Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

b-it Bonn-Aachen International Center for Information Technology

Master's Thesis

A mediator system for querying heterogeneous data in robotic applications

Rubanraj Ravichandran

Submitted to Hochschule Bonn-Rhein-Sieg,
Department of Computer Science
in partial fullfilment of the requirements for the degree
of Master of Science in Autonomous Systems

Supervised by

Prof. Dr. Erwin Prassler
Prof. Dr. Manfred Kaul
Nico Huebel
Sebastian Blumenthal

April 2019

I, the undersigned below, declare that this work has not previously been submitted to this or any other university and that it is, unless otherwise stated, entirely my own work.

Date	Rubanraj Ravichandran

Abstract

In robotic applications, sensor-generated data is often ignored after robots utilize the data for making decisions and sometimes save into persistent storage. Since these data are not adequately modeled and stored, it makes it hard for someone who wants to replay the experiments or to find faults in the sensor data. It is even more difficult to debug this massive amount of data if there are multi-robots involved in a task. It is likely that different vendors/developers develop multi-robots with different database instances and attribute names to save data which introduces heterogeneity in the sensor data. Heterogeneity includes additional problems like interoperability issues when sharing data between other robots and also with fault diagnosis tools. One way to overcome these issues is by employing a mediator component as a middle man for all robots and even for humans. In our approach, we designed a mediator architecture which solves integrating sensor data from different databases which are deployed on different robots. Also, the data modeling issue from the EU ROPOD project's data logger system is addressed by creating an extendable generic data model for each critical entities in the robot system. Lastly, sensor observations interoperability issue is solved by adding meaningful contexts to all the entities, and it is achieved by using JSON-LD data representation. Overall mediator component is developed with GraphQL as a base framework and JSON-LD to represent the response data. This choice of GraphQL and JSON-LD provides further advantages to the system such as a single query language to fetch sensor data regardless of databases used in the robots and context-based data model.

Acknowledgements

I would like to express my gratitude to Prof. Dr. Erwin Prassler and Prof. Dr. Manfred Kaul for their valuable and constructive advice during the planning and development of this research work. I feel so grateful to Nico and Sebastian for taking time out of their busy schedule to answer so many questions. Without their help, I would not have been able to complete the project in such a proficient and timely manner. Also I would like to thank Santosh Thoduka from ROPOD team for sharing the necessary sensor data for experimentation purpose and answered my questions as quick as possible.

I wish to say special thanks to various members from Zendri Gmbh; Dave, Jens, Janek, Lars, and Nicolas for giving the opportunity to explore and have hands-on experience on novel technologies which are being used in this research work and supported me personally throughout my master's course.

I would like to extend my thanks to my mom, and girlfriend Bindya for proof-reading my report and motivating me to finish my thesis work on time.

Finally, I would like to show appreciation to all my friends Akhilesh, Ashutosh, Deepan, Kishaan, Livin, Mohandass, Naresh, Pardeep Naik, Pradeep Krishna, Rajat, Ramesh Kumar, Senthil, and Sindhu for being excellent support for me through my ups and downs, and for nostalgic moments.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID Atomicity, Consistency, Isolation, and Durability

API Application programming interface

CBOR Concise Binary Object Representation

FDBS Federated Database System

FMEA Failure Mode and Effects Analysis

HDT Header, Dictionary, and Triples

HTTP HyperText Transfer Protocol

IRI Internationalized Resource Identifier

JSON JavaScript Object Notation

JSON-LD JavaScript Object Notation for Linked Data

OGC Open Geospatial Consortium

RDF Resource Description Framework

REST Representational State Transfer

ROS Robot Operating System

SQL Structured Query Language

SenML Sensor Measurement Lists

UML Unified Modeling Language

URL Uniform Resource Locator

UUID Universally Unique Identifier

ZMQ Zero Message Queueing

1

Introduction

Robots generate a large amount of data from different types of sensors attached to it and also from its hardware components. Typically, these data come from different sources and requires different treatment. It can roughly be split into two categories:

- Streams of raw sensor data: This data is typically produced at high frequency and can be small (like the "tick" of an encoder) or large blob data (like a point cloud generated by a laser scanner).
- Meta-data: This is connecting different types of sensor data with the information and knowledge required to interpret them (like which robot has produced that data using which sensor with which settings at which position, applying which algorithm while performing what task). E.g., a point cloud of a laser scanner is not very useful if it is unknown from which position it was taken with which type of laser scanner and which settings.

In our previous research work [23], we have conducted an extensive qualitative and quantitative analysis to find better databases and architectures that effectively store these data and consume it for further operations. Results from our previous work show that a single database is not suitable for every robotic scenario. For example, in terms of handling large BLOB data, MongoDB stored them faster but reading the data was slower compared to CouchDB [23]. Also, to complete a given task robot depends on multiple sources of information from internal sensors, as well as external sources for example world model, kinematic model, etc..

Adoption of multiple databases for robotic applications requires a unique way of mediation to view multiple databases as a single federated database. The mediator approach helps to integrate data from different sources and produce an only result back to robots. Mediator abstracts the information of how data is being stored in various data sources from a robot and allows robotic applications stream data to mediator independent of databases used in the back-end.

To map the data generated by robots with multiple databases, the mediator system requires a proper data model predefined in the context of robotic applications. Modeling robot produced data helps to generalize the structure of data and defining relations between different entities (e.g., tasks, sensors, robots, location) in a robotic application scenario. If we have well defined robotic data models, then the mediator will get the ability to mutate or query data from different data sources. Also, it is essential that any robotic use-cases should be able to extend these data models.

As mentioned in these papers [2, 12, 3, 6, 6, 24], mediators are being used to integrate data from different data sources, and few architectures support single data model (e.g., SQL), and others recommend for different data models (e.g., SQL, NoSQL, document store, etc..). Also, they differ from query languages, ease of implementation, and components used in their architecture. This project mainly focuses on defining semantic models for sensor data to make it more interoperable with other systems or even in multi-robot systems, and implementing a mediator system which acts as a middle-ware between robots and databases.

