**Implementing Real-Time Database Server in ROS Platform Using Microservices Architecture**

Rahul Sathiyamoorthi

School of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, 5000, Australia.

[a1784128@student.adelaide.edu.au](mailto:a1784128@student.adelaide.edu.au)

**Abstract**

Different sub-systems carry out various functions to perform several tasks but struggle to act as a coordinated system and synchronise the data between them. However currently developed systems are typical monolithic architecture followed by a top-down design. The operation and maintenance of the systems get harder when the scale of the application gets larger. In this paper, a microservices based architecture that integrates different services and synchronises the data between different sub-systems is proposed to solve the present dilemma. A service management architecture based on microservices is designed to provide a highly efficient development mechanism. The existing diverse robot software packages of Robot Operating System (ROS) are used to deploy the microservices technology without any additional software. The server synchronises different services and important data processing algorithms which are decoupled in the server that increases the computational power and response time of the robots. A centralised database is used which stores all the information of the robots and the scenario undertaken, which will be helpful in post-mission user analysis, query data if the main robot shuts down, and finding the robots when lost in the buildings by tracking the path coordinates of the robots.

*Keywords –* database server, microservices, ROS, system integration.

**1. Introduction**

In hazardous environments for humans, a multi-robot system can be used to explore in extremely limited disaster conditions and to determine the location of a threat. The fundamental theory behind a multi-robot system suggests assigning smaller sub-problems to individual robots in a network and allowing them to communicate with one another to solve complex problems [1]. The obvious benefit of multi-robot exploration is its concurrency, which can significantly reduce the mission time. This project focuses on narrowing the gap of localising hazards by using a multi-robot system. The function of the robots is distributed accordingly, the explorer robot constantly explores the whole area to identify potential hazards and creates the map. Secondly, the warden robot acts as a warden for the identified hazard. Finally, the guide robot identifies the people who are stranded inside the hazardous area using Bluetooth Low Energy (BLE) beacons. A custom database server is deployed for data management, data synchronisation between robots, and post-mission analysis. The scope of the project is classified into three individual sub-divisions,

1. Hazard detection and localisation using sensor fusion.
2. Implementing BLE beacons for identifying people stranded in the hazardous area.
3. Implementing a database server for data management and integrating the whole system.

This paper is discussing the development of a database server in the ROS platform by following microservices architecture.

With the rapid growth of sensor systems, the amount and type of information that needs to be processed by the robot’s onboard processors is rapidly increasing. However, the execution capabilities of robots are also limited to their onboard devices, which significantly increases their computational load and can become bloated and inefficient [2]. In this context, the concept of a database server makes it unnecessary for robots to have strong computing, decision making, sorting and filtering information, and data storage capabilities, rather than only to connect to relevant offboard servers and obtain required services. The database server can be considered as a paradigm of system integration, data storage, and acquisition regarding robots.

This paper explains the design of a real-time database server platform that can provide the local robots with various functional services to perform their desired functions. The platform consists of a set of services within the microservices architecture, which integrates and synchronises several sub-systems in a multi-robot system. It is called a real-time database because it uses real-time processing to handle the workloads which constantly changes throughout the runtime.

The main contributions of this paper are as follows:

1. The architecture developed in this paper provides a place to integrate and synchronise different robots running autonomously with different operations.
2. Usually, the robots handle the data management and decision-making factors, which makes the computational capacity of the robots reduce over time. In this paper, the database server is designed to handle them such that the robots are not overloaded with processes.
3. Due to the flexibility of the architecture, the database server is designed such that the warden robots do not need to depend on the explorer robot for important information. Thus, the operation of robots continues even if the explorer robot shuts down.
4. The system is designed such that even if the database server is shut down, the robots use the lightweight database to store and perform the hazard and map services. This makes the system robust by continuing the operations even if the core of the system fails.

The rest of the paper is outlined as follows. Section 2 explains the literature review of the project. Section 3 describes the aim and objectives. The background and architecture designs are presented in Sections 4 and 5. Section 6 describes the software required to perform this project. The implementation and test results are described in Section 7. Discussions are explained in Section 8. Finally, Section 9 explains the conclusion and recommendations of this project. The real-time experimental data are tabulated in the Appendix A and the sequential diagram notations are described in the Appendix B.

**2. Literature review**

Cloud computing, also known as cloud computing technology, is one of the most active areas of Info-Communication Technologies (ICT) [3]. Cloud computing is a form of internet-based computing in which users can access, create and customise different application, hardware, and data over the internet using a web browser. It also highlights that the combination of cloud computing and robotics is an inevitable trend. Arumugam et al. [4] proposed Davinci based on Hadoop and ROS, which demonstrates the benefits of cloud computing for service robots in vast environments in scalability and parallelism. Du Z et al. [5] presented an architecture design of ‘robot cloud’ which combines the power of robotics and cloud computing. They use SOA to expand the capabilities and decision-making factors of physical robots. Nan Tian et al. [6] described Robotics and Automation as a Service (RAaaS) prototype where the robots access a remote server which hosts robust planning system. The above research mainly focuses on practical application areas but do not present systematic architectures for cloud robotics which is the focus of the paper.

Furthermore, the European Union (EU) has also launched ‘RoboEarth’, a ground-breaking cloud robotics initiative [7]. Then in 2014, Google, Microsoft and Stanford developed a new project named ‘RoboBrain’ [8] which allows robots to learn and share representations of knowledge. These developments suggest that cloud robotics can be used to extend the robots’ knowledge and skills effectively and efficiently. However, using distributed monolithic architecture, in which the services, data access code and the user interface are combined into a single program from a single platform. However, the monolithic architecture has inherent defects.

To solve the problem of deploying more applications, a lightweight architecture is proposed based on microservices. Microservices architecture consists of small autonomous services that work together to provide distributed services which fulfil user requirements. It is an architectural pattern emerged from Service-Oriented Architecture (SOA), emphasizing self-management and lightweight mechanisms [9]. Their independent deployability is advantageous for continuous delivery and can scale independently from other services. Also, they are easier to maintain and fault-tolerant since the failure of one service will not break the whole system [9]. Through microservices architecture (MSA), a system can be split into sub independent and loosely coupled services running as service instances. There are several differences between MSA and SOA. For example, the MSA uses simple messaging system for communication, whereas the SOA uses Enterprise Service Bus (ESB) for communication [10]. Another difference is that MSA focuses on service choreography where it interacts between multiple services, while SOA relies on both service orchestration and service choreography, represents control form one party’s perspective [10]. Butzin et al. [11] used microservices approach on the Internet of Things (IoT), building multiple applications from set of different services. In [12], server-client container was implemented, where server container contains different services and monitoring tools. The client container contains the monitoring agents and the facilities to forward the data to the server. Logstash was used as data processing pipeline which allowed them to collect data and send to the desired location. These works explore the microservices architecture in cloud robotics and emphasise its potential benefits. However, they don’t concern about the network availability in remote areas which results in high latency, cybersecurity threats due to the dependant of third-party services, and data privacy issues.

Choosing an effective data management system is a challenge. The authors of [13] examined at which database would be optimal for storing sensor data. They claimed that using Not only SQL (NoSQL) databases is more beneficial than traditional relational databases (RDBs) because it is specifically designed for unstructured data (document-oriented), so performing read or write operation on a single data entity is faster. Couchbase and MongoDB are the most popular document-based databases because it can handle large amounts of data in an agile manner [14]. According to [15] comparative study, MongoDB provided lower execution times than other databases, which is important when an application provides support for around thousand queries simultaneously. They recommended MongoDB instead of MySQL where the application will be storing large amount of data and queries lots of data.

ROS is a framework for the development of robotic applications. For commonly used functionalities, it provides services which deal with hardware abstraction, system control, message transfer between processes and package management. Adopting the standardized ROS software package model, reuse in robotic software development can be significantly prompted [16]. Another feature of ROS is its message transfer publish-subscribe model. ROS is originally designed to operate in a single robot or local area network [16]. However, the several existing ROS packages are made to perform on a single robot and do not contain the ability to provide on-demand services to multiple clients. Moreover, using the publish-subscribe model, large amount of bandwidth will be considered during constant message transfer with larger intervals. The platform proposed in this paper are aimed to address these challenges.

**3. Aim and objectives**

The paper aims to develop a real-time database server in the ROS platform by following microservices architecture. The objectives are as follows,

1. Develop microservices architecture to accommodate different services using the ROS platform.
2. Deploy a centralised database to store the required information and able to communicate with the server.
3. Deploy a personal database to prevent data loss during communication loss.
4. Synchronise data transfer between multiple robots.
5. Develop system architecture for Multi-Robot System (MRS) integration.

**4. Background**

This section mainly introduces ROS and the concepts of microservices and relevance between them.

