:{PORT}...")
    print("")
    dbThread = threading.Thread(target=update_db)
    dbThread.start()
    while True:
        conn, addr = server.accept()
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()
```

The transactions are handled by the *handle_client()* method which contains concurrency and handling of critical resources of the server. The code is as shown below:

```
def handle_client(conn, addr):
    global QUIET
    connected = True
    while connected:
        response = ""
        parsed = None
        parsedcmd = ""
        parsedval = ""
        lock.acquire()
        try:
            rcv = conn.recv(HEADER).decode(FORMAT)
            if rcv == DISCONNECT_MESSAGE:
                connected = False
            elif not ("{" in rcv and "}" in rcv):
                response = jparser.errjson("SERV","Malformed RCV")
            else:
                parsed = jparser.readjson(rcv)
                if parsed == None:
                    response = jparser.errjson("SERV","Unparseable RCV")
```

```
                else:
                    parsedcmd = parsed['cmd']
                    parsedval = parsed['val']
                    if parsedcmd == "TAP":
                        response = jparser.writejson("SERV", "RES", tap(parsedval))
                    elif parsedcmd == "ADD":
                        response = jparser.writejson("SERV", "RES", add_user(parsedval))
                    elif parsedcmd == "REM":
                        response = jparser.writejson("SERV", "RES", rem_user(parsedval))
                    elif parsedcmd == "MON":
                        response = jparser.writejson("SERV", "RES", monitor()[0])
                    elif parsedcmd == "LIM":
                        response = jparser.writejson("SERV", "RES", setlimit(parsedval))
                    elif parsedcmd == "SER":
                        response = jparser.writejson("SERV", "RES", search_user(parsedval))
                    #print(response)
            if not QUIET and "DISCONNECT" not in rcv:
                ui.standardPrint(addr, parsedcmd, parsedval, jparser.readjson(response)["val"], f"({len(premises)}/{LIMIT})")
            conn.send(response.encode(FORMAT))
        except Exception as e:
            ui.exception(rcv, e, addr)
            response = jparser.errjson("SERV","SRV Exception")
            conn.close()
            lock.release()
            return
        lock.release()
    conn.close()
    return
```

The list of all important methods found in *Server.py* is listed below.

Method	**Description**
`start()`	Delegates the runtime of the server by listening to connection attempts to which it launches a respective thread for that accepted connection.
`handle_client()`	Handles the thread call which completes the transaction calls by the client depending on the input received.
`load_memory()`	Loads the given 'DB' text file to the memory as a list.
`setup_server()`	Setups the server connection such as IP address and port no.
`update_db()`	Executes updates to the 'DB' file based on the ID list found in memory. Execution will be made on a separate function (probably via *start()*) and may be event driven (i.e., when there are new IDs in the memory) or time-driven (i.e., per elapsed time).

It is expected that the server will eventually experience errors especially if there are too many consecutive transactions for the server to handle. Hence, for any error that may occur, the server will simply respond "ERROR" back to the client to indicate that the transaction was not able to conclude appropriately.

The JSON string format and its keys' description are as shown below :

`{"dev_id":"deviceID", "cmd":"command", "val":"value"}`	
Key	**Value Description**
dev_id	Contains the device name of the client device that sent the JSON string. Useful for identifying which embedded device sent the data. For the server and admin clients, the *dev_id* is set as "`SERVER`" and "`ADMIN`" respectively.
cmd	Contains the command is sent by the client to the server (i.e., *ADD, REM MON, LIM, SER, TAP*). Only two commands are sent by the embedded device which are SER and TAP. The admin client can send all commands except for TAP. Errors are designated with an ERR value
val	Contains the value associated with the command. The command MON for the embedded system will only contain an empty string value.

The server uses the following Python libraries which are shown in the table below.

Library	**Description**
UI	A custom library that handles UI displays and mechanisms.
JSONParser	A custom library that handles JSON encoding and decoding.
socket	A Python library used to handle websockets programming
re	A Python library used to handle regex functions
time	A Python library used to handle sleep calls.
datetime	A Python library used to handle date & time data
threading	A Python library used to handle threads and thread management.

Client Hardware and Software Architecture

The primary client hardware architecture will be composed of an RFID reader connected to a microcontroller, with the RC522 and ESP32 being the specific hardware of choice. In the system, this hardware pair will be responsible for capturing data from RFID cards which will then be transmitted to the server for processing. A Python equivalent was also made to simulate multiple transactions occurring at the same time. The client will simply send the ID number detected by the RFID to the server to which it will receive either one of four responses:

Response	**Description**
`ENTER`	ID No. received is valid and is allowed to enter the premises. (LED: GREEN)
`EXIT`	ID No. received is valid and will exit the premises. (LED: YELLOW)
`LIMIT`	ID No. received is valid but the premises are already full. (LED: RED)
`INVALID`	ID No. received is invalid. (LED: RED)

A client simulator was created to enable testing and evaluation of the server application's reliability and performance. The client simulator was designed to be able to execute

multiple threads that will simulate multiple people tapping their RFID on the system simultaneously. The libraries used by the Python client are found in the table below.

Library	**Description**
UI	A custom library that handles UI displays and mechanisms.
JSONParser	A custom library that handles JSON encoding and decoding.
socket	A Python library used to handle websockets programming
re	A Python library used to handle regex functions
time	A Python library used to handle sleep calls.
datetime	A Python library used to handle date & time data
threading	A Python library used to handle threads and thread management.

Additionally, an admin console was also created which can manage the server's operational parameters. The admin can access the server to either add, remove, or search for users. The admin console can also monitor and set the limit of how many people can be inside a certain premise. Do note that the admin console also uses similar libraries as found in the normal client.