1.1 Motivation

Streamlining the data produced from different sensors in robotic applications is a tedious task, and there are no specific standards to organize the data in terms of making relations between the entities and also giving context to the data. It will be even more complicated when we have a multi-robot platform and sharing data between them, and backing up the data into a database for fault diagnosis.

Currently, in the ROPOD[1] project, a single black box component has been designed with data loggers to run the robot test cases and store the sensor generated data in MongoDB. During the experimentation black box stores the data produced

[1]ROPOD is an EU funded project to develop "Ultra-flat, ultra-flexible, and cost-effective robotic pods for handling legacy in logistics"

by the sensors as dumps into a single MongoDB instance locally. The three main challenges of data storage and retrieval are,

The first problem is, considering the sensor data stored as a collection of logs without data modeling makes the consumer[2] inability to execute meaningful queries against the data.

The second problem is missing contexts and the entity-relationship model. For example, if a consumer tries to query the data from dumps, then the consumer doesn't get additional information in the result such as which sensor produced this data from which robot/black-box at which location and time, and which person created the test case. What we mean by "missing context" is if humans read the data they will understand what's the meaning of each parameter, but if a different robot/black-box tries to consume the data produced by other robots then it will fail. Therefore the context about the data should be shared somewhere globally.

Finally, sharing the data generated by multi-robots to solve a collaborative task and fetching data from multi-robots by a human controller to debug the test case.

These significant issues inspired us to find a suitable Entity-Relationship data model and unique mediation system to query heterogeneous sensor data from multiple data sources regardless of the database type.

1.2 Structure

- Section 2 concisely describes the background knowledge of the topics which are relevant to this work.
- Section 3 shows the related work which has been carried out earlier in the field of Federated databases and data modeling, and highlights the importance of declarative data fetching.
- Section 4 discusses problem formulation for this research work and section 5 gives detailed explanation of our approach and methodology for the mediator system.
- Section 6 includes the complete implementation details of the mediator system and section 7 explains the experiment procedure.

[2]A consumer can be either humans or machines.

- Section 8 evaluates the developed mediator component based on requirements and existing black box data logger system in ROPOD project, and section 9 shows the limitations and a road map for the future work.

2

Background

This section elaborately discusses the concepts which help the readers to understand the remaining parts of this research work.

2.1 Data model

"A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to properties of the real world entities. For instance, a data model may specify that the data element representing a car be composed of a number of other elements which, in turn, represent the color and size of the car and define its owner." [28]

In Database Management Systems context, a data model depicts the structure of data stored and obtained from the database. Different database systems follow different ways of how they store the data physically in the device, and the users may choose the database based on their application requirements. A good data model determines the overall performance of the application. In Database Management Systems context, data model depicts the structure of data stored and obtained from the database. Different database systems follow different ways of how they store the data physically in the device, and the users may choose the database based on their application requirements. A good data model determines the overall performance of the application.

2.1.1 Types of data models

There are many types of data models available. However, some important data models mentioned below are widely approved and utilized in various fields.

- *Relational data model* is one of the traditional data models which represents the attributes as column names and the actual data in the form of rows.
- *Graph data model* is the newcomer in the market and solves many problems in social networking applications. It stores the data as properties in nodes as well as in the edges that connect two different nodes.
- *Document data model* is the competition for Relational data models since this data model does not stress uses to provide a valid data schema. So, users can store data like documents or semi-structured data.
- *Column family data model* stores the data in individual column and columns that fall under the same category can be grouped as a column family.
- Few databases in the market support a mixture of these existing data models and those databases are called as *Multi-model* databases.

2.2 Database Schema

The database schema is a formal language used to define "the blueprint of how a database is constructed" [29]. Schema definition differs from database to database and also based on the data model that the database uses. For example, in a relational data model database, schema includes the table, column names, column data types, views, packages, procedures, functions, and relationships.

2.3 Query Language

A query language is a mechanism to read or access the data stored in a database. All databases provide their own query language implementation to let users execute the query and get results from the database. For example, MongoDB have MongoDB Command Line Interface based on javascript, MySQL have Structured Query Language, Cassandra have Cassandra Query Language.

2.4 Federated database

A federated database system acts as a meta-database management system which "maps different autonomous databases into a single federated database" [30]. Each independent homogeneous databases are located in different places and interconnected through the network connection. Federated database acts as a middle man between these databases regardless of how the data is stored and merge the results from all the databases. Federated databases allow users to use a unique querying platform and execute a single query to read and store the data from various types of databases even though the databases are heterogeneous. Federated database systems divide the single query into subqueries according to the underlying database systems and accumulate the result sets from each subquery and combine them into a single final response. FDS internally have different types of wrappers to translate the subqueries to appropriate database query language.

In FDBS architecture may consist of centralized or decentralized (distributed) databases where centralized system controls a single database instance, and decentralized system controls multiple dependent/independent database instances. An FDBS can be a nonfederated database system if any one of the databases is non-autonomous in the participating group. So it means that a system can be called as FDBS only if all the participating databases are entirely autonomous and should "allow partial and controlled sharing of their data" [30].

2.4.1 Loosely coupled FDBS

Loosely coupled federated systems define their own schema format that is used by everyone to access the databases involved in the federation. This approach forces the user to learn the federation schema to work with multi-databases.

2.4.2 Tightly coupled FDBS

Tightly coupled federated systems have separate processes for every database involved in the federation to build and export integrated federated schema.

2.4.3 Heterogeneity

There are significant factors that cause the heterogeneity in database systems such as semantics, structures, and the query language. Semantic heterogeneities occur if there is a conflict between the meaning of attributes, and there are a few well-known conflicts mentioned below,

- Data representation conflict
- Data conflict (missing attributes)
- Metadata conflict
- Precision conflict
- Naming conflict
- Schema conflict

Structural heterogeneities happen when there are different primitives between the two data models.

2.4.4 Data abstraction

Data abstraction in the context of database systems hides crucial complexities through many levels 2.1, and it is vital for constructing a federated database system.