4.1 Robot Operating System

ROS is an open-source robot software framework which includes a set of tools, libraries, and protocols that enable developers to concentrate on the development of robot software rather than the complexity and diversity of robot platforms. ROS offers a message request/response model to implement the inter-process communication and message transfer. *Nodes* are processes that perform computation in ROS. They are combined into a graph and communicate each other using ROS topics, services and actions. The response/request model of ROS relies on *ros\_service* for message transfer. Services are one of the ways to pass data between nodes. It is just synchronous remote procedure calls; they allow one node to call a function that executes in another node. Services are more efficient than topics as they are used where the data transfer is discontinuous.

A brief communication is as follows, In the server, a new service with unique name is created with a service type. In the client side, it waits for the service with that specific unique name to be available. Then a handle is created and requests are sent to the server through the handle. Initially, the input and output calls are defined in a *service-definition* file. Secondly, one node will send a request to the other node. The request will contain a message. Then the server will process the request and send a response message.

4.2 Microservices

Microservices is a software development architecture in which a single application is made up of many loosely coupled and independently deployable smaller services. These microservices adopt a lightweight communication protocol to interact with each other and provides more flexibility in software development. One of the most important processes in a microservices architecture is the service registry [17]. The client services request for each service instance should be registered with the registry as soon as it starts up so that the client can make the appropriate service request. The ros\_service package in ROS registers the registry with the service name as soon as it starts up. The client node will be waiting for the registry to be started in the other end. A brief mechanism about the registry is described in Fig. 1.

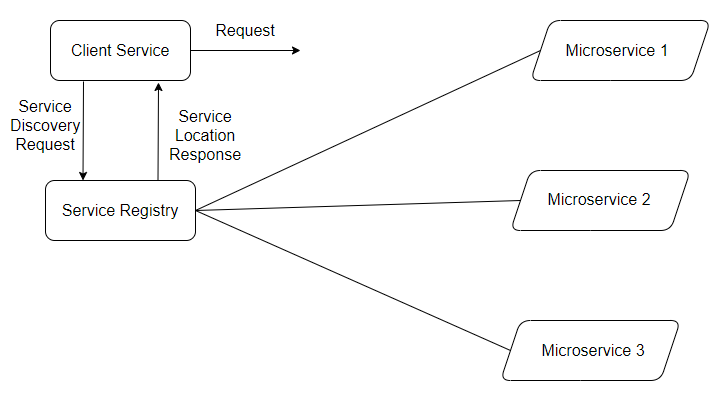


Fig. 1. Microservices architecture [18].

**5. Architecture Design**

In this section, the whole system architecture and the methods used to integrate the services are explained in detail. The basic architecture design of the platform is shown in Fig. 2.

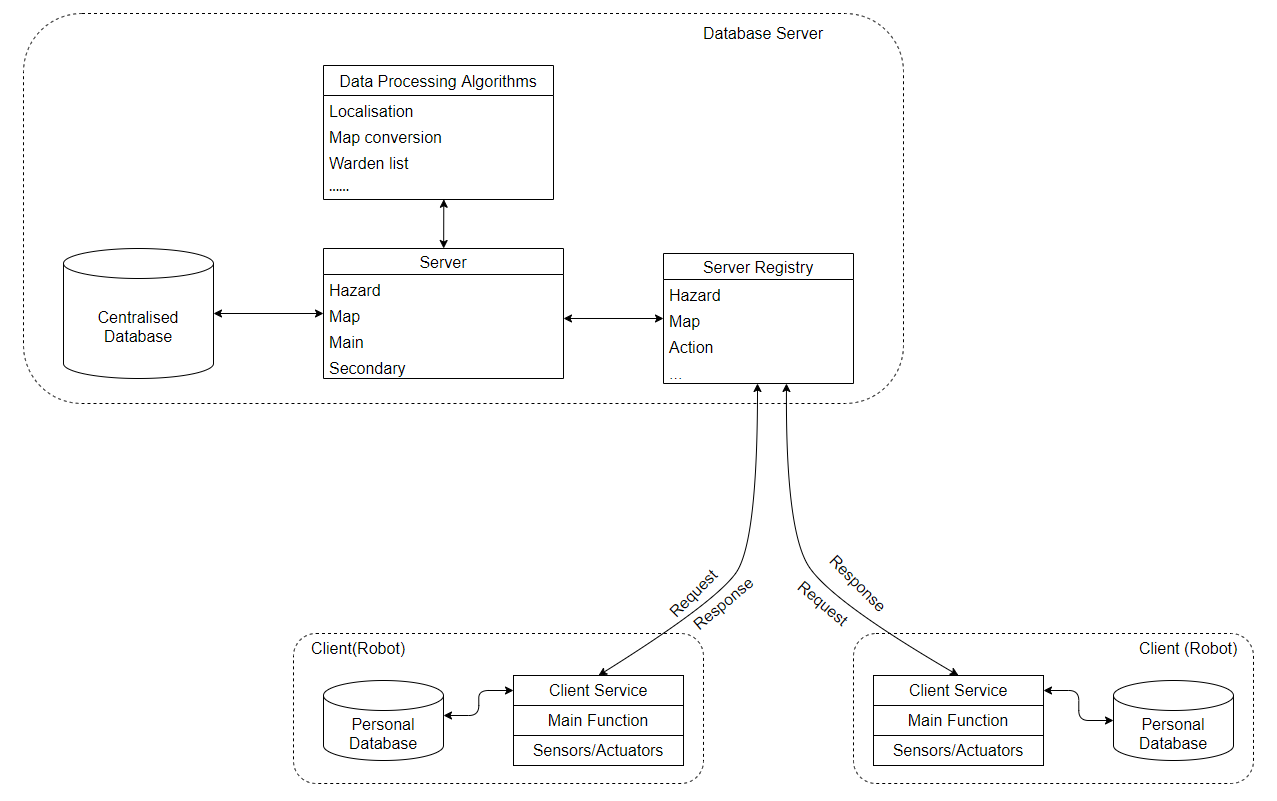


Fig. 2. Basic system architecture.

5.2 Database server

The database server is typically a dedicated computer that hosts the database. A database server is developed for communication between the real-time database and the robots (clients). It uses ros\_service that provides various services to communicate with other nodes. Importantly, the server integrates different services and synchronise the operation to run flawlessly. This database server relies exclusively on the client-server model for database access. In the client-server computing model, the server hosts all the services and the client connects to the server for message transfer. Typically, the client sends a request to the server for specific data. The server is programmed such that it knows what the client is asking for and takes the information from the database. Then it responds to the specific information to the client. Fig. 3 shows the sequence diagram of a client-server model.

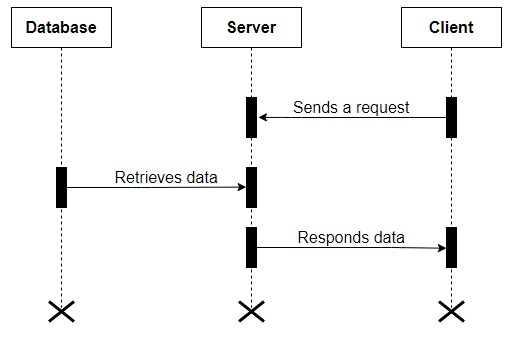


Fig. 3. Sequence diagram of a client-server model.

*Remark 1. The notations and a brief about sequence diagrams are explained in Appendix B.*

In this project, the services were categorised depending on the scenarios. They are classified as follows:

1. *Main sub-server.* The main sub-server is responsible for the following operations, real-time event services, beacon ID service, beacon info service, beacon goal service, localisation state services, and warden list services.
2. *Hazard sub-server.* The hazard sub-server is responsible for handling hazards detected by the robot.
3. *Map sub-server.* The map sub-server is responsible for handling map Portable Gray Map (PGM) files and YAML Ain’t Markup Language (YAML)files created by the robot.
4. *Secondary sub-server.* The secondary sub-server is responsible for monitoring the robot hardware information such as battery percentage, robot waypoints, and data processing speed. It directly subscribes to the robot’s published information, thereby not disturbing the main servers.

The reason for deploying different sub-services is because the hazard and map servers need to constantly iterate for certain data like tracking the localisation state of the robots and publish notification of new hazards for the robots.

           The explorer robot is equipped with a hazard and map server and it will act as a server with crucial functionalities when the main, hazard, and map server goes off due to some technical problems. The data will be stored in a personal database (which will be explained in the following section) and simultaneously it will monitor the server for its status. Once all the servers are up and running, the robot will send all the stored information to the server, thereby saving all the information even during technical difficulties. The warden and guide robots which are clients, use the service registry to connect to the relevant services to update or retrieve information from a centralised database. The main script of the robots does not have the functionality to connect to the services. So, a dedicated script for the client with a service registry is implemented which talks to the main script for the data transfer to the database server. A detailed architecture is shown in Fig. 4. For example, if the main function wants to send a new detected hazard to the server, it will send the data to client services and continue its operation. These client services lookup for the corresponding services and communicate with the database server, thereby updating the new hazard to the server.

