University of Stuttgart
Institute of Industrial Automation and
Software Engineering

Integration of Realtime Caching in a Dynamic Geofence System for Indoor Logistics

Study Project

Submitted at the University of Stuttgart by
Sowjanya Krishna

INFOTECH

Examiner: Prof. Dr.-Ing. Michael Weyrich
Supervisor: Hannes Vietz, M.Sc

19.02.2023

Table of Contents

TABLE OF CONTENTS ..... **II**
TABLE OF FIGURES ..... **III**
TABLE OF TABLES ..... **IV**
TABLE OF ABBREVIATIONS ..... **V**
GLOSSARY ..... **VI**
ABSTRACT ..... **VII**
1 INTRODUCTION **8**
1.1 Objective 8
2 CONCEPTUAL IDEA **10**
2.1 Caching System 10
2.2 Real-Time Caching system 11
2.3 Analysis of different cache tools 12
2.4 Redis Real-Time Cache Database 12
3 IMPLEMENTATION **15**
3.1 System Architecture 15
3.2 Backend 16
3.2.1 Flow of Program Execution 16
3.2.2 Redis implementation 17
4 RESULTS **21**
5 BIBLIOGRAPHY **24**
DECLARATION OF COMPLIANCE **25**

Table of Figures

Figure 1: Caching architecture [3] 10
Figure 2: Redis Insight 14
Figure 3: System Architecture 15
Figure 4: Flow of execution 17
Figure 5: Redis on Docker 18
Figure 6: RedisInsight 18
Figure 7: Redis stream data 20
Figure 8: Tag detection a) Tag is outside the geofence b) Tag is in the blue zone of the geofence c) Tag is in the red zone of the geofence 21
Figure 9: Warning a) Tag is outside the geofence b) Tag is in the blue zone of the geofence c) Tag is in the red zone of the geofence 22
Figure 10: Enlarge/Shrink a) Geofence shrunk as AGV speed is less b) c) Geofence is enlarged as AGV speed is high 23

Table of Tables

Table 1: Redis data types 19

Table of Abbreviations

MQTT	**M**essage **Q**ueuing **T**elemetry **T**ransport
AGV	**A**utomated **G**uided **V**ehicle
ARENA	**A**ctive **R**esearch **E**nvironment for the **N**ext generation of **A**utomobiles
IIoT	**I**ndustrial **I**nternet **o**f **T**hings
URL	**U**niform **R**esource **L**ocator
GUI	**G**raphical **U**ser **I**nterface
JSON	**J**ava**S**cript **O**bject **N**otation
SQL	**S**tructured **Q**uery **L**anguage
HTTP	**H**ypertext **T**ransfer **P**rotocol
ORM	**O**bject-**R**elational **M**apping
API	**A**pplication **P**rogramming **I**nterface

Glossary

Geofence	A virtual geographic boundary, defined by GPS or RFID technology, that enables software to trigger a response when a mobile device enters or leave a particular area.
Back-end	Refers to parts of a computer application or a program's code that allow it to operate and that cannot be accessed by a user.
Front-end	The frontend of a software program or website is everything with which the user interacts
AGV	Self-guided vehicles are material handling systems or load carriers that travel autonomously throughout a warehouse, distribution center, or manufacturing facility, without an onboard operator or driver
Localization	Deployment of networks of sensors, able to collect and transmit data in order to determine the targets position
Redis	Redis is a distributed, in-memory key-value database, cache, and message broker with optional durability that is used as an in-memory data structure store.
Caching	The process of storing data into a cache, a temporary storage designed to speed up access to data and boost system and application performance.

Abstract

Geofences are virtual zones used to control and influence the movement of driverless transport systems, such as AGVs. Geofences are critical to the functioning of modern indoor logistic systems. To control and influence the AGVs' logistical flow, they establish virtual zones on internal maps, limiting their operation within a specified area and preventing collisions with objects and other AGVs. The location data of these AGVs is published using message protocols.

Real-time monitoring of geofences is essential to adaptively change them during runtime and place them correctly around moving AGVs. This requires constant monitoring of the current position of all AGVs. Real-time locating systems such as GPS or RFID systems publish location information using message protocols like MQTT, enabling real-time location monitoring, and tracking across numerous devices and systems. However, an event-driven approach like MQTT may not be suitable for applications that require more synchronous or real-time connectivity with other devices and systems. Additional tools or middleware may be required to buffer and handle the location data in a more organized and predictable manner to ensure efficient and prompt use of location data.

This paper aims to integrate a real-time caching system into an existing system for dynamic geofences used in indoor logistic systems, which are often based on driverless transport systems. The current geofence system uses MQTT messages to receive the locations of AGVs in real-time, but it faces challenges in combining with other tools that do not work asynchronously. The proposed solution is to split the system into two applications - one that receives MQTT messages and stores them in a cache, and another that runs a Flask web server that reads information from the cache. The integration of the real-time caching system improves the responsiveness and scalability of the system and enables a visualization front end to receive the latest coordinates of geofences and AGVs via a REST API.

Key Words: *Geofence, Automated Guided Vehicles(AGV), Redis, Realtime caching*

1 Introduction

Geofencing is a technology that uses GPS or RFID to create a virtual boundary around a physical location. Geofencing is used in the context of Automated Guided Vehicles (AGVs) to limit the region in which the AGV can operate, and to prevent it from operating outside that area. Because it enables AGVs to function securely and effectively inside a specified area, geofencing is an important tool for them. AGVs can stay on course and within schedule by employing geofencing to prevent collisions with objects and other AGVs.

Safety-critical indoor localization of mobile robots through 5G is one of the most promising application areas. The use of automated guided vehicles (AGV) is expanding, and there is a growing demand for reliable vehicle management and monitoring [1]. Real-time locating systems, such GPS or RFID systems, publish location information using message protocols like MQTT (Message Queuing Telemetry Transport). It is feasible to create real-time location monitoring and tracking across numerous devices and systems by using MQTT. This enables organizations to make better decisions based on precise and current location data and can be valuable in a variety of applications, such as monitoring, fleet management and supply chain logistics.

Although while MQTT is very effective and can send position changes rapidly and reliably, it might not be well-suited for applications that need more synchronous or real-time connectivity with other devices and systems. To make sure that the location data is used efficiently and promptly, for instance, it may be difficult to combine a location tracking system with a legacy system that is not event-driven. Also, it could be challenging to synchronize the data with other systems that demand more constant and regular updates if the location updates are coming in at irregular intervals. To connect the event driven MQTT protocol with the other systems in these circumstances, additional tools or middleware may be required. It can be necessary to construct a message queue, a caching layer, or other technologies to buffer and handle the location data in a more organized and predictable manner.