The admin commands and expected server responses are as follows:

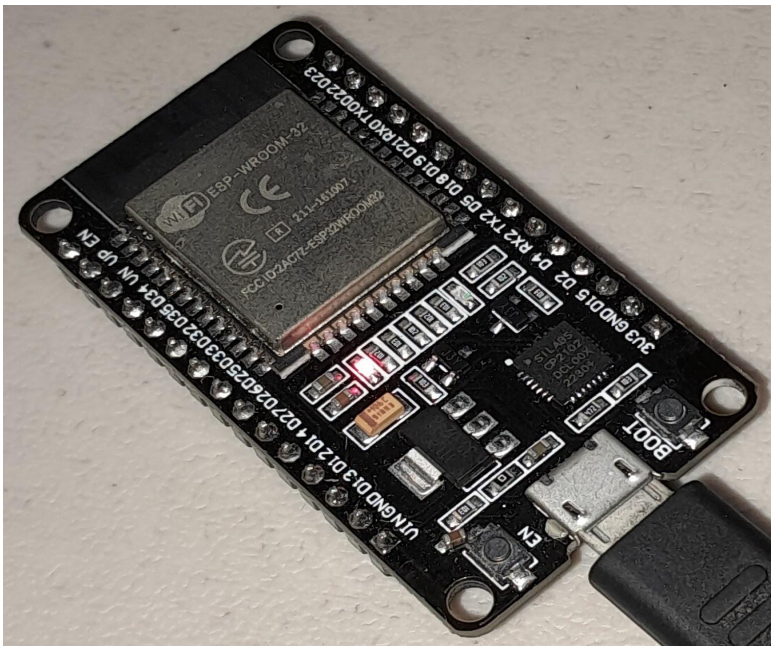
Command	**Expected Server Response**	**Description**
`ADD`	`ADDED/!ADDED`	Adds an ID to the DB Memory
`REM`	`REMOVED/!REMOVED`	Removes an ID to the DB Memory

`MON`	`P:{People in Premises}/`
	`L:{Limit}/`
	`DB:{DB Memory}`
	
	`Formatted as "P:{}/L:{}/DB:{}"`
		Queries number of people in premise, entry limit, and size of DB in memory.

`LIM`	`LIMIT SET = {LIMIT}`	Sets new entry limits
`SER`	`FOUND/NOT FOUND`	Searches for ID

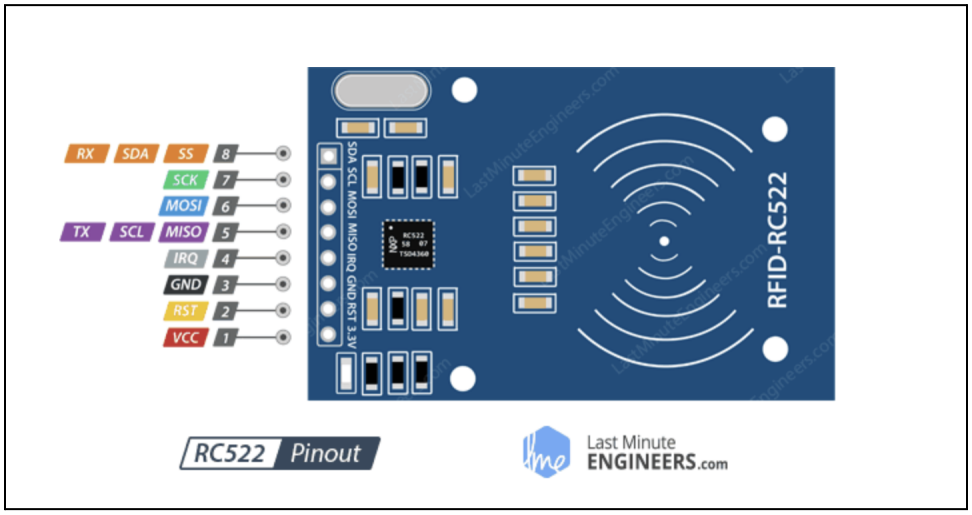
The client has also an embedded system equivalent which is handled by the ESP32 microcontroller. The specifications of the client hardware (embedded system) are as follows:

- ESP32 (Microcontroller)
    - A low-cost and low-power microcontroller with Wi-Fi and Bluetooth connection capabilities.
    - It is powered by an either single or dual-core Tensilica Xtensa 32-bit LX6 microprocessor and is designed for Internet of Things applications.
    - The ESP32 features an 802.11b/g/n HT-40 Wi-Fi transceiver for wireless communications which will be relied upon for communicating with the server.
    - Pin multiplexing is an option for the ESP32. This allows multiple peripherals to share a single GPIO pin. In total, there are 30 available pins on the ESP32-C3-32S variant. A diagram of the available pins, along with their functions may be found below.



ESP32 Microcontroller

- RC522 (RFID)
    - Based on MFRC522, it is a low-cost and easy-to-interface RFID reader.
    - Usually paired with a rewritable RFID card with 1 KB of storage
    - Uses 13.56MHz frequency to communicate with RFID tags and follows the ISO 14443A Standard.
    - Uses either SPI, I2C, or UART serial communication protocols which makes it readily compatible with ESP32.
    - Features eight pins for GPIO communication. Notably, it has an interrupt pin available which can be useful for setting triggers once the RFID reader detects an RFID card/tag.