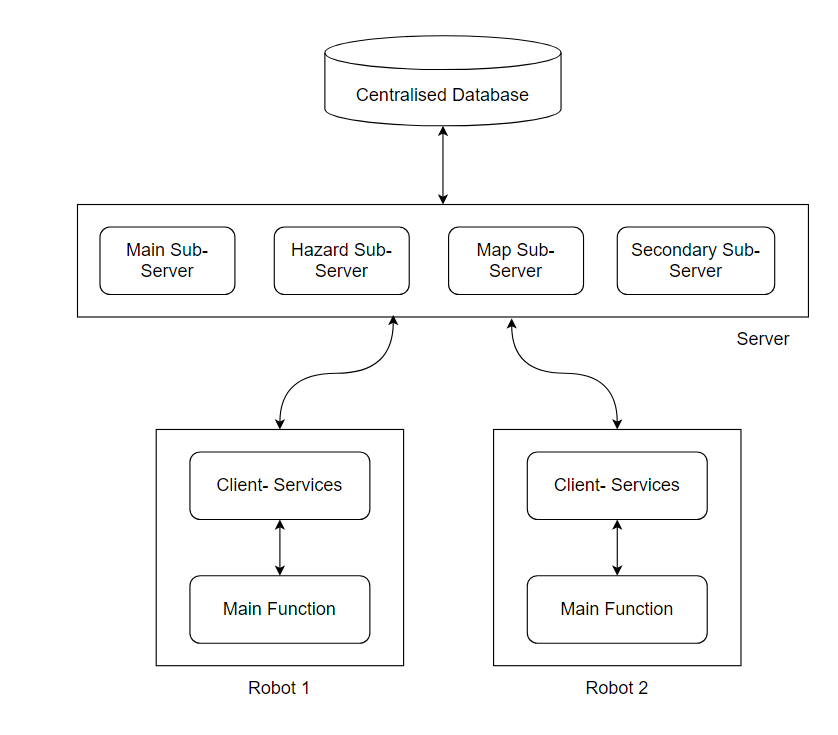


Fig. 4. Synchronisation of client-services and main function

5.1 Centralised database

The centralised database stores the core information of the project such as hazards, map data, beacon information, roles of the robots, and crucial events that occurred during the project in real-time. The database is located on the computer and interconnected with the server. We use a database mainly to store vital information, retrieve data if any communication is lost between robots, enable collaboration of many services, and post-mission analysis. There are four operations performed by the server for data manipulation, insert and retrieve data, compare and update for creating document ID, and event listener for monitoring changed occurring in database. Fig. 5 shows the operations performed by the server.

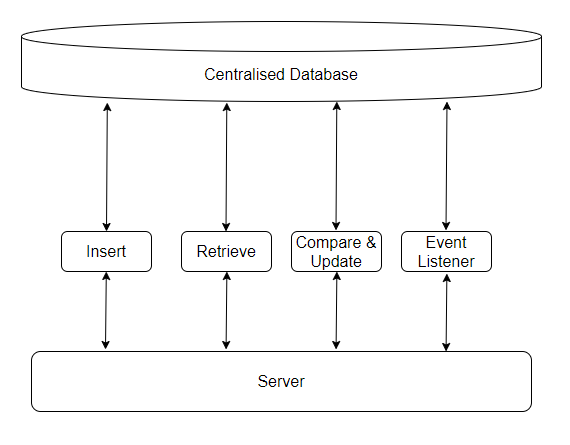


Fig. 5. Operations performed by the server for querying data from database

MongoDB, a document-based database is used because of its document data model which is a powerful way to store and retrieve data. Its horizontal, scale-out architecture can support huge volumes of both data and traffic [20]. It uses JavaScript Object Notation (JSON) formatting language rather than storing data in tables of rows and columns unlike Structured Query Language (SQL) databases [21]. A comparison of both MySQL, an open-source relational database management system, and MongoDB formatting style is shown in Fig. 6.

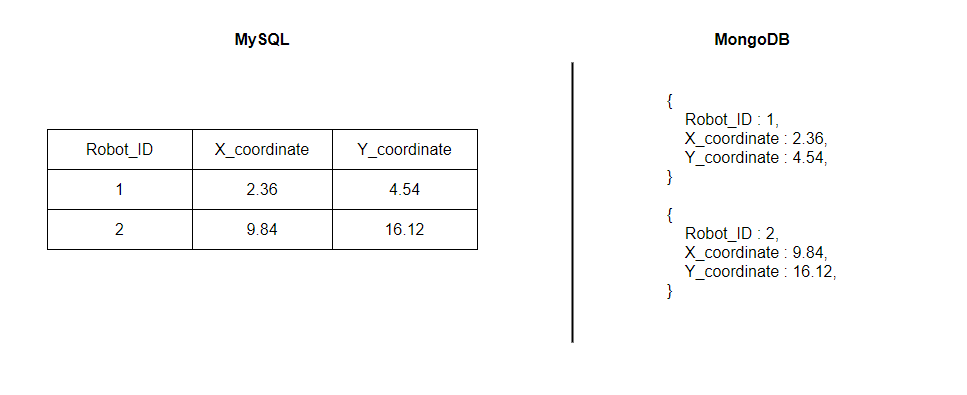


Fig. 6. Comparison of MySQL and MongoDB

.

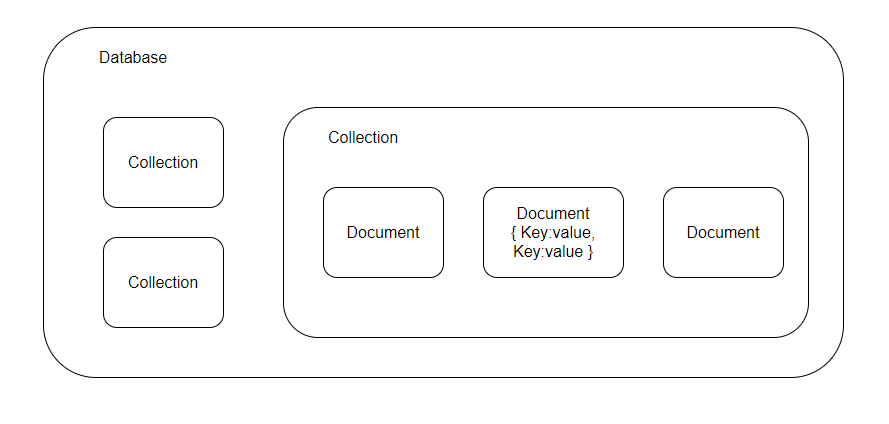


Fig. 7. Structure of MongoDB database

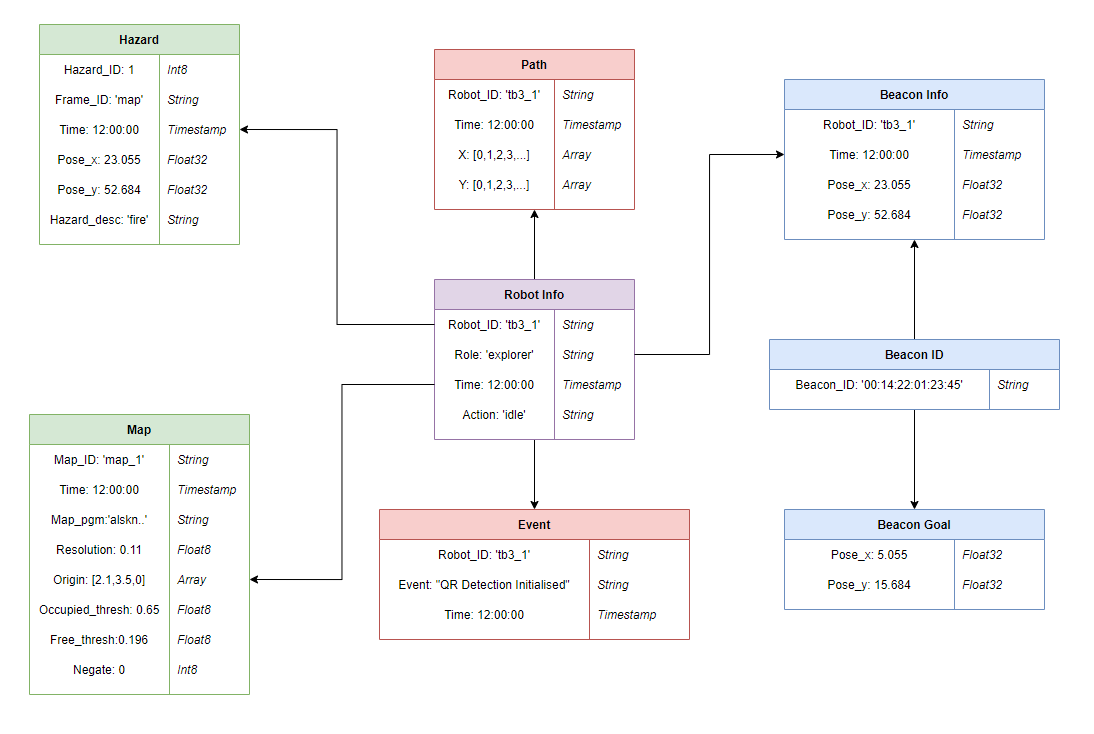


Fig. 8. Centralised database schema

*Remark 2: Fig. 8 follows the MongoDB schema which is a JSON type document schema that allows to define the shape and content of the documents and embedded documents in a collection.*

The structure of MongoDB is classified as follows, unlike Relational Database Management System (RDBMS), there is no table, instead, there are collections. The collections collect one or more documents. The structure of MongoDB is shown in Fig. 7. The database for this project consists of more than eight collections. These collections are mainly used to store the real-time information captured by the robots and monitoring subsystem. Fig. 8 shows the centralised database schema used in this project.