1.1 Objective

The goal of this project is to identify and integrate a real-time caching system that stores MQTT messages for other processes to read concurrently and in real-time. To ensure that multiple processes can read from the cache concurrently and in real-time, use of thread-safe data structure such as a thread-safe queue or a thread-safe dictionary must be implemented. Each process needs to read from the cache create a separate thread that reads messages from the queue and processes them. By splitting the geofence system into two applications, that they run independently of each other and avoid blocking each other due to the Global Interpreter Lock (GIL). This allows to

decouple the MQTT processing from the Flask web server and allows them to run independently of each other. It also allows multiple processes to read from the cache concurrently and in real-time, improving the scalability and responsiveness of the system.

2 Conceptual idea

This chapter incorporates concepts implemented in the project is covered in this chapter. This chapter explains the use cases that have been developed, how they have been analyzed.

2.1 Caching System

The cache is a component of hardware or software that stores data that can be retrieved more quickly from one source than from another. To keep track of frequently responding to user requests, caches are typically used. The results of prolonged computing operations can also be stored using it. To make data access faster, data is cached and stored at a location other than the primary data source. Caching is used to speed up a system performance significantly. We need to use caching to reduce a system's latency. Caches often contain the data that has been accessed most recently since it is possible that data that has been recently requested may be requested repeatedly. Caching is therefore necessary in these situations to reduce the amount of data that needs to be retrieved from the database [2].

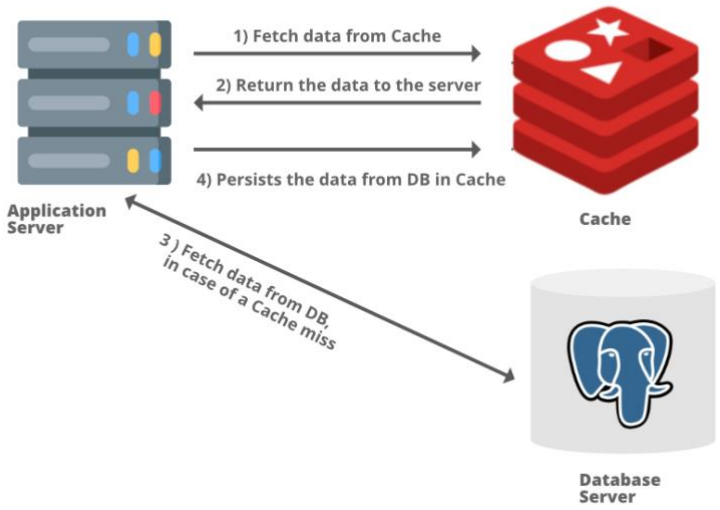


Figure 1: Caching architecture [3]

Application performance and scalability that depend on frequently requested data might benefit greatly from caching solutions. Consider the following while fitting a caching system:

- Choose the data that will benefit from caching: Not every data needs to be cached, thus it's critical to choose the data that will. This includes data that is frequently accessed, data that is slow to retrieve, or data that is used in an application that is crucial for performance [4].

- Choose the best caching technology: There are a variety of caching technologies available, and each has advantages and disadvantages. In-memory caching, disk caching, and distributed caching systems like Redis or Memcached are a few popular solutions. The technology selected will be determined by elements including the dataset's size, the performance level required, and the infrastructure at hand [5].
- Configure the caching system: After deciding on a caching technology, you must set it up so that it functions with your application. To reduce cache misses, this may entail optimizing cache placement, customizing cache size, and setting up cache expiration regulations [4].
- The caching system needs to be monitored and tuned because a variety of things can have an impact on its performance. It's crucial to continuously assess the caching system's performance and make any modifications. This could entail updating hardware, adjusting cache settings, or altering caching algorithms [5].

2.2 Real-Time Caching system

Real-time caching systems are designed to improve the performance and responsiveness of applications by storing frequently accessed data in a cache that can be quickly accessed by the application.

A real-time caching system works on the simple principle that before requesting data from a backend storage system on behalf of an application, it first determines if the requested data is already present in the cache. The caching system does not need to access the backend storage system to return data to the application if it is located in the cache. This can significantly cut down on the amount of time it takes for the program to retrieve the data.

A copy of the data that is most frequently accessed is kept in a fast storage system, like RAM or solid-state drives, using real-time caching systems. This enables the caching system to swiftly obtain data when an application requests it. The caching system also uses various algorithms to manage the cache and ensure that it contains the most relevant data.

There are various kinds of real-time caching systems, including distributed, in-memory, and disk caching. Disk caching stores data on the hard drive, whereas in-memory caching keeps it in RAM. Distributed caching uses many nodes to store data across a cluster, which can increase scalability and fault tolerance.

Web applications, mobile apps, and data processing systems are just a few examples of the many applications that can benefit from real-time caching solutions. These technologies can aid in enhancing user experience and boosting application performance by speeding up data access.

2.3 Analysis of different cache tools

There are several caching tools available, each with its own strengths and weaknesses. Here's an analysis and evaluation of some of the most popular cache tools:

1. Redis: Redis is an open-source, in-memory data structure store that functions as a message broker, database, and cache. Strings, hashes, lists, and sets are just a few of the many data structures that Redis offers. Redis is renowned for its speed and adaptability. Redis is appropriate for scalable and high-availability applications because it also supports replication and clustering.
2. Memcached: Memcached is a distributed memory object caching system with great speed. Memcached supports a small number of data structures, including strings and arrays, and is intended to be quick and easy to use. Web applications typically use Memcached to cache frequently used API replies or database queries.
3. Apache Ignite: An in-memory data grid called Apache Ignite offers distributed computation, data processing, and caching. Large datasets can be stored and processed using Apache Ignite, which is made to be scalable and highly available. Moreover, Apache Ignite provides SQL and key-value Interfaces, making it simple for developers to use.
4. Hazelcast: A distributed in-memory data grid that provides caching, data processing, and communications is called Hazelcast. Hazelcast is made to be extremely fault-tolerant and scalable, and it supports a variety of data structures like queues, sets, and maps. Hazelcast can process huge datasets or carry out concurrent computations because it also enables distributed computing.
5. Varnish: Varnish is a web application accelerator that can function as a reverse proxy, load balancer, or cache. Varnish is made to boost web applications' performance by caching frequently visited material, such images, or static files. Varnish supports a large number of HTTP capabilities, such as request routing rules, load balancing methods, and caching policies.

It's crucial to take into account aspects like performance, scalability, usability, and community support when comparing various cache technologies. For straightforward caching requirements, Redis and Memcached are common options, however more sophisticated use cases call for Apache Ignite, Hazelcast, and Varnish. In the end, the selection of a cache tool will be based on the particular needs of the application and the infrastructure at hand.

2.4 Redis Real-Time Cache Database

Redis is an in-memory data structure store that works as a message broker, cache, and database. Redis is used in web applications, real-time analytics, and other performance-critical systems because of its speed and adaptability.