- Physical level abstraction is the lowest level of abstraction which handles the data storage in the real systems.
- Logical level abstraction handles the type of data being stored in the database and the relationships between those data.
- View level abstraction is the highest level of abstraction which modularizes the big database system into smaller structures because users may not need to access complete information about the database, rather they interest in few parts of the database.

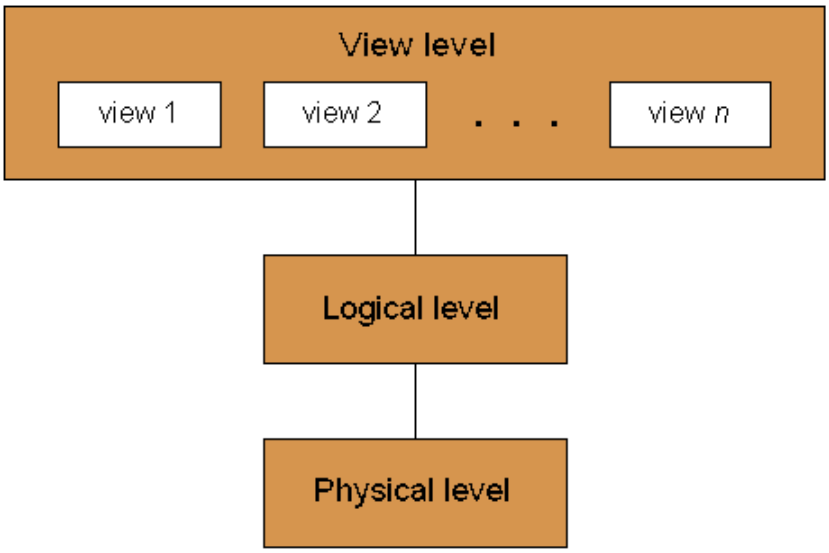


Figure 2.1: Different layers of data abstraction in database systems [31]