5.3 Personal database

During exploration, the explorer sends a newly identified hazard to the server, there can be chances of the database server switching off due to any technical issues. The data will be lost if the client services are unable to communicate with the server. To solve this problem, a lightweight database is used to store the information when the server is down. The reason for using a lightweight database is due to the limited low capacity of the robots. Fig. 9. shows an activity diagram of mechanism of the personal database in robots. Three functions are running simultaneously in the background, database server monitoring, map updating, and hazard updating. The database server monitoring function constantly monitors the status of the database server. Whenever it turns off, the detected hazard is stored in the personal database. For the personal database, JSON structure database is used, where a ‘. json’ file is created and the data is stored in JSON format. TinyDB is a lightweight document-oriented database that is written in Python and has no external dependencies [21]. For the create, read, update and delete (CRUD) operations, the TinyDB query language is used. The hazard data insert statement looks like this:

Get = Query()

check\_haz\_exists = db.get (Get.haz\_id.exists())

haz\_count = db.count (Get.haz\_id)

haz\_count = haz\_count + 1

db.insert ({'haz\_id': haz\_count, 'frame\_id': frame\_id, 'pose\_x': pose\_x, 'pose\_y':pose\_y, 'haz\_desc':desc })

The reason for using TinyDB is because of the similar structure of MongoDB, which will be suitable for synchronising the data without any third-party libraries.

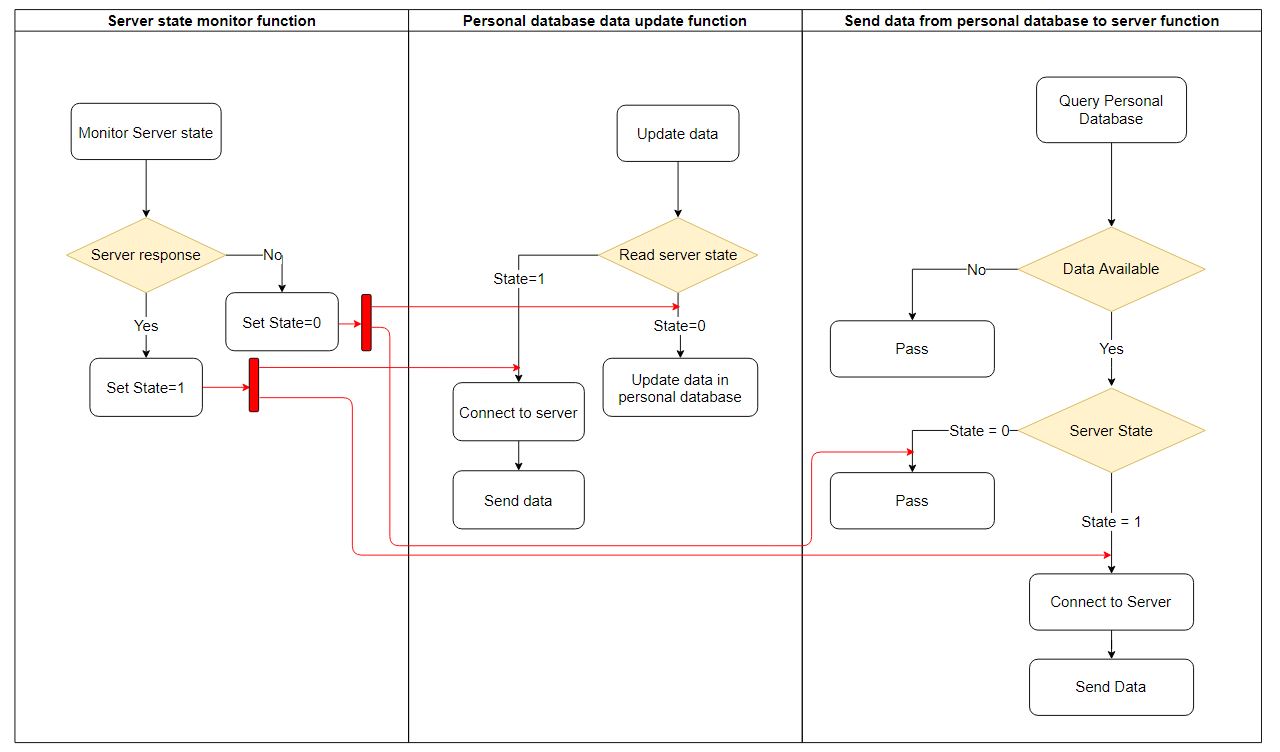


Fig. 9. Mechanism of personal database

*Remark 3: The red lines in Fig. 9 indicates that the ‘State’ value is dependent on server state monitor function.*

5.4 System integration

The database server plays an important role in system integration. In this project, several operations are running autonomously, to bridge them together and synchronise the functions between them is missing. The database server solves this problem by connecting all the operations through unique services and bridges the gap of synchronisation. The robots can function as an individual process like going to a specific goal, and identifying QR codes, but they cannot monitor the data flow and listen to the response messages by the other robots. The database server helps in synchronising the robots by collecting the data and storing it in the database, notifying other robots of new information, and responds the data when the robot requests the notified information. Thereby synchronising the two robots to a specific process. The server bounds multiple robots together by connecting them through services and synchronising them together for a specific process. Thus, the database server is required for system integration as the robots are set to perform a single function, it cannot handle the data management, data synchronisation, and decision-making algorithms simultaneously. The structure of system integration is shown in Fig. 10.

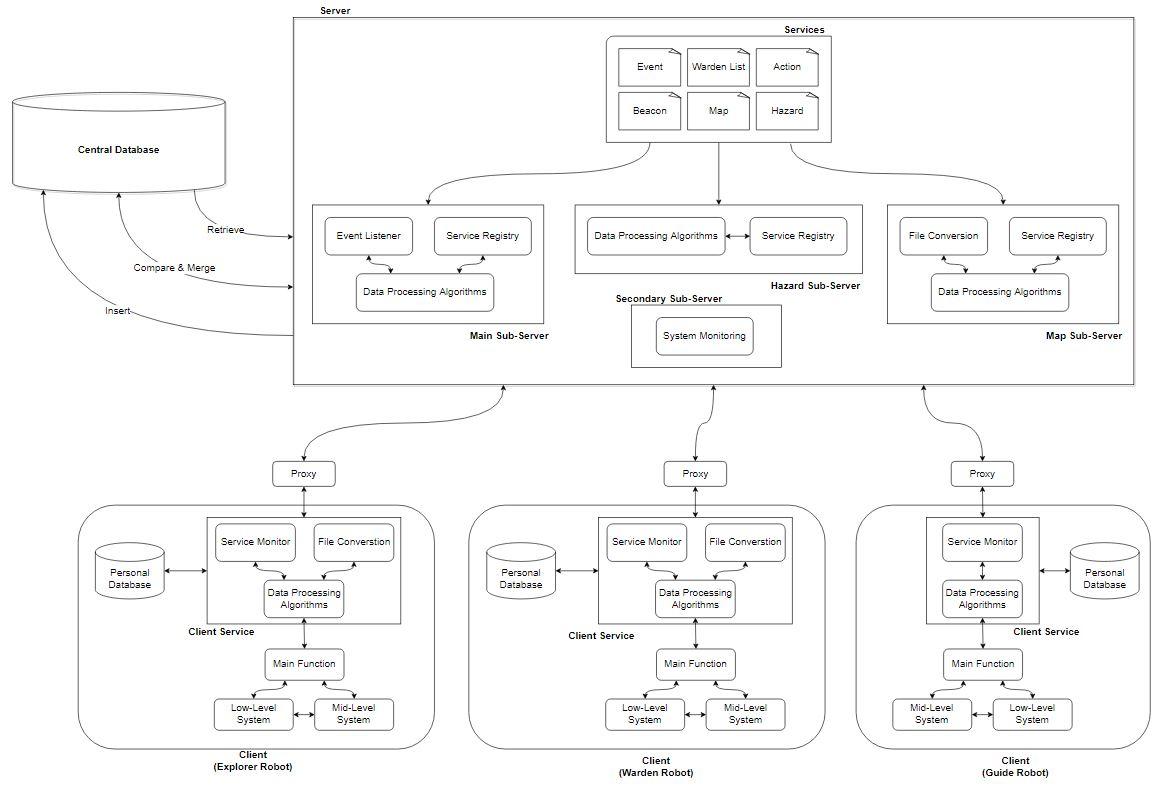


Fig. 10. Overall integrated system architecture of database server

**6. Software requirement**

6.1 Robot Operating System

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms. ROS can be operated in several operating systems. Debian, Ubuntu LTS, and Linux Mint are commonly recommended for ROS. Ubuntu 16.04 Xenial Xerus (LTS) operating system is used in our project. The proxy, service registry, and services from the ROS platform are used in this project. Python version 2.7 is used as the programming language in this project.