Key features and benefits of Redis cache [6]:

1. In-memory caching: Redis is incredibly quick and effective in caching frequently requested data because it keeps data in RAM, where it can be accessed instantly. Strings, hashes, lists, and sets are just a few of the many types of data that Redis is capable of storing.
2. Persistence: Data can be stored to disk periodically or in real-time due to Redis' capability for data persistence. In the event of a system failure or restart, this guarantees that data is not lost.
3. High-availability: Redis is suited for high-availability and scalable applications since it allows replication and clustering. Redis has the option to automatically replicate data across several nodes, enhancing performance and fault tolerance.
4. Performance: Redis is known for its performance and low latency. Redis is excellent for real-time applications and high-throughput systems since it can handle millions of operations per second.
5. Community support: Redis has a large and active community that offers assistance, documentation, and third-party integrations. Redis is also open-source, so it can be modified to match particular needs and is free to use.

The overall performance and scalability of web applications, real-time analytics, and other performance-critical systems can be enhanced with the help of Redis, a strong and adaptable caching solution. Redis is a standalone cache that can be used alone or in conjunction with other caching tools to offer a more complete caching solution.

Redis supports various data types, each with its own set of commands and properties

1. String: The most fundamental data type in Redis is a string. They are capable of holding any kind of data, including binary, text, and numerical data. Redis offers a number of commands, including SET, GET, INCR, DECR, APPEND, and others, for working with strings.
2. Lists: Lists are collections of strings that are arranged. Redis offers several commands, including LPUSH, RPUSH, LPOP, RPOP, LLEN, LRANGE, and others, for working with lists.
3. Sets: Sets are collections of distinct strings that are not sorted. Redis offers several commands, including SADD, SREM, SMEMBERS, SINTER, SUNION, and others, for working with sets.
4. Sorted sets: Sorted sets resemble sets, except each component also gets a score. Redis offers several commands, including ZADD, ZREM, ZRANGE, ZSCORE, and others, for working with sorted collections.
5. Hashes: Maps between string field names and string values are known as hashes. Redis offers a number of commands for interacting with hashes, including HSET, HGET, HDEL, HKEYS, and HVALS.

6. Bitmaps: A unique kind of text known as a bitmap represents a bit array. Redis offers a number of bitmap-related commands, including SETBIT, GETBIT, BITCOUNT, BITOP, and others.
7. HyperLogLogs: A probabilistic data structure for counting unique elements in a set is called HyperLogLogs. Redis offers several commands, including PFADD, PFCOUNT, PFMERGE, and others, for working with HyperLogLogs.
8. Streams: An append-only log-like data structure is a Redis stream. With streams, you may simultaneously syndicate and record live events. Redis stream use cases include, for instance: event sourcing (e.g., tracking user actions, clicks, etc.),sensor tracking (e.g., readings from devices in the field)

RedisInsight combines the Redis CLI and a graphical user interface to handle any Redis deployment. Visual interaction and viewing of data are both possible.

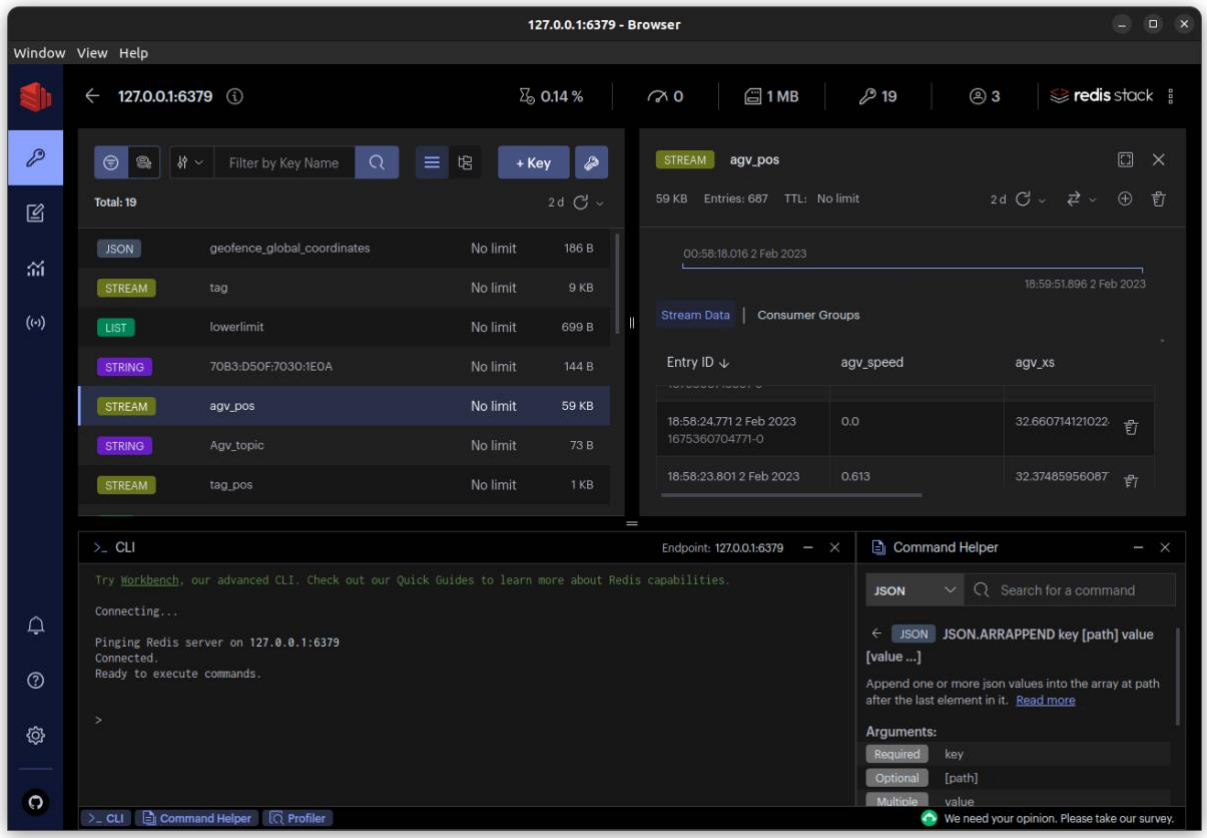


Figure 2: Redis Insight

3 Implementation

The concepts that have been implemented are thoroughly explored in a technical manner. First, the software architecture is described, along with existing localization system. Second, the backend implementation and sequence flow is explained.