2.5 JSON Schema

JSON Schema is extensively used in this research work for mapping possible sensor input data attributes and types with the GraphQL type definitions. JSON Schema is "a JSON based format for defining the JSON data" [15]. It is a complete specification to define the types of each field in the data, restrictions for those fields, and marks the required fields in the data. A simple example is illustrated in the figure 2.2 which shows the JSON schema for a piece of robot information.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Robot schema",
  "type" : "object",
  "properties": {
    "name" : { "type": "string" },
    "macAddress" : { "type" : "string" },
    "accessUri" : { "type": "string", "format" : "uri" },
    "manufacturedYear" : { "type": "integer", "minimum" : 2010 }
  },
  "required": ["name", "macAddress"],
}
```

Figure 2.2: Simple JSON Schema represents the properties and constraints of Robot JSON document.

In the JSON Schema document, first attribute '$schema' defines the JSON schema draft version which is followed to create this document. This attribute helps the consumers or tools to parse the document with appropriate versions. Next, the 'title' attribute defines the title for the JSON schema, 'type' defines the data type of the real JSON data. For example, the given example tells that the robot information will be an 'object'. Then, 'properties' filed holds all the possible attributes of the robot object and their types and format. One can add a constraint on any field in the properties like 'manufacturedYear' property have a constraint of 'minimum' value should be 2010. Also, users can define the format for the value like date, URI, uuid, etc. Finally, the required field indicates the name and macAddress properties are mandatory.

2.6 @context

@context in JSON-LD document used to map the terms to their original context. A term represents a key-value pair in a JSON document. With the help of context, a term can be expanded to a full URL.

2.7 Node js

Node.js is a cross-platform JavaScript run-time environment used to run JavaScript programs without a browser. JavaScript programs are meant to execute in the client side browser's JavaScript engine. NodeJS let the developers write and run JavaScript programs in an isolated environment called node which uses Google's V8 JavaScript engine. It is used to develop command line tools and server-side scripting, and it overcomes the gap between client and server side programming since traditionally developers use two different languages on client and server side. In our research work, we develop the mediator component as a Node js application for a various good reason, and it will be discussed in the mediator component implementation section.

2.8 MongoDB

MongoDB is a document database that provides high performance, high availability and automatic scaling. MongoDB stores data in the form of document and it consists of field and value pairs which are similar to JSON objects. The value

may include other documents, arrays, and arrays of documents. A set of documents belongs to a collection.

2.9 MySQL

MySQL is a well-known Relational Database Management System. It stores data in tables and relations can be defined between tables using primary and foreign keys to represent the connection between data. It indexes data based on the primary key which improves the speed of reading query execution. Data model or schema should be defined before inserting data into tables. It provides ACID property to provide strong consistency on the data and it also available depends on the chosen configuration.

2.10 Docker

"Docker is an open source platform for developing, shipping and running applications" [8]. Important components of docker are, images, and containers. Images are nothing but a software package or operating system like Ubuntu, Nginx, etc., Using docker platform, one can quickly combine images and necessary software packages to deploy a container. A container is an isolated component, which runs individually in the docker engine, and multiple containers can be connected and communicated with each other in the same network. Docker engine shares the host machines resources (CPU, Memory) with the running containers. We can write a new image file on top of other base image file to build our customized containers.

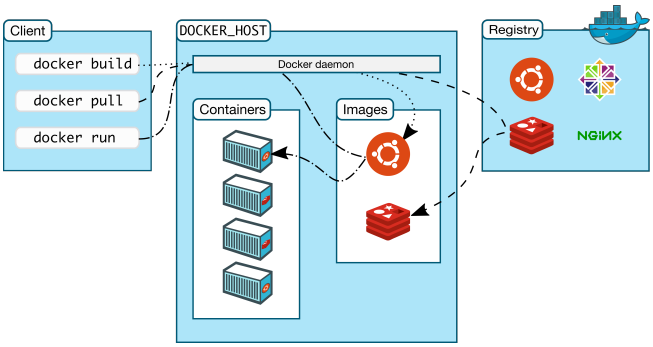


Figure 2.3: Docker architecture [9]

3

State of the art analysis

3.1 Databases for Robotic applications

Dietrich et al. [7] integrated a well known NoSQL database Cassandra with the Robotic Operating System(ROS) to handle the data in smart environments. Evaluated this approach in two scenarios, within a realistic robot exploration, and with randomly generated data. The final solution is compared with two other solutions that are commonly used in ROS applications(rosbag and mongo_ros). The research results show that Cassandra can handle terabytes of data and their time-stamp mechanism allows querying and retrieving data without additional efforts. But this approach compared their results only with two other storage mechanisms.

Fourie et al. [14] discussed the importance of memory recall mechanism for robots briefly. Also, implemented a two-level database layer in the centralized server, to store simple data(odometric values) in a graph database (Neo4j) and more considerable sensor data like RGB and depth image and laser scans in key-value store (MongoDB). With this architecture, they can achieve the sharing of workload between robot and database server. As a result, for executing this query server took between 10 to 100 milliseconds. In this work, they have given an initial motivation to the robotics community that using databases is possible and advisable, but the author didn't specify why Neo4j and MongoDB for data storing even though there are a lot of database technologies available.

Fiannaca and Huang [13] briefly explains the difficulties in handling the log

messages from ROS(Robotic Operating System) and alternate ideas to store ROS logs for utilizing it in the future. To overcome the challenges in storing logs in a flat file system, they wanted to use the relational database or NoSQL database. So a benchmarking test has been conducted to evaluate and find which one is best. Three databases (MongoDB, PostgreSQL, SQLite3) from NoSQL and relational database family has been chosen for benchmarking. But in recent years, PostgreSQL and SQLite3 databases are little outdated, and MongoDB is not only the NoSQL database available. New NoSQL databases are emerging every month and might perform better than MongoDB. So these evaluation results are not useful to choose the best NoSQL database to store sensors logs and benchmarking only one NoSQL database is not an optimal choice.

From all these approaches discussed above, they focused on particular applications/settings and are always comparing only very few databases within this limited setting. Moreover, they benchmarked the performance with a single robot in most of the cases. Therefore, in our previous work [23], we have conducted a qualitative and quantitative analysis of varieties of databases to find the best performing databases for multi-robot applications. But the results show that the performance differs for different scenarios and type of sensor data. Hence to support multi-database in robotic applications, we chose to create a mediator component with generalized semantic data models that help to solve querying heterogeneous data in a meaningful way. In the next section, we review types of existing semantic data models and mediator architectures that support this research work.

3.2 Semantic Data models for sensors

Su et al. [26] highlights the interoperability issues in IoT sensor data and also says that these data should be useful for multiple applications rather than dependent on specific domain. To make the machines interpret the meaning of sensor data, author suggested to use Semantic Web technologies such as Resource Description Framework (RDF). Even though SenML is an evolving technique to model to sensor data, but it is lacking reasoning capabilities and interoperability with other devices. To overcome the issues in SenML generic model, author proposes a technique to represent IoT sensors as a Knowledge Based Systems by transforming SenML to Resource Description Framework.

As an advantage, the transformed data can be analyzed and helpful to take more meaningful actions. SenML is specially designed for resource constrained devices, hence additional information to contextually understand the data has not been included intentionally. Each entry of SenML data should have the sensor parameter name and other attributes such as time, value, etc. Also it supports custom attribute called Resource Type (rt) to let the users add their own attributes and this allow users to include contextual information.

With the help of transformation we could adopt RDF structure to model robot generated sensor data, But Charpenay et al. [5] points out that RDF data structure is not suitable for resource constrained devices like micro-controllers and also analyzed the issues in RDF such as verbosity and complexity in processing knowledge. To overcome this issues, author carried out an extensive analysis between RDF and JSON-LD structures. JSON-LD was published by W3C, and it serves as an alternative for RDF. Using JSON-LD one can represent the context for the data which is more important in robotics field such that other robots can understand the data based on context. Results shows that JSON-LD compaction coupled with EXI4-JSON or CBOR outperforms state-of-the-art (HDT) with **50 - 60 %** compaction ratios. This is one of the reason why we chose JSON-LD to represent the context in our data models.

3.3 Mediator architectures

In this section, different approaches for mediator architectures and mechanisms for querying heterogeneous data are presented to grasp the core ideas and how could we optimize the existing mediator models.

Fahl et al. [12] proposed an active mediator architecture to gather information from different knowledge base and combine them to a single response. AMOS[1] architecture uses Object-Oriented approach to define declarative queries. This distributed architecture involves multiple mediator modules to work collaboratively to collect the required piece of information and produce final result. Primary components of AMOS architecture are,