6.2 MongoDB

MongoDB is an open-source document database that uses JSON formatting language to store the information. MongoDB 4.4.6 is used as the centralised database for this project, and MongoDB Compass 1.26.1 is used as the GUI for the database to view and export the data for user analysis.

6.3 TinyDB

TinyDB follows the similar structure of MongoDB. TinyDB v4.4.0 is used as the personal database query language for the CRUD operations.

**7. Implementation and experimental results of database server in ROS platform**

Microservices architecture was designed as a solution for high availability and high-performance requirements in large-scale business situations. We don't have the conditions to simulate such a scenario right now, so we'll just focus on the feasibility of developing a cloud robotic application platform using a microservices architecture. In the previous sections, we have described the methodology of a different system integrating using microservice architecture. In this section, the implementation of services in different scenarios will be explained. Also, this section explains the data transfer between different processes using ros\_service.

7.1 Client-server model

The client-server model is used in our project to request and to respond data. The server contains different services with its corresponding call-back function, where its operation is programmed. The input and output data types are assigned. The procedure for data request is as follows:

1. The client services use the service registry to find the required services.
2. It sends the request using the corresponding service file where the data type is assigned.
3. The request is sent through the proxy to the server. Then the server picks the required information from the database and responds to the client through the proxy.

7.1 Map transfer

The explorer explores the unknown environment and creates a map constantly. As discussed earlier, there are two files created by the robot, a PGM file, a greyscale map and the other is a YAML file, the information of the grey map is stored. Since the explorer cannot send the files to the warden, the database server helps in transferring the map information to other robots. The real-time experimental data is shown in Appendix A. 1. Fig. 11 shows the procedure for map data transfer between robots. The procedure is as follows:

1. The explorer creates a map and the explorer client service takes the PGM file and converts it to string format because a file cannot be transferred wirelessly without converting it into a string.
2. Then the explorer client service reads the YAML file and exports the information inside it.
3. Then the new PGM and YAML file is sent to the server.
4. The map server gets the new information from the explorer and publishes a notification for other robots.
5. On the other hand, the warden robot constantly reads for notification, sends a request to the server whenever a new map notification is published.
6. The server reads the request and sends the corresponding PGM and YAML information to the explorer client service.
7. Then the string data is converted to PGM and YAML files using *base64* and *pyyaml* Python libraries.

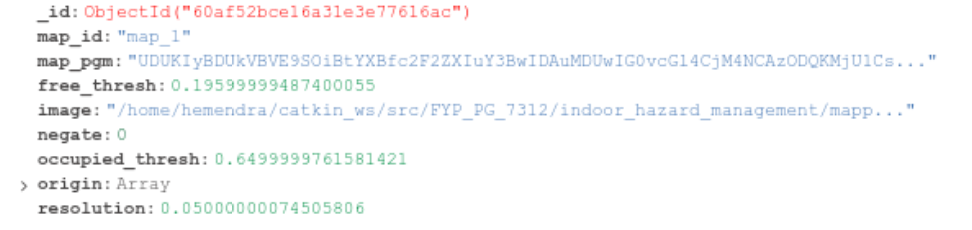


Fig. 12. Map information in database.

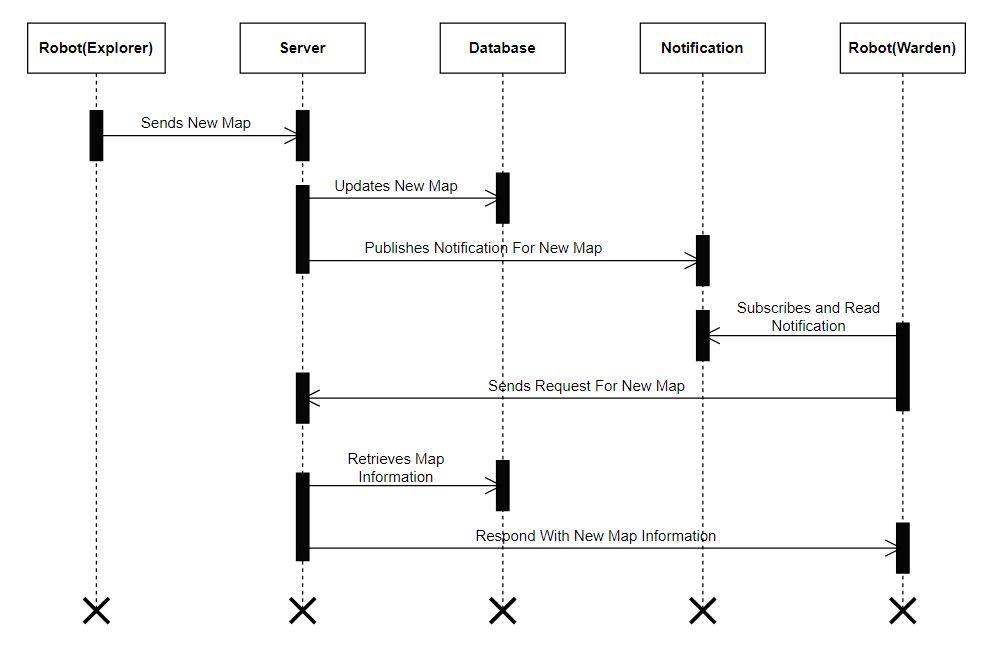


Fig. 11. Sequence diagram of map data transfer between explorer and warden robot

7.2 Hazard transfer

The transfer of hazard information between the explorer robot and warden robot is exactly similar to the map transfer except for the data conversion. Unlike map information, the number of hazards is not fixed, so the hazard server should assign a unique ID for the identified hazard such that it will be able to pick the required hazard. When the hazard client service requests for the hazard information, the hazard server calculates the number of stored hazards in the database and the highest number is the recent hazard. It retrieves the recent hazard and responds to the request with the new hazard information. The real-time experimental data is shown in Appendix A. 2. Fig. 13. shows the sequence diagram of hazard data transfer between explorer and warden robot and Fig. 14 shows the hazard information in the databases.