3.1 System Architecture

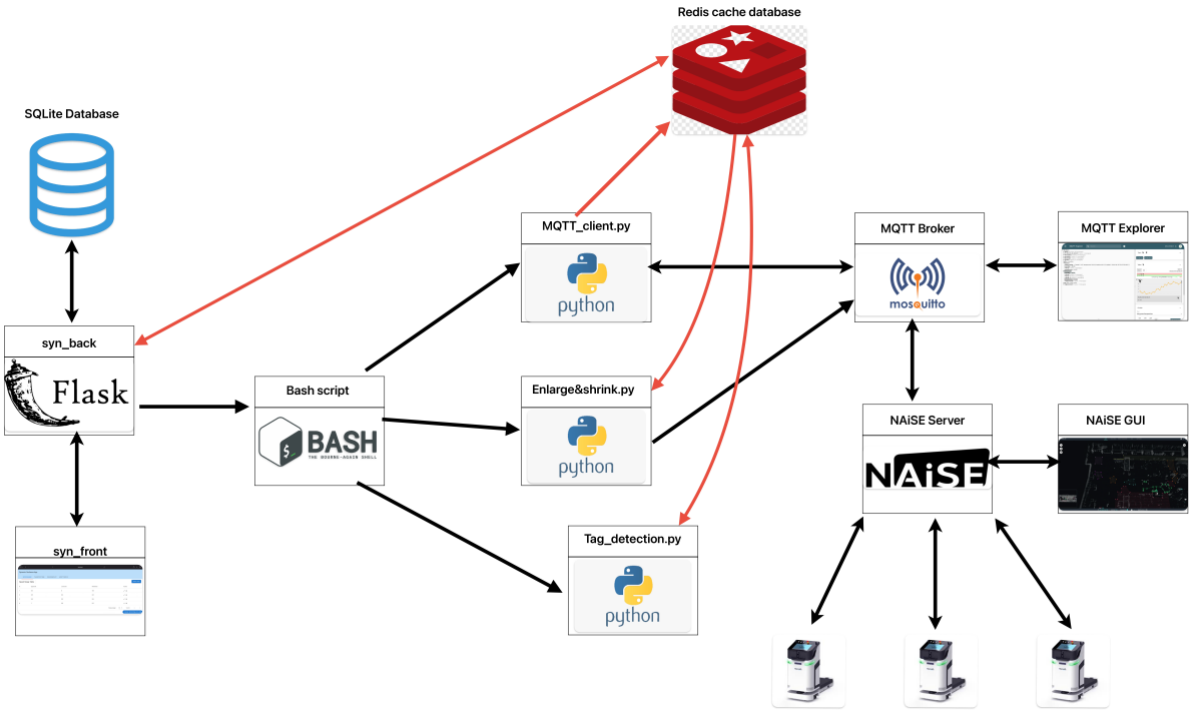


Figure 3: System Architecture

In Figure 3, the implemented architecture can be seen with all modules defined. The backend module is implemented in Flask python framework. All the communication towards the remaining modules is initiated through this module. The database used is SQLite as a backend database. The client side (User interface) is implemented on Vuejs. The user input from the frontend is communicated to the SQLite database using the library SQLAchemy which enables Object-Relational Mapping (ORM). This practically means that the operations done on SQLite database can be initiated from flask using Python queries. This speeds up the process of converting SQL queries into python. The AGV's (Rexroth active shuttle) communicate to the NAiSE server, an MQTT broker is implemented as communication protocol to enable bidirectional communication. To overcome the issue of multithreading, Redis real-time caching system is integrated to store MQTT messages and support concurrent reads. When the connect API is called, the flask backend launches a subprocess that executes the bash script, which launches three Python applications and the redis-cache that was launched on a Docker container simultaneously. The MQTT-client application establishes a connection with the MQTT broker, subscribes to all the messages, and

streams to the Redis cache, all the necessary data in specific data types. The Geofence Enlarge Shrink and Tag Detection applications read the necessary streamed data from the Redis cache and publish the output back to the frontend in real-time and on HTTP requests. NAiSE GUI rendered from NAiSE server which provides as 3D map of the ARENA with real-time movements of the AGV's, Tags and with a feature to draw polygon shaped geofences.

3.2 Backend

Flask framework is utilized to implement the web application. This centralized component communicates with all the modules. Flask runs the application locally at the address http://127.0.0.1:5000/. In Flask, routes are the primary interface for the application or the end user, they are frequently built using the URL pattern. The backend is also implemented with Redis caching system. The geofence system shall be split into three applications to resolve the multithreading issue, Just one thread can ever execute Python bytecode at once because of the Global Interpreter Lock (GIL) feature of the language. This means that multiple threads cannot execute Python code simultaneously on multiple cores or processors. To address this issue, the applications are run asynchronously, that can run concurrently without blocking each other and the data is exchanged via redis caching. This way, the incoming MQTT messages and HTTP requests are handled concurrently without suspending Flask during the processing of MQTT messages.

3.2.1 Flow of Program Execution

In this section, the flow of the program execution and the method implemented to run parallelly the two dynamic geofence features (enlarge/shrink, tag detection) will be briefly explained.

In order to launch the app, run the entrypoint.sh bash script in terminal, which simultaneously launches the frontend syn_front, the syn_back flask application, and the docker command to start redis-cache database. When a user enters data in the frontend and clicks the connect button on the MQTT Topics webpage, the HTTP request is sent to flask and connect API call is invoked, which runs the entryscreens.sh bash script, which simultaneously executes three python programs, and adds the user's data to the SQLite primary database and to Redis cache database in JSON-format. The two algorithms have different dynamicity factors affecting the geofence. Enlarge and shrink algorithm, varies the geofence coordinates by redrawing the original shape of the geofence. In Tag detection algorithm, the geofence is divided into multiple zone areas maintaining the geofence's geometry. Both the algorithms enlarge_shrink and tag_detection is running in endless loops, independent of each other. Since the flask framework is written in python and one of python's main weakness is its inability to have true parallelization due to global interpreter lock. Integrating a real-time caching system is implemented to deal with this problem.

Figure 4 represents a graphical overview of the program. The MQTT client program subscribes to the AGV, tag, and geofence topics and adds the user data from to Redis. The mqtt_client receives MQTT messages at irregular intervals, and Redis' streaming

concept is used to add this data to the cache in real-time. Enlarge Shrink and Tag Detection algorithms run separately from one another, reading the necessary streaming data from Redis in real-time, and running the algorithm in an endless loop:

- The Enlarge Shrink program updates the geofence coordinates and sends them to the MQTT broker.
- Tag zone warnings are streamed into the redis cache by tag detection, and when a user requests them at the front end, Flask does not suspend itself and responds to the HTTP request right away with a tag detection API call that retrieves the most recent warning information from the redis cache and displays it to the user.