- Integrator - Gather data from multiple data sources that have different data representations.

[1] Active Mediators Object System

- Monitor - Monitor service always watch for any data changes and notifies the mediators. This is helpful in the case where system needs an active updates to change its current task.
- Domain models represents the models related to application which helps to access data easier from any database through a query language.
- Locators helps to locate mediators in the network.

Integrator module is built with two internal components called IAMOS[2] and TAMOS[3]. First "Integration AMOS (IAMOS)" parse the query and send individual requests to Translational AMOS modules which are responsible for heterogeneous data source. Then, all TAMOS modules return the individual results to IAMOS for integrating all the results. To query multi databases from IAMOS, IAMOS servers are mapped with TAMOS servers with the help of Object-Oriented query language.

Ahmed et al. [2] developed a system called Pegasus that supports multiple heterogeneous database systems with various data bases models, query languages and services. Pegasus predefines its domain data models based on object oriented approach and also supports programming capabilities. These objects are created and mapped with the types and functions with the help of HOSQL[4] statements. HOSQL is a declarative object oriented query language which is used by Pegasus to manipulate data from multiple data sources.

Pegasus system supports two types of data sources, local and native data sources. Whenever a new data source joins Pegasus system, schema integrator module imports schema from data source and update its root schema with the new schema types. The final integrated schema shows the complete blueprint of the different data sources participates in the data integration. Pegasus system work-flow is comparatively similar to AMOS architecture, but they use different query language and data modeling strategies.

Chawathe et al. [6] developed project Tsimmiss extract information from any kind of data source and translates them to a meaningful common object. Unlike AMOS and Pegasus, Tsimmiss follows a straight forward approach to define the data

[2]Integration Active Mediators Object System
[3]Translation Active Mediators Object System
[4]Heterogeneous Object Structured Query Language

model which is a self-describing object model. Each object must contain a label, type and value itself. Label can be used by the system to understand the meaning of the value and type shows the observed value type. Objects can be nested together to form a set of objects.

Tsimmiss tool offers a unique query language called OEM-QL and this language follows the SQL query language pattern to fetch the data from mediators. Mediators resolves the query and send separate requests to respective data sources to retrieve the information and merge them together to give a single response back to user.

In the articles discussed above, mediators are targeted to extract information from different data sources that could be different databases or data from file-system. But Rufus system proposed by Shoens et al. [24] focus only on semi structured data stored in file system for example documents, objects, programming files, mail, binary files, images etc. Rufus system classifier automatically classifies the type of file and apply a scanning mechanism on those files to extract the required information and transform them to the appropriate data model which is understandable by Rufus system. Rufus can classify 34 different classes of files.

Papakonstantinou et al. [22] proposes a Mediator Specification Language (MSL) that helps the mediator to understand the schema and integrates the data from unstructured or semistructured source. MSL overcomes the major problems in existing mediator systems for example,

- *Schema domain mismatch* - Attribute name mismatch or attribute is fragmented to sub-attributes between different domains. For example, domain A uses attribute "fullname" that consists of both first and last name, and domain B uses two attributes "first_name" and "last_name" to store users full name.
- *Schematic discrepancy* - Storing data between tables inconsistently. For example, in "domain A" users data is stored in the "user" table, and their status is stored in "status" table, and in "domain B" the status is stored in "user" table itself.
- *Schematic evaluation* - Data schema and attributes may change over time without notifying mediator system. For example, the "age" attribute is added in one domain or data type of "price" is altered from Integer to Float.
- *Structure irregularities* - Dissimilarity between data schemas from different

domains.

During translation of original information from different sources to a single object it should be important that, all data sources should have the required attribute and the name of the attribute should be same. Otherwise, mediator system will not be able to process the information to a single answer. External predicates and Creation of the Virtual Objects in MSL solves the problems mentioned above.

Arens et al. [3] built a mediator which is flexible to map domain level query different data-sources and efficient to plan the query execution to reduce the overall execution time. Information source models provides relations between the super class and subclasses, and also the mapping between the domain models and information from heterogeneous sources. SIMS uses Loom as a representational language to make objects and relationship between them. SIMS supports parallel query access plan that makes the mediator to access information independent of data sources and the user will get the final answer as quick as possible.

In summary, all the architectures reviewed above focus on one significant activity of mapping the heterogeneous data to a single representation. However, each architecture proposes its own way of representing data and a tool to query the federation data. This way imposes more load on the developers to translate their existing data schema into the mediator required representation and learn a new query language. We overcome this issue with the help of our database schema translator that automatically translates the schema of the existing data model to the mediator required format. Also, using views to abstract data schema is one possible approach to solve heterogeneity. However, in our strategy, we tried to add on-demand contexts using JSON-LD for each entity in the robot ecosystem to make the data consumable by other robots/tools. In the next section, we describe why context is important to identify the attributes in a data semantically. Finally, none of the architectures considered declarative data fetching, but in this research work, we give attention to declarative data fetching frameworks which improves the flexibility for the users to fetch only required attributes. It will also be useful when robots want to save data and reduce latency in poor network connectivity areas.

3.4 Declarative data fetching frameworks

Cederlund [4] performed an extensive comparison between REST, GraphQL and

Falcor by declarative data fetching. They evaluated all three frameworks based on latency, data volume, and many requests with real-world test cases. Also, they analyzed the efficiency of filtering done by the frameworks.

Their results reveal that Relay+GraphQL decreases the response time under parallel and sequential data flow. Furthermore, the response size is decreased when using the frameworks rather than REST API's. However, within the frameworks, Falcor response time and size is high compared to GraphQL. Ultimately, both the frameworks reduced the number of network calls to a single request.

As a conclusion, Cederlund [4] suggests to use custom REST endpoints since the frameworks increase the size of requests, but it still depends on the application requirement. In our application, Robots might work in the places where limited network access available, so we definitely use these frameworks as a base on our mediator to reduce the response size. Moreover, the observation data may have too many information which are unnecessary for the other robots or tools to work on. Therefore only picking the necessary fields in the response profoundly reduce the response size as well as the response time.

Many mediator systems have been developed in the past to support integrating heterogeneous information from different data sources. All of them built with different architectures, query language, and execution optimization. In our mediator approach we focus mainly on,

- How different type of robot generated data will be stored in multiple data sources?
- A unique context based data model to represent the components attached with each robot and data generated by them.
- Semantic query language to communicate with mediator. Unlike traditional query languages we would like to attempt new way of querying data, for example Graphql.
- GUI tool to visualize and analyze the robot generated data in a meaningful way.

4

Problem Statement

Our previous work results reveal not all databases reacts similarly for different heterogeneous data from robotic applications. Also, there are no concrete data models has been defined in the context of robotic applications. For example, the data logger in the black box designed for ROPOD project uses MongoDB to store data from different sources such as Ethercat, Zyre, ZMQ, and ROS topic. The data is being transformed into a simple flatten JSON document to store the values. These documents are stored under a single collection[1] which is created for each ROS topic or other sources. Each record holds only the information of data generated by the sensors or application itself, but these values are not useful without additional details for example, who created the data, if it is a robot then what type of robot-generated this data from which location? Then in what context other systems should interpret this data.

Listing 4.1: geometry_msgs/Pose ROS topic