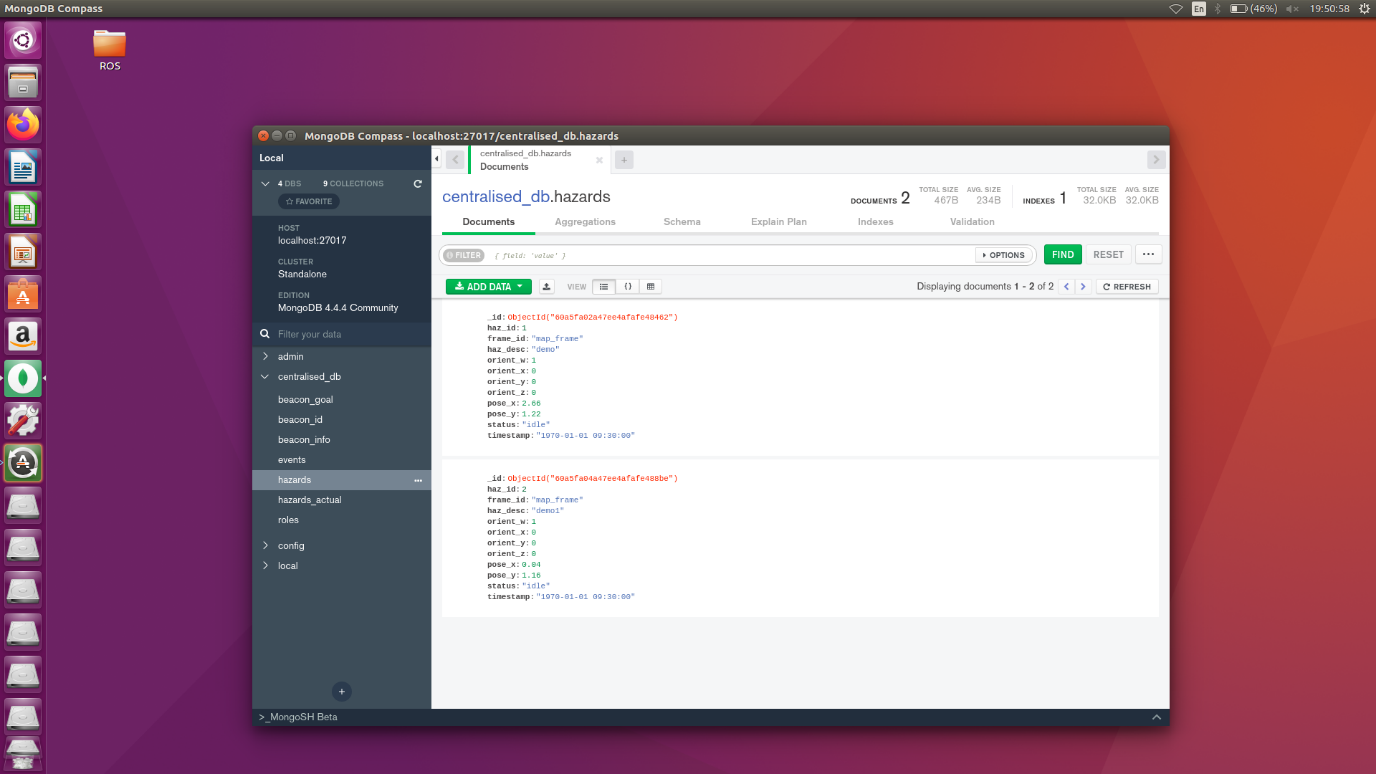


Fig. 14. Hazard information in database.

Generally, the hazard information is stored in the database and sent to the warden robots in the First in, First out (FIFO) method. Whenever the warden robots reach the desired hazard location, it updates the warden client service, and eventually, the client service sends the updates to the hazard server. The reason for updating this information is for the hazard server to know which identified hazards are occupied. This occupied action algorithm is used in certain scenarios where the warden robots are in idle mode while localising, during this period the robots do not listen to the notification and there are high possibilities that they will miss the hazard notification. This problem is explained in detail in the following section.

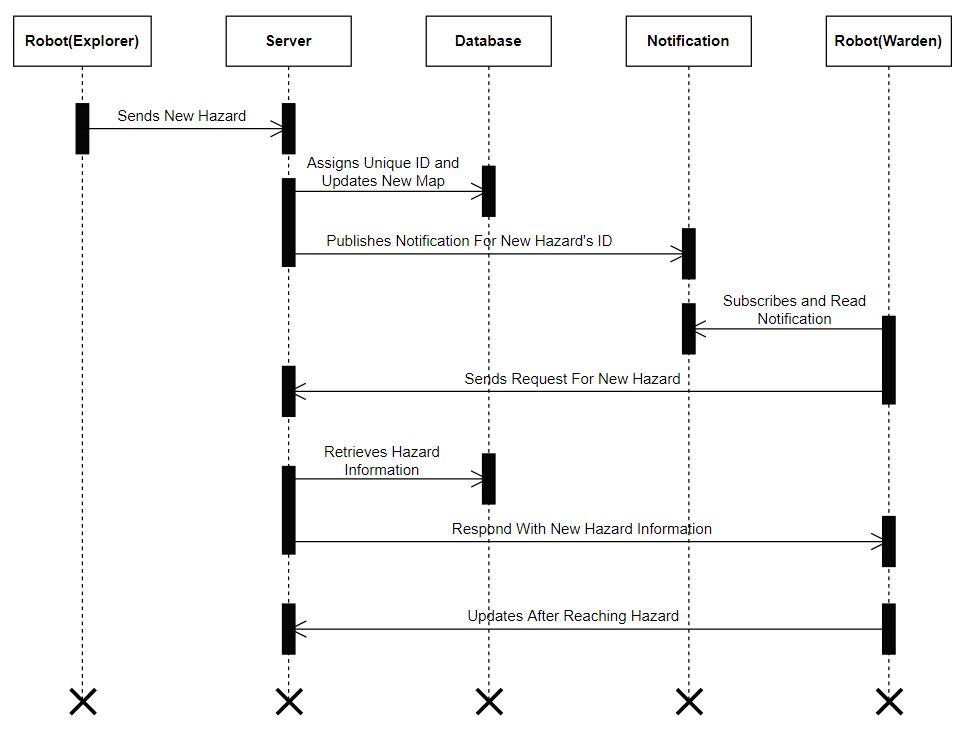
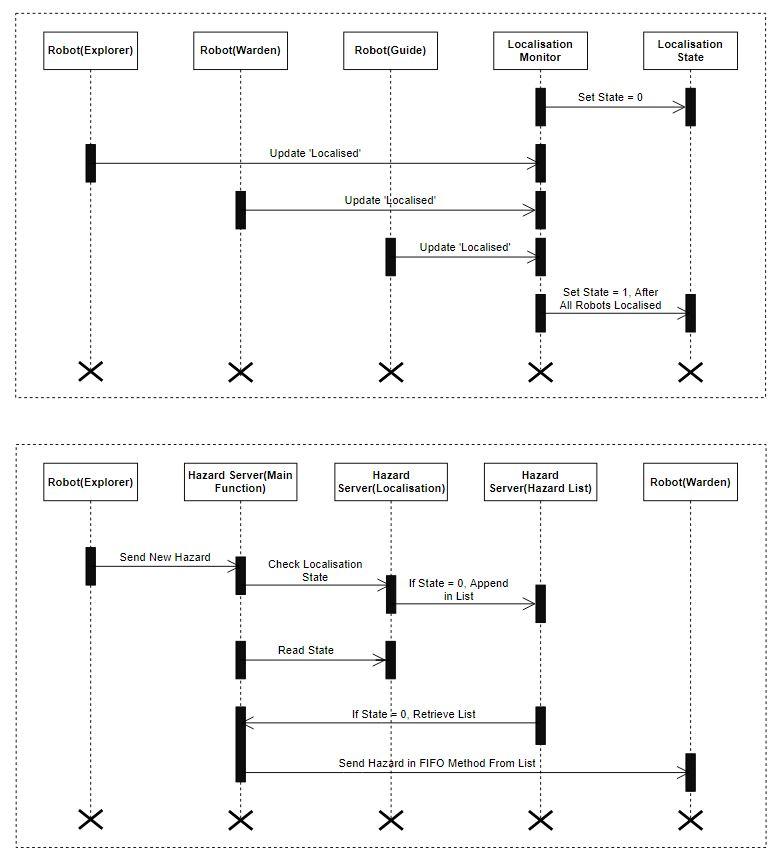


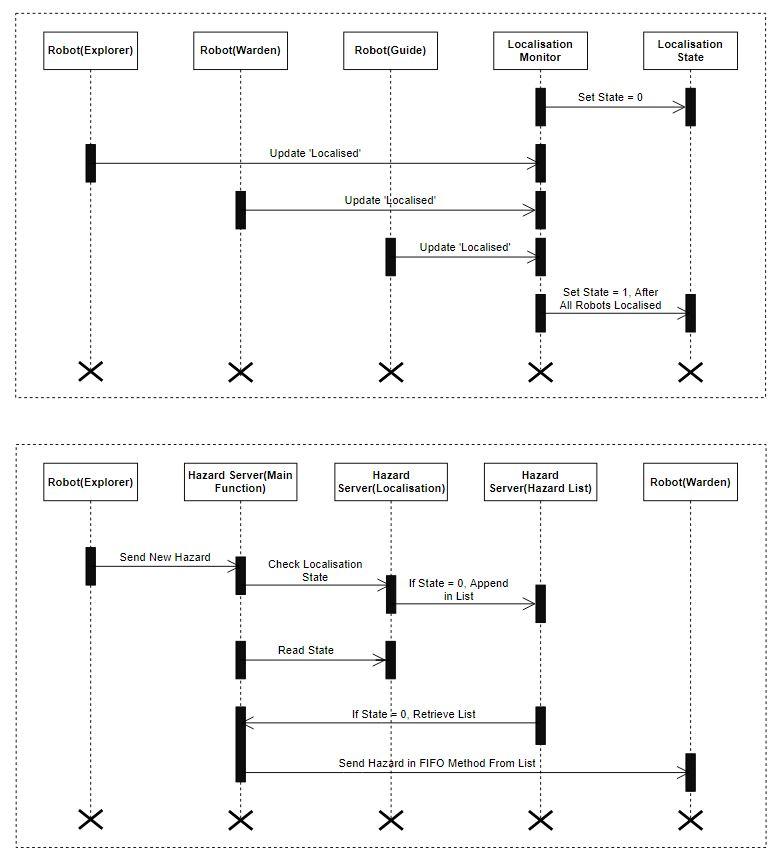
Fig. 13. Sequence diagram of hazard data transfer

7.3 Localisation state

Initially, all the robots localise to get position them in the map frame. During this period the robots do not perform any functions. The explorer robot starts to explore the floor and if it detects any hazard, it sends it to the server and it publishes a notification. Since the warden robot is currently localising it will ignore the notification and the detected hazard might go in vain. This problem yet again solved using a database server. Functions are running simultaneously in the hazard server; one is to check whether the warden robots have been localised and the other is the hazard manipulating function. If the explorer client service sends new hazard information to the hazard server, it checks whether the robots are localised. If not, it will append in a list and hold it without losing the information. Then after the robots are localised, it will send the hazard information in the FIFO method. If the robots are localised, then the standard hazard function which was explained above is executed. The real-time experimental data is shown in Appendix A. 3. Fig. 15 shows a sequence diagram explaining this process in detail.