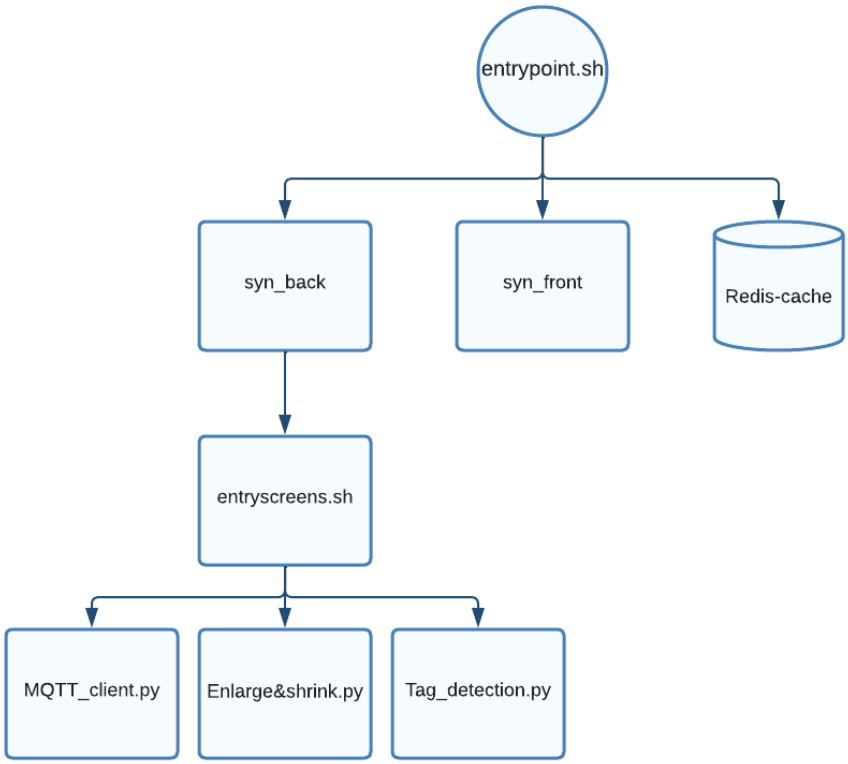


Figure 4: Flow of execution

3.2.2 Redis implementation

Running Redis on Docker can make it easier to deploy and manage Redis instances, especially in a containerized environment. Pull the Redis image redis/redis-stack which features RedisInsight and the Redis Stack server. Due to the inbuilt RedisInsight's ability to see data, this container is ideal for local development.
To start a Redis Stack container using the redis-stack image, the following command is run on the terminal [7]:

```
docker run -d --name redis-stack -p 6379:6379 -p 8001:8001 redis/redis-stack:latest
```

This will start a Redis container with the name “redis-cache" this can be used in the applications or connect to it using a Redis client.

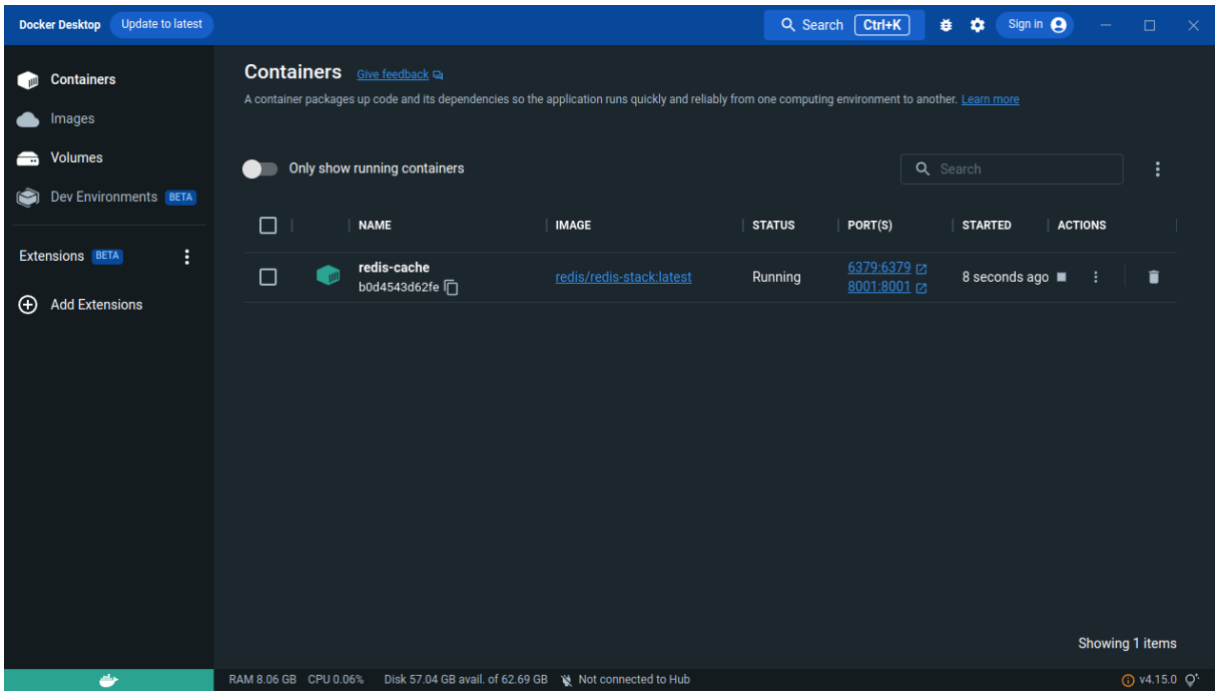


Figure 5: Redis on Docker

The docker run command previously mentioned also makes RedisInsight accessible on port 6379. Pointing the browser to localhost:8001 will allow the usage of RedisInsight.