```
double timestamp
double position/x
double position/y
double position/z
double orientation/x
double orientation/y
double orientation/z
```

[1]Collection in MongoDB is similar to a table in an RDBMS.

```
double orientation/w
```

For example in the black box, geometry_msgs/Pose ROS topic will be flattened to a simple JSON document as shown in the listing 4. In the above example, 'position/x' is a key and the value will be attached with it. Now only with position x,y,z and orientation x,y,z,w, another system which consumes this data would not be able to say who generated this data or at which location this data is being generated and if the other system is doing mathematical calculation, then this data is missing its own context such as unit, dimensions, etc.

Periodically, these massive amounts of data are dumped and backed up to a file system or cloud. After every test run in the black box, a report is generated using the FMEA tool which contains the information regarding the test and components involved in it. Also, these reports include the file location where the dump is stored. During fault diagnosis, these dumps will be restored manually to the database and fetch data using the querying tool provided by the black box itself.

This approach is not scalable and inefficient in terms of multi-robot systems since there will be individual database instances running in each robot. Moreover, this querying tool is incapable of making queries on multiple MongoDB instances at a time.

In terms of supporting various types of databases setup for robots, there is no systematic approach to store and retrieve data from external sources. Also, a well-defined data model hasn't defined yet that can map robot components (e.g., sensors) to a robot and even with the world model (e.g., locations). In this case, no mediator system has been developed before to connect between robots and different databases.

5

Concept and Methodology

5.1 Approach

To find the best data models for robotic applications and build a scalable mediator, we begin with SOA analysis 3 to find out approaches that have been followed through before for similar data integration applications. After defining the data model, we will collect a list of recent querying techniques and review them based on the features and possibility of adopting them with the mediator as a base. For review, we would like to consider current well-known querying techniques such as Graphql, and Falcor. At first, our mediator will support only the databases which are selected based on the results from our previous research work [23] and other data sources used by ROPOD such as OpenStreetMap. Then, schema's will be defined to map the data being generated by the robot and the data sources. To reduce the complexity of identifying appropriate data-sources by the robot, in our architecture mediator will dynamically choose the data-source respective to the type of data that robots want to store and retrieve. Still, the configuration will be adjustable according to the scenarios. The architecture proposed in this research is a general design and can be used by anyone who wants to store and analyze the data from the multi-robot system contextually. Finally, to show how one can configure the mediator to an external tool to visualize the heterogeneous data from the mediator in a meaningful way, we used GraphiQL [1] GUI web application to make queries against the mediator system.

[1] https://github.com/graphql/graphiql

5.2 Analyzing existing sensor data model specifications

In this section, we analyze the existing data model specifications which are widely used to represent sensor generated data. Additionally, to find out best context based data models we reviewed Resource Description Framework and JSON-LD.

5.2.1 SenML

SenML is an encoding format to represent the sensor values as simple as possible; therefore a simple microcontroller can process it with little memory and computation resource.

Single SenML message can consist of an array of sensor measurements along with minimal additional information to describe the sensor itself. This information will solve the interoperability when other systems try to understand the data. However, due to efficiency reasons, it can hold only a few meta information such as sensor name, and their unit. Before analyzing the examples, we can look at the semantics defined by SenML in the below table 5.1.

Figure 5.1 shows the possible basic example of SenML structure to represent a sensor generated data, and the data holds a single observation of voltage from a voltage sensor.

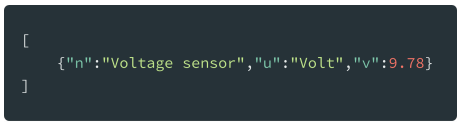


Figure 5.1: Simple SenML observation structure

Figure 5.2 shows a complex group of measurements which includes values from voltage and humidity sensors. While retrieving these measurements, the base name will be prepended with names and base unit 'RH' will also be added to all measurement except for those measurements that have a unit. For instance, in the above example voltage measurement have its unit 'V' attached to it already. So in this case, the base unit will not be added to this measurement.

To process or semantically query complex sensor data, measurements should have additional context about the data. However, it is not achievable directly with

Semantics	Definitions
bn	Base name which is used to attach before all the names in the measurement and it will reduce the size of the complete message.
bt	Base time represents the start time of the first measurement being measured.
bu	Base unit, it will be useful if all the measurements have the same unit.
bv	Base value, it will be added to the values in the measurement.
bver	Version of the SenML media format.
n	Name of a sensor, and base name will be prepended with this name.
u	Represents the units of the observed measurement.
v	The value (number) generated by the sensor.
vs	String value.
vb	Boolean value.
vd	Data value (base64 encoded string).
s	Value sum.
ut	Update time.
t	Time.

Table 5.1: SenML semantics used to build SenML messages

```
[
    {"bn":"urn:ropod:testlab","bt":1.123450099001e+08,
     "bu":"RH","bver":4,
     "n":"voltage","u":"V","v":160.4},
    {"n":"humidity","t":-5,"v":13.9},
    {"n":"humidity","t":-4,"v":12},
    {"n":"humidity","t":-3,"v":12.5},
    {"n":"humidity","t":-2,"v":14.3},
    {"n":"humidity","t":-1,"v":14.4}
]
```

Figure 5.2: Complex SenML observation structure

SenML, instead of as explained in this article [26] with the help of transformation from SenML to RDF one can explicitly specify the context of each measurement.

5.2.2 OGC SensorThings API

OGC SensorThings API offers a unique way to connect sensors in IoT platform and an entity based relationship model for handling data from heterogeneous sensors. It has two major components, one for sensing part and other for tasking part. For our research work, we are not going to review the Tasking part, since our focus is towards finding a better context-based data model for robotic applications.

Sensing part follows the Observation and Measurement model from "OGC 10-004r3 and ISO 19156:2011". There are eight entities defined by SensorThings API.

5.2.2.1 Thing

Thing represents the physical things in the world or a virtual system that can be communicated via the network. Things have their own attributes such as name, description, and properties. A single thing can have many optional locations, historical locations, and data streams. The constraint is a thing that should be mapped to a single location at a given point of time.


```
{
   "@iot.id":4,
   "@iot.selfLink":"http://example.org/v1.0/Things(4)",
   "Locations@iot.navigationLink":"Things(4)/Locations",
   "Datastreams@iot.navigationLink":"Things(4)/Datastreams",
   "HistoricalLocations@iot.navigationLink":"Things(4)/HistoricalLocations",
   "name":"Production line",
   "description":"This thing is a production line.",
   "properties":{ "Manufactured by":"Hundai", "Weight":"1000" }
}
```


Figure 5.3: Structure of Thing entity

5.2.2.2 Location

Location shows the coordinates of the real location and each location may locate multiple things. Each location should be provided with its encoding type to make the other robots/users quickly understand the location type and utilize them for calculation.