1. Localisation monitor – Checks whether all robots have localised



1. After localised – Hazard information sent in FIFO method

Fig. 15. Multi-function sequence diagram of localisation state

7.4 Personal database

There are possibilities that the server might go down if any technical issues occur. TO overcome this problem, the explorer is equipped with a hazard and map server such that the explorer will act as the main server and responds to the requests of other robots. The reason for choosing hazard and map server only is because these both are the very important information required for this project and mainly, the robots are capable to run only services due to their minimal processing power. The data sent by the client services are stored in the personal database, the structure of the data is stored similar to the centralised database. The synchronisation between them is discussed in the following subsection. The real-time experimental data is shown in Appendix A. 4. Fig. 16 shows a sequence diagram of a personal database scenario.

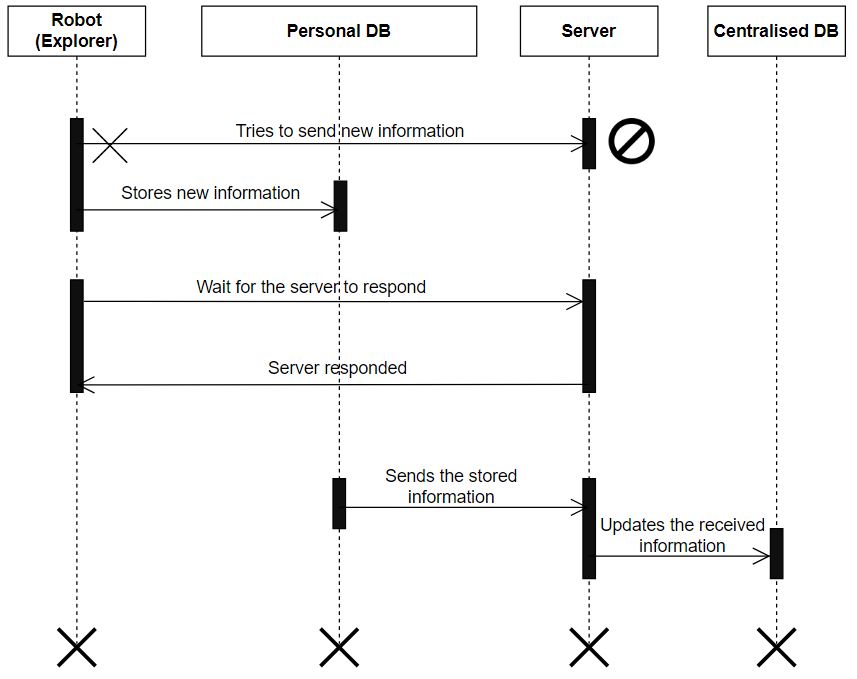


Fig. 16. Personal database scenario

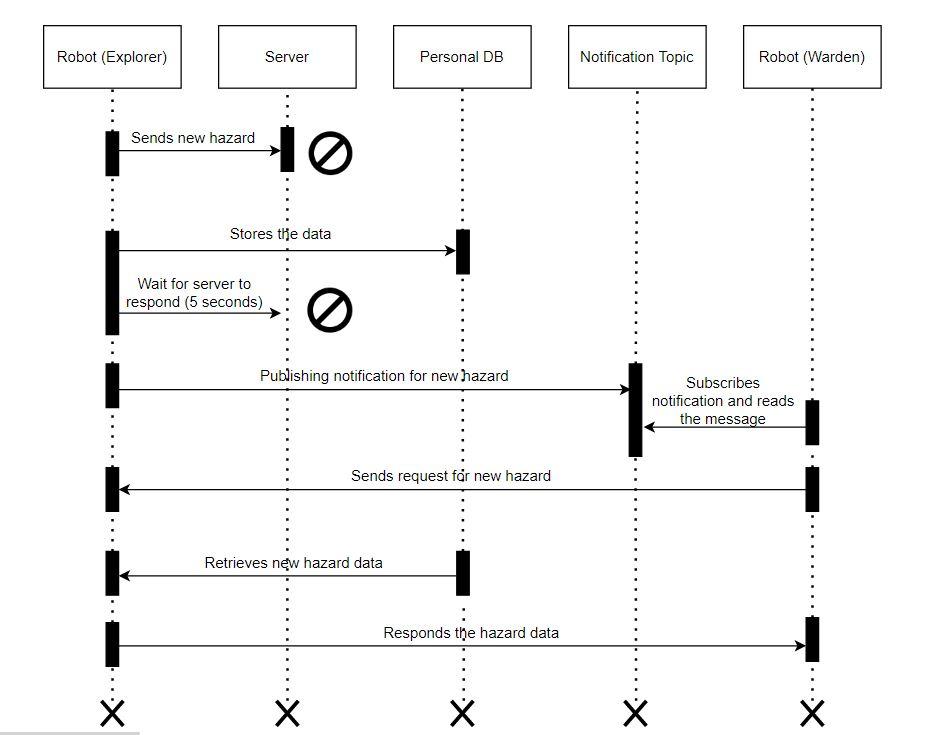


Fig. 17. Centralised and personal database synchronisation

7.5 Centralised and personal database synchronisation

When the database server is shut down, the explorer client services are going to execute the hazard server function and it will be storing the hazard information. If the server turns back on and starts running in the midway, it might not be aware of the past detected hazard stored in the personal database of the explorer robot. So, it is important to synchronise the data between the personal database and centralised database. For synchronising the data, we use both the database server and explorer client services. In the explorer client service, a function monitors the database server’s state all the time. When the server starts running back again, the explorer services detect that and send the data stored in the personal database to the main hazard server. Thereby both the databases synchronise when centralised database goes on and off. The real-time experimental data is shown in Appendix A. 5. Fig. 17 shows the working of centralised and personal database synchronisation.

**8. Discussion**

The system is scalable such that the number of robots can be increased, but the robot ID of each robot has to be mentioned in the database. Regarding the roles of the robots, currently, it is fixed in the robots but a methodology has been proposed and due to time constraints, it has not yet been implemented. So, a user should understand the system and implement the roles to the robots manually during execution. Moreover, only the important scenarios which are discussed above have been successfully tested and implemented but there are other scenarios like warden checking the hazard and updating the database that has not been considered. Due to reflections, the camera frequently detects false hazard information. Fortunately, the QR code filtration is done in the data processing section, where it reads the information and neglects if the hazard description is empty or has random numbers.

**9. Conclusion and recommendations**

In this paper, we have presented a new database server in the ROS platform using microservices architecture for hazardous environment applications. The proposed system architecture is demonstrated using sequence diagrams, schematic and structural diagrams. Moreover, the microservices architecture makes the interactive process between ROS service and other sub-systems more convenient. Without the database server, the robots might find a hard way to communicate and transfer large data, and also, there might be a halt in operation if the explorer robot fails to operate. The database server solves various crucial problems as discussed and also helps in storing and retrieving information, integrate several systems, and helps in analysing the post-mission for research purposes. There are recommendations regarding scalability to the proposed architecture, which are as follows:

1) Currently, the inputs are given directly in the database, which might be complicated if any user without knowledge of it to handle. So, a custom Graphical User Interface (GUI) can be designed and implemented.

2) The data management and decision-making factor are currently performed in the server. It can be upgraded by running all the computation in the server thereby increasing the performance of the robots.

**10. Acknowledgements**

This paper and the study behind it would not have been possible without the exceptional help from Professor Cheng-Chew Lim and Co-Supervisor Dr. Syed Imranul Islam. From my first experience with the project's introduction to the final paper, their passion, expertise, and exact attention to detail were an inspiration and kept my work on track. Members of my project team have also helped me with my goals and guided me in the right direction. I am also grateful for the informative feedback provided at books & texts by the anonymous peer reviewers. This research has been enriched by the generosity and expertise of one and all in countless ways and rescued me from many mistakes; those that undoubtedly remain are solely my fault.