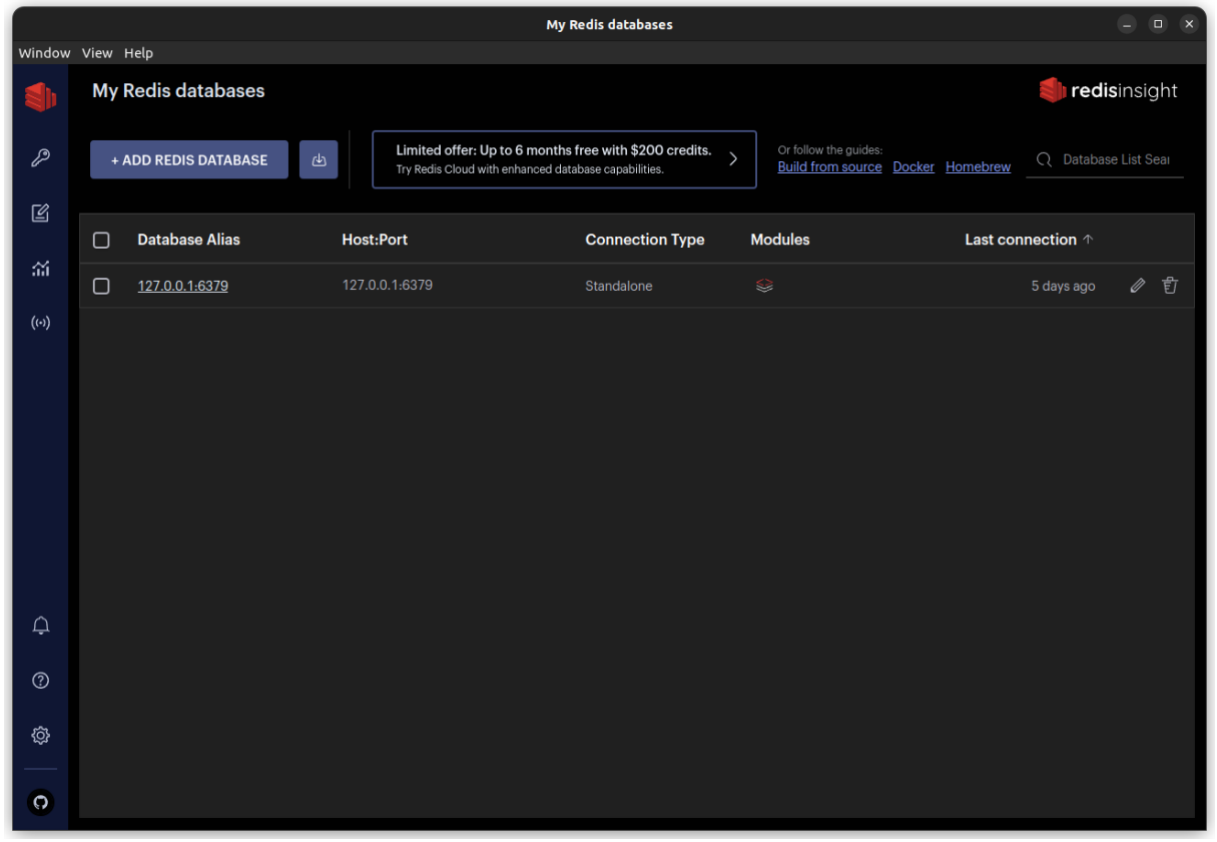


Figure 6: RedisInsight

In Redis, the key name is a string value that uniquely identifies a specific data item stored in the database. When using Redis, it's important to choose meaningful and descriptive key names to make it easy to locate and manage the data in the database. The below table gives an overview of all the keywords and their datatypes, used in the application.

Table 1: Redis data types

Key Type	Key Name	Data	Description

JSON	user_input	{
		"Agv_name":"aagv7"
		"Agv_topic":"naise/tag/aagv7"
		"Naise_tag":"naise/tag/041E"
		"Geofence_id":"Geofence"
		"upperlimit":[
		"0":0.2
		"1":0.5
		"2":0.8
		"3":1]
		"lowerlimit":[
		"0":0
		"1":0.2
		"2":0.5
		"3":0.8]
		"off_set":[
		"0":-0.2
		"1":0.3
		"2":0.6
		"3":0.9]
		"zone_offset_value":0.2
		}
			User input data from the frontend

STREAM	agv_pos	Entry ID:1675360791896-0 agv_speed: 0.613 agv_xs: 31.2820039401 agv_ys: 20.1437074772 agv_angle: -0.0958837	AGV data from MQTT client
STREAM	geofence_xcoordinates	Entry ID: 1677514478360-0 geofence_xcoordinate0: 24.5865239 geofence_xcoordinate1: 27.5133114 geofence_xcoordinate2: 27.5540146 geofence_xcoordinate3: 24.4856846	Geofence global x-coordinates from MQTT client
STREAM	geofence_ycoordinates	Entry ID: 1677514478361-0 geofence_ycoordinate0: 24.4605646 geofence_ycoordinate1: 24.3217445 geofence_ycoordinate2: 21.7643338 geofence_ycoordinate3: 21.8416749	Geofence global y-coordinates from MQTT client
STREAM	tag_pos	Entry ID: 1677514899592-0 tag_x: 26.7 tag_y: 22.93	Tag coordinates from MQTT client
STREAM	warning	Entry ID: 1677515689434-0 warning: Tag is inside the geofence red-zone	Tag detection warning to frontend

Depending on the unique use case and requirements, there are various ways to integrate real-time sensor data to Redis. Redis stream is the best choice for this application's use case. Redis Streams is a data structure that enables the storage and processing of real-time data. By creating a stream for each sensor and adding new data to the stream as it becomes available, Redis Streams is used to add sensor data to Redis. The use of XADD in Redis will immediately create a new ID for that consists of a dash, a sequence number, and a timestamp with millisecond precision. For instance, 1656416957625-0. Then, provide the field names and values to be stored in the new stream entry after that.