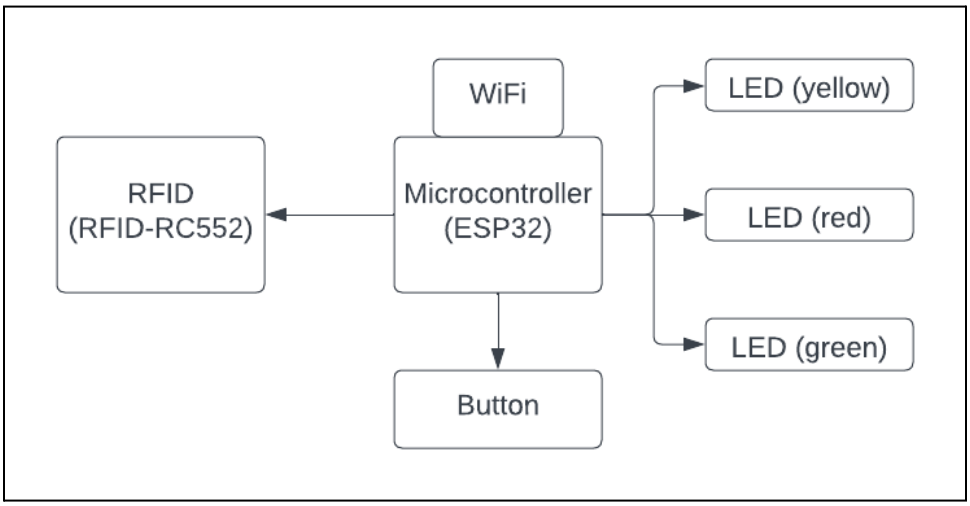


RC522 RFID Reader Pinout

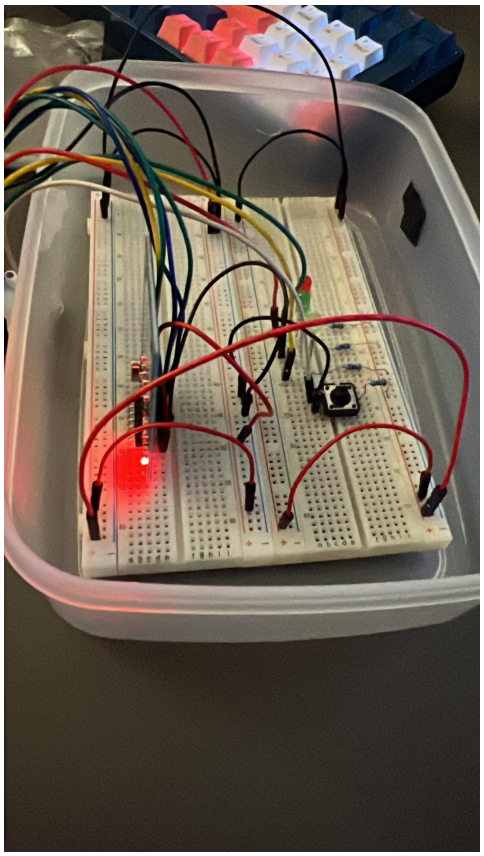
The table below shows the libraries used by the ESP32 client.

Library/API	**Description**
MFRC522.h	It is a library used for reading and writing to an RFID card/tag
SPI.h	It is a library used for allowing communication with serial devices and setting up serial pins
FreeRTOS	It is an API used for thread creation, control, and management
ArduinoJson.h	It is a library used for building and parsing JSON formatted data
WiFi.h	It is a library used for WiFi connection and client management

The image below shows the embedded system's block diagram.



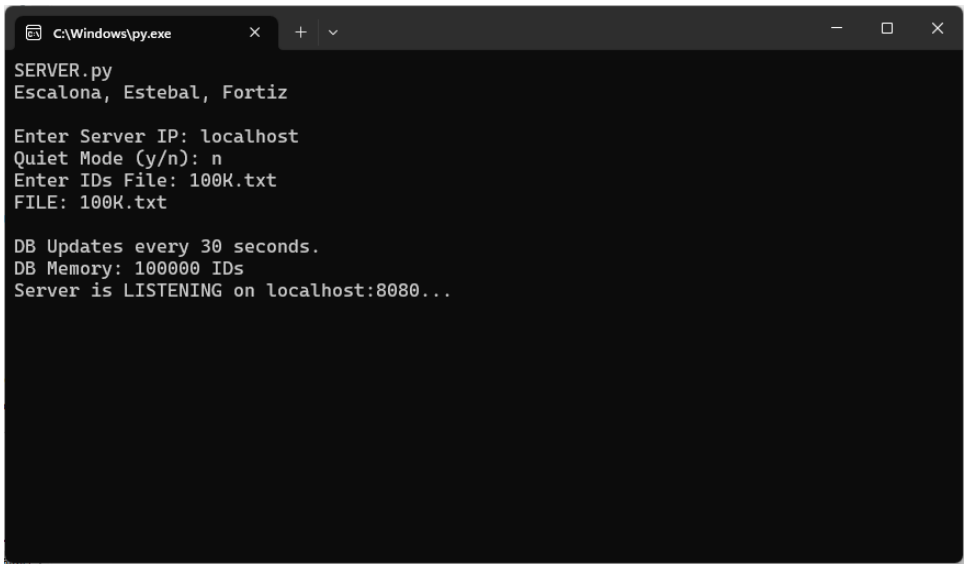
Block Diagram of Embedded System



Actual Embedded System HW

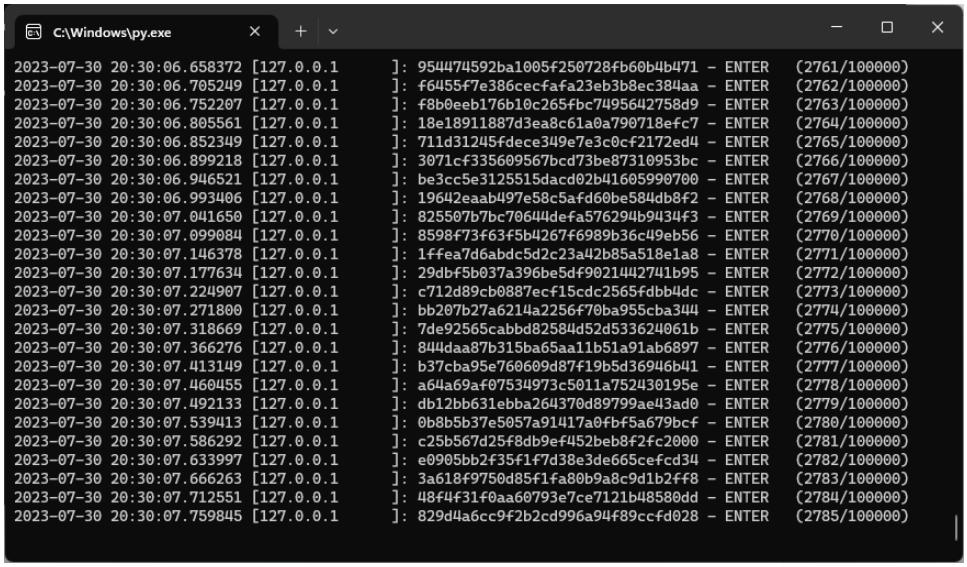
Server Application

The server application is a standalone application that will continuously listen to client requests and will respond accordingly.