**References**

[1] A. Gautam and S. Mohan, "A review of research in multi-robot systems," 2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS), 2012, pp. 1-5, doi: 10.1109/ICIInfS.2012.6304778.

[2] James Kuffner. 2010. Cloud-enabled Humanoid Robots. In *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*. Nashville TN

[3] P. Mell, T. Grance, The NIST definition of cloud computing, Commun. ACM53(6) (2011)1–7.

[4] R. Arumugam et al., "DAvinCi: A cloud computing framework for service robots," 2010 IEEE International Conference on Robotics and Automation, 2010, pp. 3084-3089, doi: 10.1109/ROBOT.2010.5509469.

[5] Z. Du, L. He, Y. Chen, Y. Xiao, P. Gao and T. Wang, "Robot Cloud: Bridging the power of robotics and cloud computing", *Future Generation Computer Systems*, vol. 74, pp. 337-348, 2017. Available: 10.1016/j.future.2016.01.002.

[6] N. Tian et al., "A cloud robot system using the dexterity network and berkeley robotics and automation as a service (Brass)," 2017 IEEE International Conference on Robotics and Automation (ICRA), 2017, pp. 1615-1622, doi: 10.1109/ICRA.2017.7989192.

[7] D. Hunziker, M. Gajamohan, M. Waibel and R. D'Andrea, "Rapyuta: The RoboEarth Cloud Engine," 2013 IEEE International Conference on Robotics and Automation, 2013, pp. 438-444, doi: 10.1109/ICRA.2013.6630612.

[8] M. Waibel et al., "RoboEarth," in IEEE Robotics & Automation Magazine, vol. 18, no. 2, pp. 69-82, June 2011, doi: 10.1109/MRA.2011.941632.

[9] D. Taibi, V. Lenarduzzi and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study", Proceedings of the 8th International Conference on Cloud Computing and Services Science, 2018. Available: 10.5220/0006798302210232.

[10] P. Di Francesco, P. Lago and I. Malavolta, "Architecting with microservices: A systematic mapping study", *Journal of Systems and Software*, vol. 150, pp. 77-97, 2019. Available: 10.1016/j.jss.2019.01.001.

[11] B. Butzin, F. Golatowski and D. Timmermann, "Microservices approach for the internet of things," 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), 2016, pp. 1-6, doi: 10.1109/ETFA.2016.7733707.

[12] A. Balalaie, A. Heydarnoori and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," in IEEE Software, vol. 33, no. 3, pp. 42-52, May-June 2016, doi: 10.1109/MS.2016.64.

[13] J. S. van der Veen, B. van der Waaij and R. J. Meijer, "Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual," 2012 IEEE Fifth International Conference on Cloud Computing, 2012, pp. 431-438, doi: 10.1109/CLOUD.2012.18.

[14] Jing Han, Haihong E, Guan Le and Jian Du, "Survey on NoSQL database," 2011 6th International Conference on Pervasive Computing and Applications, 2011, pp. 363-366, doi: 10.1109/ICPCA.2011.6106531.

[15] C. Győrödi, R. Győrödi, G. Pecherle and A. Olah, "A comparative study: MongoDB vs. MySQL," 2015 13th International Conference on Engineering of Modern Electric Systems (EMES), 2015, pp. 1-6, doi: 10.1109/EMES.2015.7158433.

[16] S. Wen, B. Ding, H. Wang, B. Hu, H. Liu and P. Shi, "Towards migrating resource-consuming robotic software packages to cloud," 2016 IEEE International Conference on Real-time Computing and Robotics (RCAR), 2016, pp. 283-288, doi: 10.1109/RCAR.2016.7784040.

[17] Chris Richardson and Floyd Smith. 2016. Microservices: From Design to Deployment. <https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/> [Accessed 31 May 2021]

[18] D. Bhamare, M. Samaka, A. Erbad, R. Jain and L. Gupta, "Exploring microservices for enhancing internet QoS", *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 11, p. e3445, 2018. Available: 10.1002/ett.3445.

[19] MongoDB. 2021. Why and When to Use MongoDB. [https://www.mongodb.com/why-use-mongodb](https://www.mongodb.com/why-use-mongodb#:~:text=Companies%20and%20development%20teams%20of,of%20both%20data%20and%20traffic) [Accessed 02 June 2021]

[20] MongoDB. 2021. Comparing the Differences – MongoDB vs MySQL. <https://www.mongodb.com/compare/mongodb-mysql> [Accessed 02 June 2021]

[21] Markus Siemens. 2015. tinydb. <https://github.com/msiemens/tinydb> [Accessed 01 June 2021]

**Appendix A – Experimental Data**

In this appendix, the results of the above presented sequence diagrams are tabulated.

1. Hazard transfer

Table 1 Hazard transfer from explorer to warden robot via database server

|  |  |  |
| --- | --- | --- |
| **Hazard information** | **Detected data by explorer robot** | **Received data by warden robot** |
| Pose X | 2.66 | 2.66 |
| Pose Y | 1.22 | 1.22 |
| Orientation X | 0 | 0 |
| Orientation Y | 0 | 0 |
| Orientation Z | 0 | 0 |
| Orientation W | 1 | 1 |
| Time | “2021-06-01 18:25:26.5645” | “2021-06-01 18:25:26.5645” |
| Hazard Description | “Fire” | “Fire” |

1. Map transfer

Table 2 Map transfer from explorer to warden robot via database server

|  |  |  |
| --- | --- | --- |
| **Map information** | **Data created by explorer robot** | **Data received by warden robot** |
| Map PGM file | Map info in string | Map info in string |
| Time | “2021-06-01 18:25:26.5645” | “2021-06-01 18:25:26.5645” |
| Resolution | 0.11 | 0.11 |
| Origin | [2.1,3.5,0] | [2.1,3.5,0] |
| Occupied thresh | 0.65 | 0.65 |
| Free thresh | 0.196 | 0.196 |
| Negate | 0 | 0 |

1. Localisation state

Case 1: Before warden robots localise

Table 3 Events occurred before warden robots localise

|  |  |
| --- | --- |
| **Events** | **Number of hazards** |
| Hazards detected by explorer robot | 2 |
| Database server holding the hazards | 2 |
| Expected hazard information to publish notification by server | 0 |
| Published notification for hazard information by server | 0 |

Case 2: One warden robot localised; one warden robot not localised

Table 4 Events occurred when one warden robots localise and other not localised

|  |  |
| --- | --- |
| **Events** | **Number of hazards** |
| Hazards detected by explorer | 2 |
| Database server holding the hazards | 2 |
| Expected hazard information to publish notification by server | 0 |
| Publish notification for hazard information by server | 0 |

Case 3: All warden robots localised

Table 5 Events occurred after all warden robots localised

|  |  |
| --- | --- |
| **Events** | **Number of hazards** |
| Hazards detected by explorer | 2 |
| Database server holding the hazards | 0 |
| Expected hazard information to publish notification by server | 2 (FIFO method) |
| Publish notification for hazard information by server | 2 |

1. 4. Personal database

Table 6 Personal database scenario

|  |  |  |
| --- | --- | --- |
| **Server state** | **Hazards detected by explorer robot** | **Hazards received by warden robot** |
| Server not responding | 1 | 1 |
| Server responding | 1 | 1 |

1. 5. Centralised database and personal database synchronisation

Table 7 Centralised database and personal database synchronisation

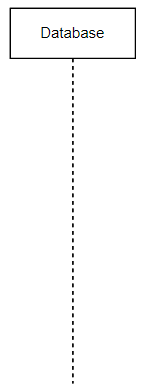
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Server state** | **Expected hazards in personal database** | **Hazards in personal database** | **Expected hazards in centralised database** | **Hazards in centralised database** |
| Server responding | 0 | 0 | 1 | 1 |
| Server not responding | 1 | 1 | 1 | 1 |
| Server responding after two minutes | 0 | 0 | 2 | 2 |

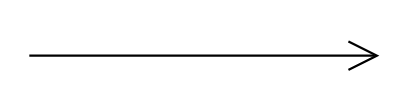
**Appendix B – Sequence Diagram Notations**

Sequence diagrams show how processes are carried out. They capture the interaction between objects in a collaborative context. Sequence diagrams are time-focused, and they visually describe the sequence of the interaction by using the vertical axis of the diagram to represent time, showing when messages are sent and when they are received. The horizontal axis shows the elements that are participated in the interaction. The objects involved in the operation are listed from left to right depending on when they take part in the message sequence. However, the elements can be in any order.

The vertical axis represents time proceedings down the page. The vertical space in an interaction diagram is not relevant for the duration of the interaction.

The notation for the sequence diagram is as follows,

 Lifeline - It is a named element which depicts an individual participant in a sequence diagram.

 Data transfer

 Single process

 Multiple processes

 End process

 Unable to reach destination

Element not responding