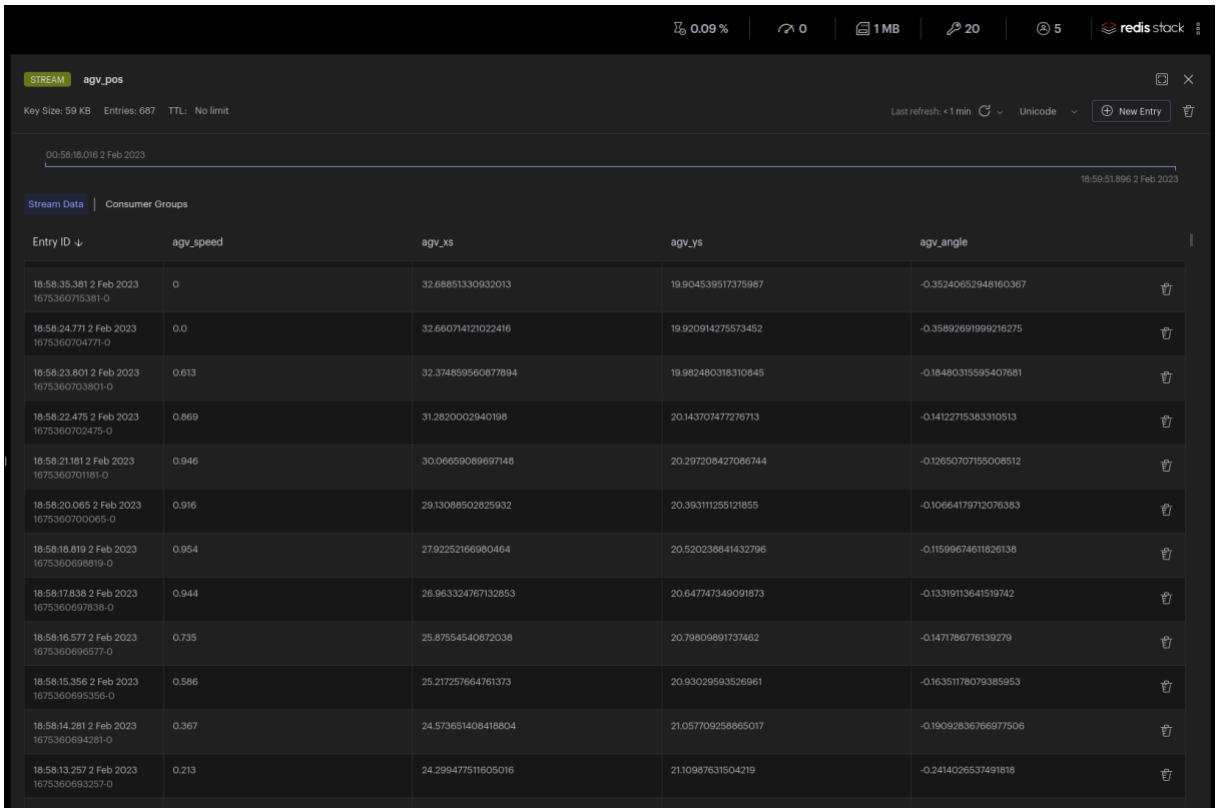


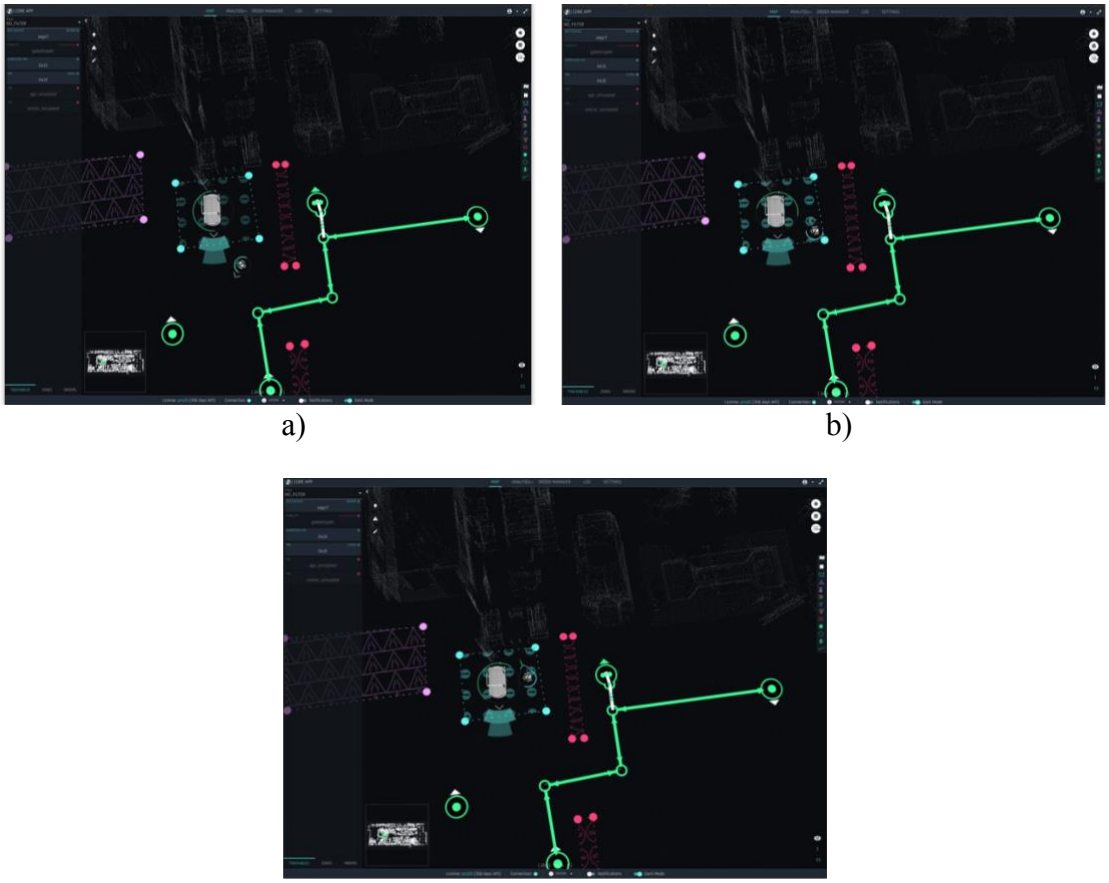
Figure 7: Redis stream data

Redis Streams is an excellent option for this use case for the following reasons:

- Ordered data: Redis Streams enables to store data in the order in which it was received, which is crucial for real-time applications where the order of the data is crucial. This guarantees that events are handled in the proper order and that data is processed in the proper order.
- Compact data storage: Redis Streams saves data in a very compact format, which lowers the amount of storage space needed for massive amounts of real-time data. This is crucial when working with high-volume data streams since it lowers storage expenses.
- Stream processing capabilities: Redis Streams offers built-in stream processing features that allow to process real-time data as it arrives. This includes assistance with data collection, filtering, and transformation, making it simple to extract real-time insights from the data stream.
- Scalability: Redis Streams is made to expand horizontally, so you can quickly add extra servers to manage growing data volumes. This makes it a strong choice for real-time applications where data volumes are projected to expand dramatically over time.

4 Results

The approach taken to solve the concurrency issue in the geofence system was to split it into two applications. One application was responsible for receiving and processing MQTT messages and saving them to a real-time caching system, while the other application ran the Flask web server and read the required information from the cache. By doing so, the issue of Flask being suspended during message processing due to the Global Interpreter Lock was avoided.
To improve the responsiveness and scalability of the system, a real-time caching system, such as Redis, was integrated to store MQTT messages in real-time for other processes to read from the cache. This allowed multiple processes to read from the cache concurrently and in real-time by using a thread-safe data structure.
By implementing this approach, the geofence system was able to operate independently of each other, improving the scalability and responsiveness of the system. Additionally, by using a real-time caching system, the system was able to handle a larger volume of messages and process them in real-time.



a) b)

c)

Figure 8: Tag detection a) Tag is outside the geofence b) Tag is in the blue zone of the geofence c) Tag is in the red zone of the geofence

Figure 8 depicts the Tag detection function, which uses a user-defined zoneoffset percentage to divide the geofence enclosing the AGV into two zones: blue and red. The geofence system detects which zone the tag is in based on its location and sends a warning to the frontend. With the integration of the cache system, the warning is now

streamed into the cache and can be accessed through the Flask API call. This approach allows for more efficient retrieval of the latest warning by the user, since the data is stored in the cache and can be accessed without needing to process new requests every time. Overall, the integration of the real-time caching system has improved the system's responsiveness and reduced the workload on the Flask server. Figure 9 shows the warning messages of the tag.