Server Running

Everytime a request has been made, it will be logged to the screen as shown below. The server also handles simultaneous requests at the same time such that errors occur minimally. Note that the verbosity of the printout can be set during the server application's setup.

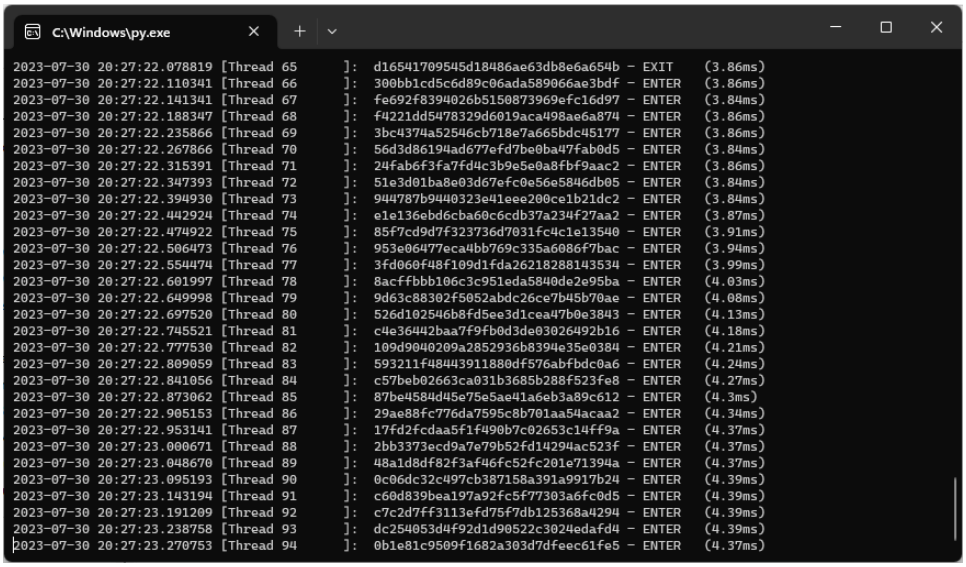


Serverside View of Transactions

The server and client (Python ver.) applications uses techniques such as threading, locks, and semaphores which are available as libraries in Python. The implementation of threads allows for asynchronous processes to occur such that it won't be blocking other processes. Python's socket libraries were also used to manage the connection of both the server and client simulator.

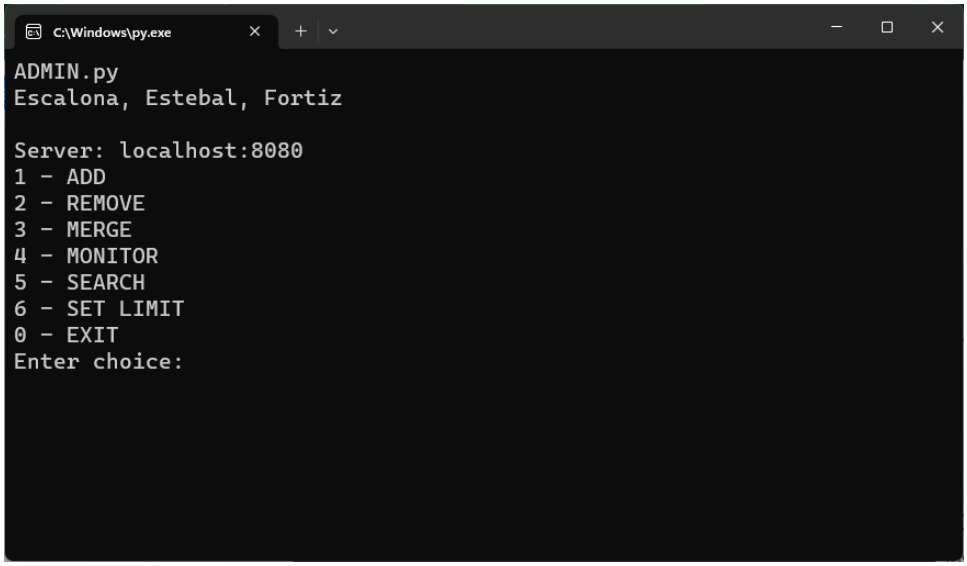
Client Application

A client simulator was built to simulate the embedded devices which send and receive data between client and server.



Client Running (Not Quiet)

The way the client simulator was able to simulate multiple embedded devices is by creating and running a set number of threads that send a random ID (valid or not) to the server which simulates a tap of an ID card in the embedded device. An Admin client was also built which contains advanced functionalities which are more associated with monitoring and modifying the database used by the server application.



Admin Main Menu

Server Performance

Challenges and Solution

1. **Server Application Freezing on Client Disconnection (Server)**

   The original design of the server application had a flaw that causes it to freeze whenever a client suddenly disconnects without issuing the proper disconnect call. It was determined that the previous iterations had no method of handling such an exception which also includes releasing the lock and connection to that client. The solution is shown below.

```
except Exception as e:
            ui.exception(rcv, e, addr)
            response = jparser.errjson("SERV","SRV Exception")
            conn.close()
            lock.release()
            return
```

2. **Updating DB File Blocks Network (Server)**

   The original design of the server application had a flaw when updating the DB file caused issues that blocked the main process that handles client requests once the DB update has been triggered. The solution was determined to have it started as a thread before the loop that handles client requests. The code for the start() and update_db() is as shown below.