```
{
    "@iot.id":5,
    "@iot.selfLink":"http://example.org/v1.0/Locations(5)",
    "Things@iot.navigationLink":"Locations(5)/Things",
    "HistoricalLocations@iot.navigationLink":"Locations(5)/HistoricalLocations",
    "encodingType":"application/vnd.geo+json",
    "name":"HBRS",
    "description":"Hochschule Bonn Rhein Sieg university of applied science",
    "location":{ "type":"Feature", "geometry":{ "type":"Point", "coordinates":[ 88.06, -95.05 ] } }
 }
```


Figure 5.4: Structure of Location entity

5.2.2.3 Sensor

Sensor entity represents the physical sensing device which generates the values. Each sensor should have at least one data stream entity relation. Each sensor is described along with its encodingTypa and an optional metadata field.


```
{
   "@iot.id":11,
   "@iot.selfLink":"http://example.org/v1.0/Sensors(11)",
   "Datastreams@iot.navigationLink":"Sensors(11)/Datastreams",
   "name":"TMP36",
   "description":"TMP36 - Analog Temperature sensor",
   "encodingType":"application/pdf",
   "metadata":"http://example.org/TMP35_36_37.pdf"
}
```


Figure 5.5: Structure of Sensor entity

5.2.2.4 HistoricalLocation

HistoricalLocation shows the thing to be in the location or vice versa, at a given timestamp. For example, consider a robot (thing) is moving from one room to another room for a specific task. To make the relation between the robot (thing) and the room (location), we create HistoricalLocation data with the reference of the robot and the room along with a timestamp. Timestamp plays a significant role here because it shows the truthness of the thing to be present physically in any given location.

```
{
   "value":[
       {
           "@iot.id":2,
           "@iot.selfLink":"http://example.org/v1.0/HistoricalLocations(2)",
           "Locations@iot.navigationLink":"HistoricalLocations(2)/Locations",
           "Thing@iot.navigationLink":"HistoricalLocations(2)/Thing",
           "time":"2019-02-17T12:00:00-07:00"
       },
       {
           "@iot.id":3,
           "@iot.selfLink":"http://example.org/v1.0/HistoricalLocations(3)",
           "Locations@iot.navigationLink":"HistoricalLocations(3)/Locations",
           "Thing@iot.navigationLink":"HistoricalLocations(3)/Thing",
           "time":"2019-02-18T14:00:00-07:00"
       }
   ],
   "@iot.nextLink":"http://example.org/v1.0/Things(4)/HistoricalLocations?$skip=2&$top=2"
}
```


Figure 5.6: Structure of HistoricalLocation entity

5.2.2.5 FeatureOfInterest

Each observation value represents the property of a feature, and with the help of FeatureOfInterest entity one can filter the observations easily. For example, FeatureOfInterest of a GPS sensor is location since it generates the coordinates of its current location.


```
{
   "@iot.id":1,
   "@iot.selfLink":"http://example.org/v1.0/FeaturesOfInterest(1)",
   "Observations@iot.navigationLink":"FeaturesOfInterest(1)/Observations",
   "name":"ROPOD lab at HBRS",
   "description":"Production line setup in ROPOD lab",
   "encodingType":"application/vnd.geo+json",
   "feature":{ "type":"Feature", "geometry":{ "type":"Point", "coordinates":[ -124.06, 121.05 ] } }
}
```


Figure 5.7: Structure of FeatureOfInterest entity

5.2.2.6 Datastream

Datastream stores the list of observations for a specific thing and a sensor. Each data stream should have at least one sensor and thing entity relationship. For example, a gateway (Thing) with a temperature sensor (Sensor) generates a list of temperature

observations under Temperature Datastream. Also, datastream holds the type of observation and area which defines the coordinates of the location from where the robot generates the actual observation.


```
{
   "@iot.id":10,
   "@iot.selfLink":"http://example.org/v1.0/Datastreams(10)",
   "Thing@iot.navigationLink":"HistoricalLocations(1)/Thing",
   "Sensor@iot.navigationLink":"Datastreams(10)/Sensor",
   "ObservedProperty@iot.navigationLink":"Datastreams(10)/ObservedProperty",
   "Observations@iot.navigationLink":"Datastreams(10)/Observations",
   "name":"production line temperature",
   "description":"This is a datastream of temerature measured in production line",
   "unitOfMeasurement":{ "name":"degree Celsius", "symbol":"°C",
       "definition":"http://unitsofmeasure.org/ucum.html#para-30"
   },
   "observationType":"http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
   "observedArea":{ "type":"Polygon", "coordinates":[ [ [100,0], [101,0], [101,0], [100,0], [100,0] ] ]
   },
   "phenomenonTime":"2019-08-01T13:00:00Z/2019-08-11T16:30:00Z",
   "resultTime":"2019-09-01T13:00:00Z/2019-09-11T16:30:00Z"
}
```


Figure 5.8: Structure of Datastream entity

5.2.2.7 ObservedProperty

ObservedProperty shows what phenomenon is being observed by Observation entity. And it should have a data stream entity referenced to it.