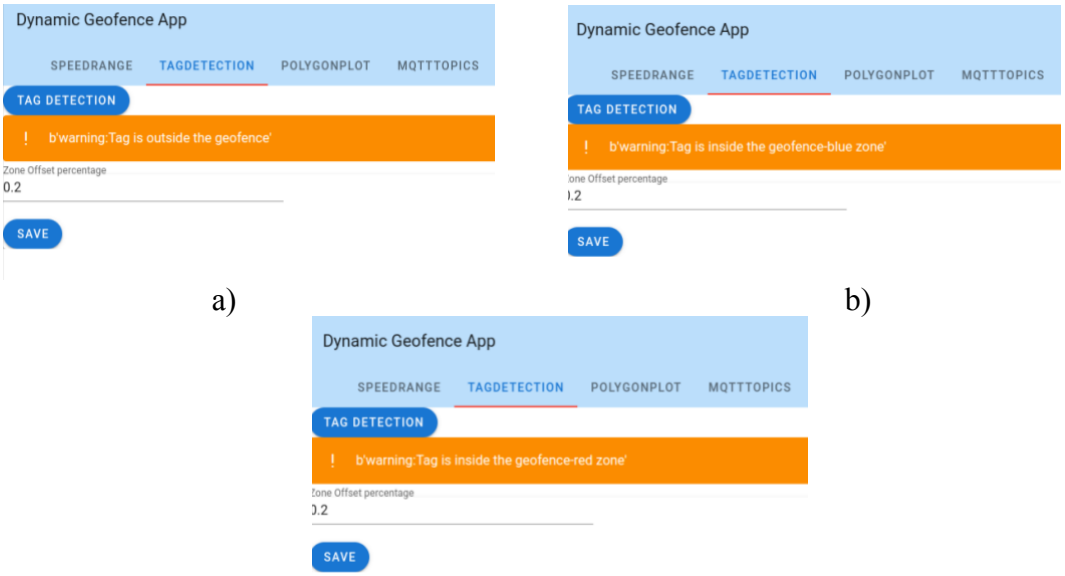
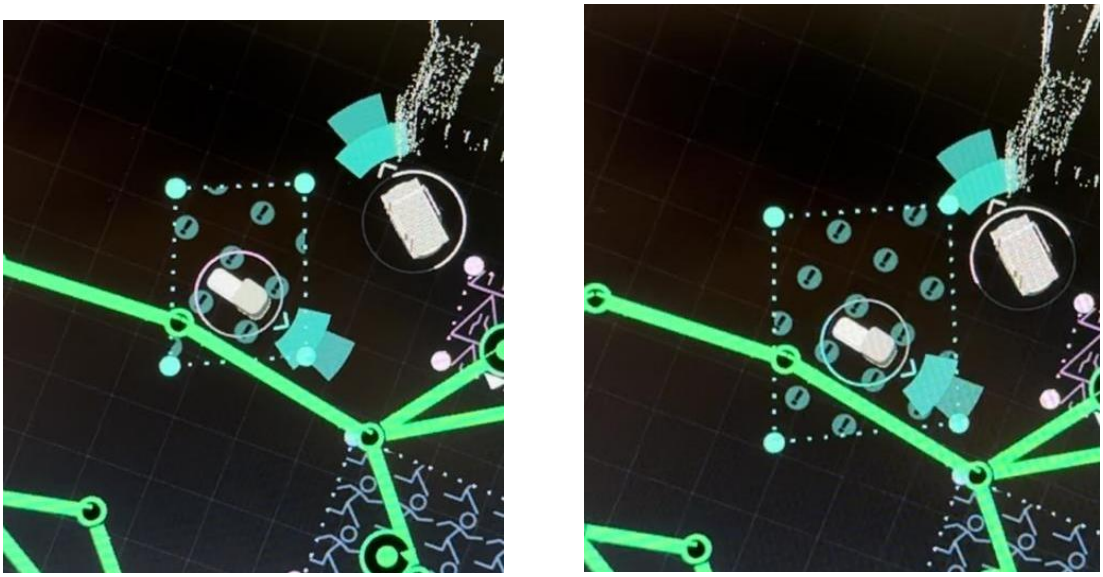
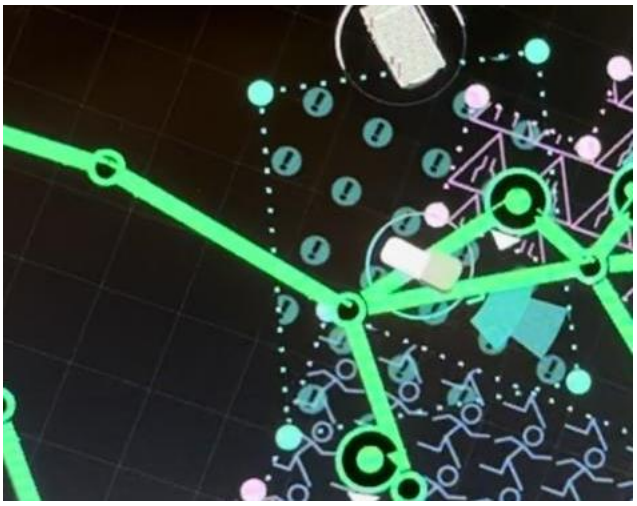


Figure 9: Warning a) Tag is outside the geofence b) Tag is in the blue zone of the geofence c) Tag is in the red zone of the geofence

Enlarging and shrinking of geofence takes place as the speed of the AGV is varied as seen in Figure 10. The data from the speedrange table is retrieved, and the appropriate offset value is applied based on the AGV speed.



a b



c)

Figure 10: Enlarge/Shrink a) Geofence shrunk as AGV speed is less b) c) Geofence is enlarged as AGV speed is high

Performance metrics such as response time: The time it takes for the system to respond to an API call, throughput: The amount of data processed by the system in a given time period after the implementation of the real-time caching system determines the improvement in throughput and scalability: The ability of the system to handle larger volume of messages and concurrent connections by gradually increasing the load on the system. The system's performance and response times show the effectiveness of the solution in improving the geofence system's concurrency issue and responsiveness.

5 Bibliography

[1] Fraunhofer Institute for Integrated Circuits IIS, "5G Positioning in Industry and Logistics 4.0," [Online]. Available: https://www.iis.fraunhofer.de/en/ff/lv/lok/5g/industrie_logistik/ueberwachung.html. [Accessed 25 April 2022].

[2] A. Chakraborty, "System Design Basics: Getting started with Caching," TowardsDataScience, 2022.

[3] geeksforgeeks, "geeksforgeeks," [Online]. Available: https://www.geeksforgeeks.org/caching-system-design-concept-for-beginners/.

[4] J. M. a. I. Nunes, "A Qualitative Study of Application-level Caching," vol. 14, 2015.

[5] D. S. Berger, "Design and Analysis of Adaptive Caching Techniques for Internet Content Delivery," University of Kaiserslautern.

[6] Redis, "Redis Cache Database," [Online]. Available: redis.io.

[7] redis, "Run redis stack on docker," [Online]. Available: https://redis.io/docs/stack/get-started/install/docker/.

Declaration of Compliance

I hereby declare to have written this work independently and to have respected in its preparation the relevant provisions, in particular those corresponding to the copyright protection of external materials. Whenever external materials (such as images, drawings, text passages) are used in this work, I declare that these materials are referenced accordingly (e.g. quote, source) and, whenever necessary, consent from the author to use such materials in my work has been obtained.

Signature: 

Stuttgart, on the <19.02.2023>