```python
def update_db():
    global LASTUPDATE
    global FILE
    global dbMemory
    LASTUPDATE = time.time()
    while True:
        if time.time() - LASTUPDATE >= UPDATE_CYCLE:
            ui.standardPrint("localhost", "UDB", "Updating DB File...", "ONGOING", "")
            try:
                lock.acquire()
                dbMemory = list(dict.fromkeys(dbMemory))
                f = open(FILE, mode="w")
                for id in dbMemory:
                    f.write(id + "\n")
                f.flush()
                f.close()
                lock.release()
                ui.standardPrint("localhost", "UDB", "DB File Updated!", "OK", "")
            except Exception as fileExcept:
                ui.standardPrint("localhost", "UDB", "DB File Updated!", "NOT OK", "")
                ui.exception("updateDB", fileExcept, "localhost")
                continue
            LASTUPDATE = time.time()
```

```python
def start():
    setup_server()
    load_memory()
    print("FILE:", FILE)
    server.listen()
    print("")
    print(f"DB Updates every {UPDATE_CYCLE} seconds.")
    print(f"DB Memory: {len(dbMemory)} IDs")
    print(f"Server is LISTENING on {IP}:{PORT}...")
    print("")
    dbThread = threading.Thread(target=update_db)
    dbThread.start()
    while True:
        conn, addr = server.accept()
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()
```

Performance Tests & Results

1. System Specifications

The following are the specifications of the systems used as the Server and Client.

System	Specifications
Server	CPU: AMD Ryzen 3 1200 (3.5Ghz; 4 Cores 4 Threads) RAM: 12GB Connection: WiFi 4 (2.4Ghz) IP Address: 192.168.2.87
Client	CPU: AMD Ryzen 5 3500U (2.1 Ghz; 4 Cores 8 Threads) RAM: 12GB (9.9GB Usable) Connection: WiFi 5 (5Ghz) IP Address: 192.168.2.252

2. DB Test Files

File Name	Hash (SHA-256)
1K.txt	482389F85A66832256899CB3CA5802858CD9D1622E838CCAE9BFDBDFD5CFF1D3
5K.txt	0462037082ED469B9847340F152B47201B8B71B8904F928BFBF1B5B674C20038
10K.txt	42255B5ACC38F248F030EF40290826F7953BF777BFC919EB08E7DD7CB60E6D8D
25K.txt	55F7947933567C10EA0A3514BCF5FC7EE16BBF0F11130C29FE3EDFDD14B920D5
50K.txt	8E8F7223A85ACF556AB76432E114159F95CEE1F6DA07107F7D23F872F65C3163
100K.txt	5AF7551A45B5D7AAAA8F7855A3F71E30E0C1E579B242C943C153D0EC2772141A
118-121.txt	B752F16F5B594192166801631DCFC724557131464DF3EB8A9F501BC85690FB24

3. Test Environment

The server and client will connect to each other via WiFi. Additionally, it should be noted that the server includes a timely sync of the DB's memory to the DB file which runs on its own separate non-blocking thread. All runs will be executed with the server in Quiet mode to reduce overheads. All tests will use the 100K.txt as the contents of the DB Memory.

4. Load Testing

This test checks the server application's capability to handle simultaneous requests without crashing. The test will be conducted 10 times and executed in a single terminal or two separate terminals (with half the load of the single terminal configuration). The LIMIT value will be set to 100000 to act as the controlled variable.

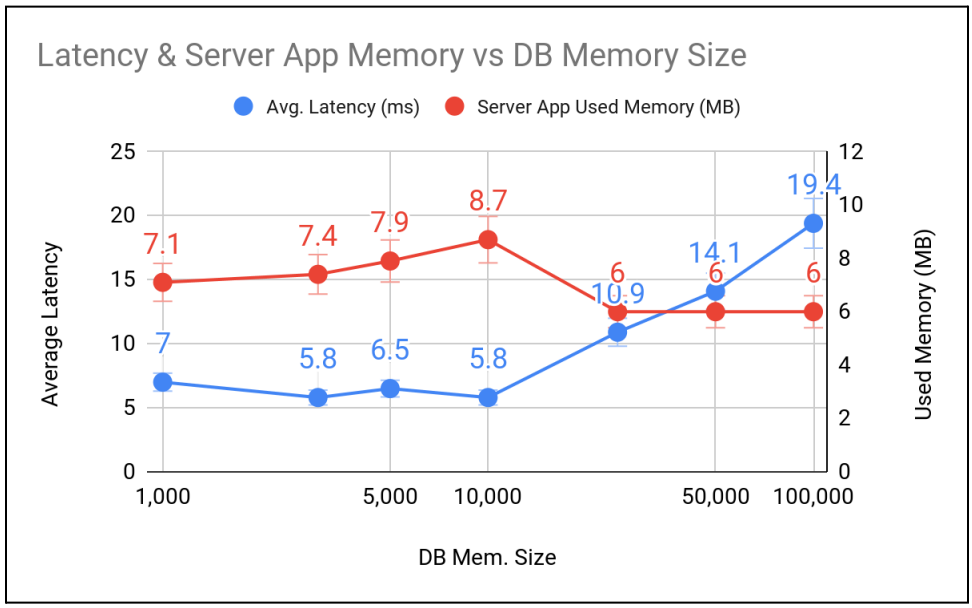
Client Thread Count	Passed
10	OK
100	OK
1,000	OK
10,000	OK
100,000	OK

5. Latency Testing by DB Memory Size

The latency test by DB Memory Size will test the server application's speed in handling the processes related to allowing entry/exit and checking the validity of the ID. The client threads will be set to a value of 1000 threads to act as the controlled variable. It is expected that there will be an increase in latency as the set LIMIT value increases.

DB Mem. Size	Avg. Latency (ms)	Server App Used Memory (MB)
1,000	7	7.1
3,001	5.8	7.4
5,000	6.5	7.9
10,000	5.8	8.7
25,000	10.9	6

50,000	14.1	6
100,000	19.4	6

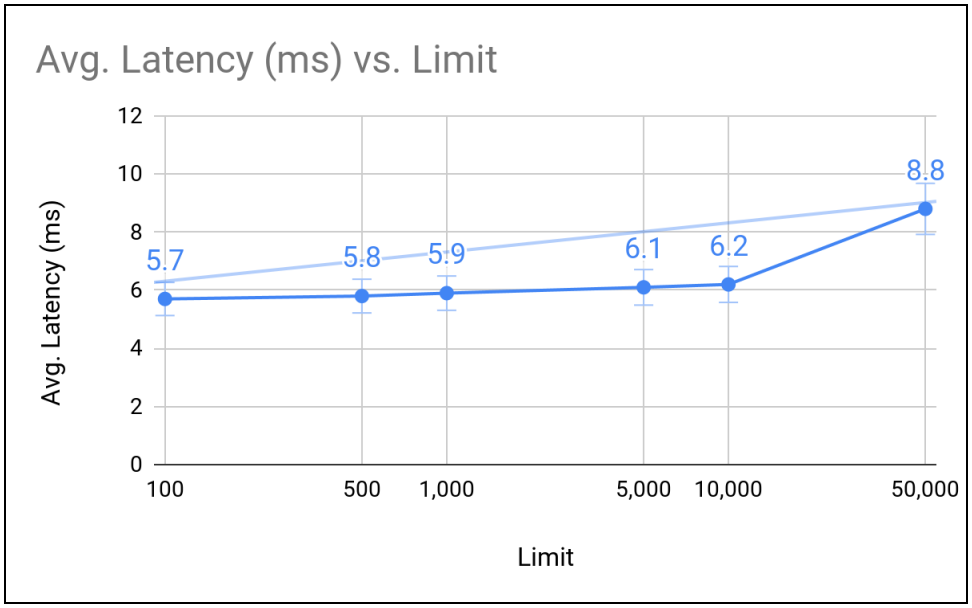


Latency & Server App Memory vs DB Memory Size

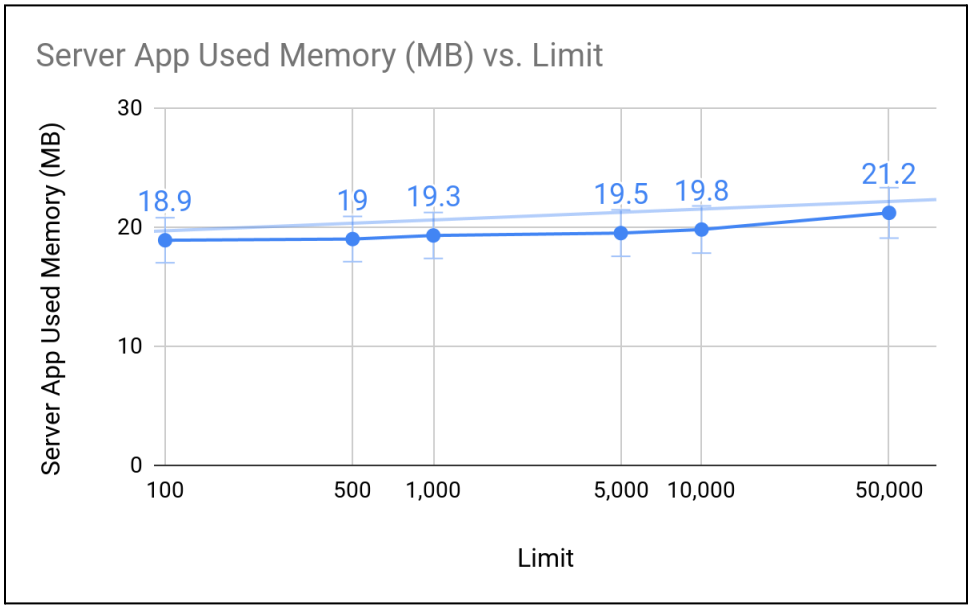
6. Latency Testing by Visitor Limit

The latency test by visitor limit will test the server application's speed in handling the processes related to allowing entry/exit and checking the validity of the ID. The client threads will be set to a value of 1000 threads to act as the controlled variable. It is expected that there will be an increase in latency as the set LIMIT value increases.

Limit	Avg. Latency (ms)	Server App Used Memory (MB)
100	5.7	18.9
500	5.8	19
1,000	5.9	19.3
5,000	6.1	19.5
10,000	6.2	19.8
50,000	8.8	21.2



Avg. Latency vs Limit

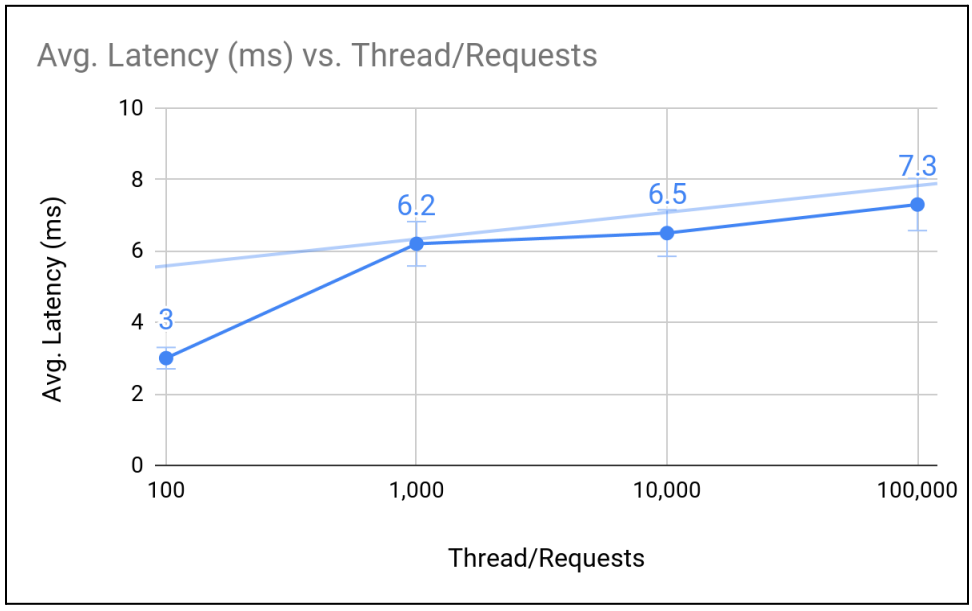


Server App Memory vs Limit

7. Latency Testing by Client Thread/Request Limit

The latency test by client thread/request limit will test the server application's overall speed in handling the client from receiving and processing the request and then returning a response to the client. The LIMIT value will be set to 100000 to act as the controlled variable. It is expected that there will be an increase in latency as the no. of simultaneous threads/requests increases.

Thread/Requests	Avg. Latency (ms)
100	3
1,000	6.2
10,000	6.5
100,000	7.3



Avg. Latency vs Thread/Request

8. Noise Test

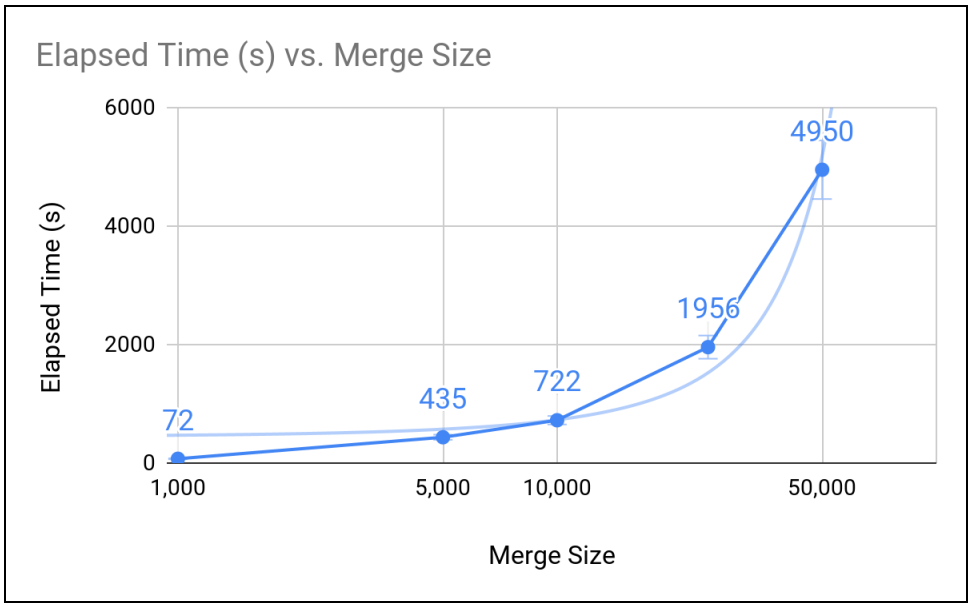
The noise test will test the server application's reliability in handling simultaneous invalid tap requests (i.e., invalid ID nos.) alongside valid tap requests. The valid tap requests will be set at a constant thread count of 1000 threads, a LIMIT value of 100000, and an invalid ID size of 10000 IDs to act as controlled variables.

Invalid Thread Count	Passed
10	OK
100	OK
1000	OK
10000	OK

9. Merge Test

The merge test will test the server application's reliability to handle simultaneous tap and add requests.

Merge Size	Elapsed Time (s)
1,000	72
5,000	435
10,000	722
25,000	1956
50,000	4950



Elapsed Time vs DB Merge Size

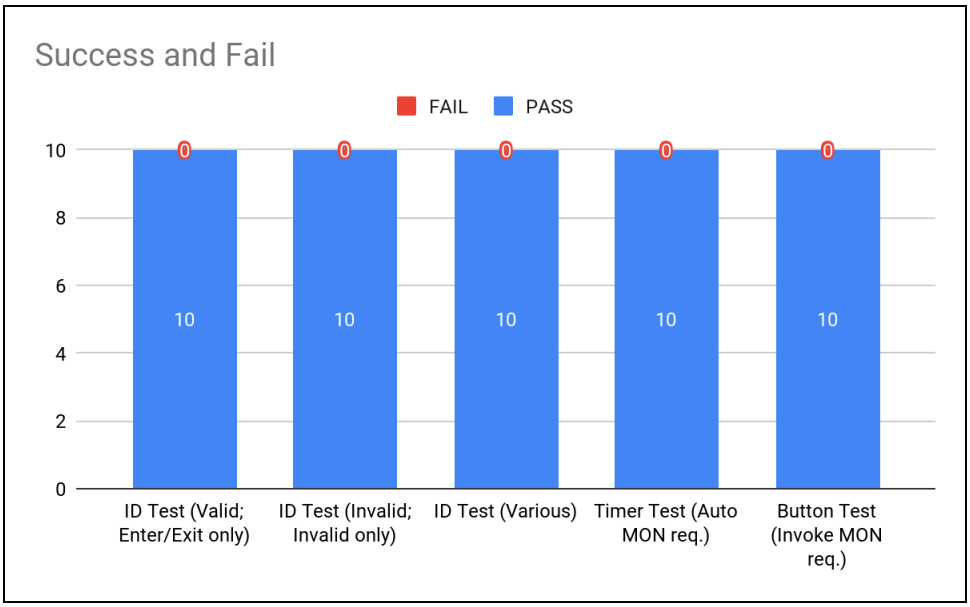
Embedded System Results and Analysis

Functionality tests were conducted for the 5 main features:

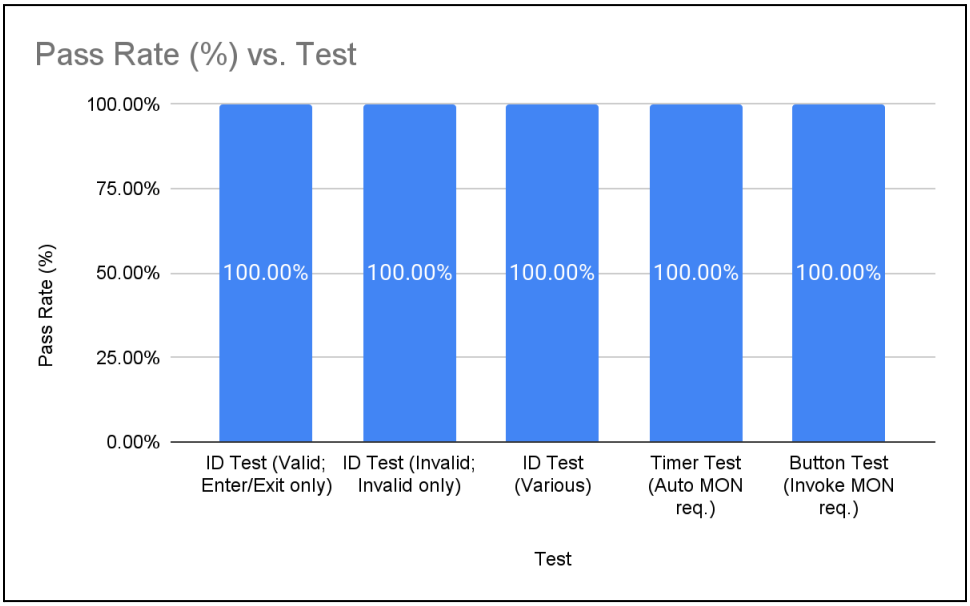
1. Tapping a Valid ID
2. Tapping an Invalid ID
3. Tapping a Valid and Invalid ID
4. Timer-based MON requests
5. Button-interrupt-based MON requests

The tables and graphs below shows the results of the tests executed 10 times for each feature/functionality.

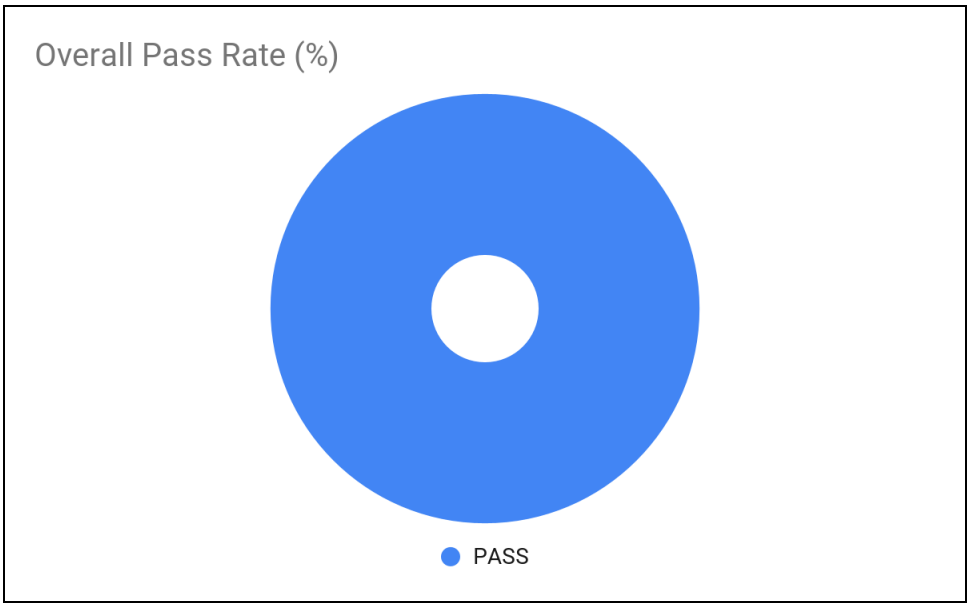
Test	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
ID Test (Valid; Enter/Exit only)	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
ID Test (Invalid; Invalid only)	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
ID Test (Various)	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
Timer Test (Auto MON req.)	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
Button Test (Invoke MON req.)	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS



Function Test of the Embedded System



Embedded System Function Pass Rate



Embedded System Function Overall Pass Rate

Conclusion

Server Application

Based on the objectives presented, the group was able to design and implement a reliable server application system that is capable of accurately counting the number of persons on a given premise. The server was able to do different functions concurrently without interrupting each other thread's operations. The server is also designed to handle errors (e.g., invalid data, disconnection, etc.) gracefully such that it won't interrupt the server application's runtime.

Such operations include concurrently/simultaneously receiving a request from the client, validating the received request and returning an appropriate response, updating the DB in memory and in the DB file. It was achieved by using the different server application design principles and techniques such as threading and locks/semaphores.

Based on the simulated testing and evaluation tests conducted, it was determined that the server application can handle up to 100,000 simultaneous requests, reaching an average latency of ±7.3ms (assuming minimal terminal console overhead). It was also determined that the server application is lightweight as it only requires less than 30MB of memory at a 50K Limit with 100K IDs in memory.

Hence, it can be concluded that the project is considered a functional success as it complies or even exceeds the requirements presented.

Embedded System

In summary, the various lessons learned from the course of NSEMBED (such as GPIO, threading, timers, interrupts, and WiFi), as well as self-learned aspects such as integrating and using an RFID module, resulted in the successful completion of the RFID Embedded System as part of the Crowd Counting System (CCS) project for NSAPDEV and NSEMBED.

The system is capable of reading an ID tag which then turns on the proper LED based on the response of the server. In addition, the embedded system is also capable of concurrently making requests to the server for the population data of the premises in either a timely or on an on-demand basis. All target functionalities passed the tests conducted to ensure its reliability and working order.

References

Getting Started with ESP32. (n.d.). Last Minute Engineers. Retrieved August 6, 2023, from https://lastminuteengineers.com/getting-started-with-esp32

In-Depth: What is RFID? How It Works? Interface RC522 with Arduino. (n.d.). Last Minute Engineers. Retrieved August 6, 2023, from https://lastminuteengineers.com/how-rfid-works-rc522-arduino-tutorial/

Python. (n.d.). *threading — Thread-based parallelism — Python 3.11.4 documentation*. Python Docs. Retrieved August 6, 2023, from https://docs.python.org/3/library/threading.html

Sato, Y. (2023, June 13). *'200 Exits?!' - Foreign Visitors Share Tales of Confusion in Tokyo's Shinjuku (and Urban Enchantment) | LIVE JAPAN travel guide*. LIVE JAPAN Perfect Guide. Retrieved August 6, 2023, from https://livejapan.com/en/in-tokyo/in-pref-tokyo/in-shinjuku/article-a0004717/