```
{
   "@iot.id":12,
   "@iot.selfLink":"http://example.org/v1.0/ObservedProperties(12)",
   "Datastreams@iot.navigationLink":"ObservedProperties(12)/Datastreams",
   "description":"Property of the temperature sensor limits the DewPoint.",
   "name":"DewPoint Temperature",
   "definition":"http://dbpedia.org/page/Dew_point"
}
```


Figure 5.9: Structure of ObservedProperty entity

5.2.2.8 Conclusion

OGC SensorThings API provide an open access to fetch all entities described above via HTTP network call, and it stores the data in a PostgreSQL relational database.

Since their architecture supports only PostgreSQL we cannot use the complete component in our mediator. However, the standard given by OGC SensorThings makes a meaning relationship between entities in IoT systems. So, our proposed data model is designed partially based on the OGC SensorThings standards and it is explained in section 6.2.2.1.

5.2.3 Resource Description Framework

Resource Description Framework (RDF) is a well known unique model to represent any resources in the universe with subject, predicate and object pattern like how humans communicate with each other.

For example, consider a simple scenario where person one says to person two that "Christoper is a magician". In this statement 'Christoper' is subject, 'is a' is a predicate (relation) and 'magician' is an object. In a simple case, person two understands the statement if the person two knows only one Christoper in his/her life. However, if person two have a reference of multiple Christoper's, then it is unclear that which Christoper he/she is referring to? Then person two asks another question, which Christoper and where he is from? So now person one makes a new statement, "Christoper is from Bonn". In this statement, "Christopher" is the same subject, 'is from' a new predicate and "Bonn" is a new Object of type city. Now a person two enough information to infer which Christoper person one is talking about.

Communication is way more easier between humans since they have a standard language model. However, What about machines? How can they communicate in a meaningful way? Alternatively, what if a person wants to communicate with a machine in the same way he/she communicates every day. This is where RDF plays a significant role in representing the data that a machine generates in a triples format aka Subject-Predicate-Object.

5.2.3.1 Components of RDF

RDF structure consists of two major components called Document and Statement. Document is the root container to have more than one RDF statements. In figure 5.11 'xmlns:rdf' attribute represents the namespace of the current RDF document

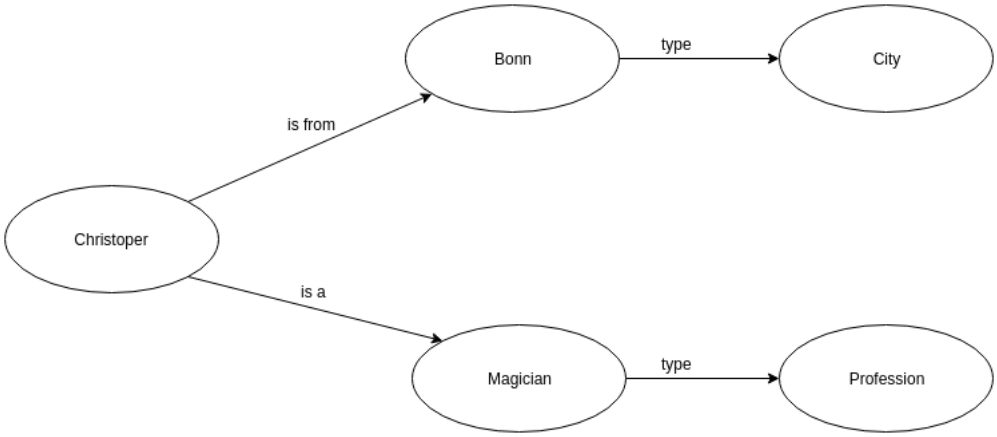


Figure 5.10: Illustrates a simple scenario in the form of RDF structure

which gives a hint to the machines about how they can parse and understand this document.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <-- RDF statement will go here -->
</rdf:RDF>
```

Figure 5.11: Basic skeleton layout of Document

Statement is a individual building block to represent a single triple. In the given example figure 5.12, RDF statement starts with subject description about Christoper and in the next level it has a predicate of 'is-a' pointing towards another resource 'Magician' which is an object in this statement. Each statement can have either a reference to other RDF statement or a value. 'Magician' and 'Bonn' statements in the example pointing to other RDF object via 'resource' attribute, and 'age' statement have a single value of '26' represents the age of the 'Christoper'.

The primary advantage of adding semantics to the data is interoperability and easy to query them for a meaningful answer. Consider an example in the context of robotic applications. In multi-robot scenarios, more than one robot might work cooperatively to complete a single task. Let say, different vendors manufacture each robot, and the sensors generate values in different units or context. Robot one wants

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:foaf="http://www.schema.org/foaf#"
   xmlns:place="http://www.schema.org/place#"
   xmlns:person="http://www.schema.org/person#">
     <rdf:Description rdf:about="http://www.schema.org/persons#Christoper">
       <foaf:is-a rdf:resource="http://www.schema.org/Profession#Magician"/>
       <place:is-from rdf:resource="http://www.schema.org/Place#Bonn"/>
       <person:age>26</person:age>
     </rdf:Description>
 </rdf:RDF>
```

Figure 5.12: Basic skeleton layout of Statement

to share its current location information which is encoded by 'latitude and longitude' type to robot 2, and robot 2 sensor generates location information with 'geohash' encoding type. So whenever these robots share location values between them, it is not possible to consume it without the context of the values being encoded. If they share values along with encoding type as a context, then each robot can transform the values to the encoding type which is currently being used for the calculation.

5.2.4 JSON-LD

JSON-LD is an approach to structure the linked data and adding semantic contexts to data in the form of IRI (Internationalized Resource Identifier). An IRI is a unique and globally accessible link via web like URI, and the format is jointly defined by World Wide Web Consortium and Internet Engineering Task Force [27]. Let us try to understand how JSON-LD solves interoperability issues with a toy example. Consider we have two robots developed by different developers and each publishing ROS Twist messages at a specific rate.

Developers of the first robot decided to publish the twist messages with the following format.

```
{
   "linear": { "x": 2, "y": 3, "z": 23.2 },
   "angular": { "x": 23, "y": 33, "z": 0.5 }
}
```

Figure 5.13: Twist message created by robot one

Developers of the second robot decided to publish the twist messages with the following format.

Figure 5.14: Twist message created by robot two

We can see from both the format, underlying data is the same, but the representation is different. In the first robot, the linear and angular velocity has been identified with the keys "linear" and "angular", and in the second robot, the same data will be identified with the keys "linear_velocity" and "angular_velocity". In a multi-robot environment, if all the robots know what data they are exchanging and how it has been encoded, then there are no issues. However, what if two stranger robots want to communicate or transfer data with each other? For example, robot one wants to navigate from one location to another without colliding with other robots in the environment. With the formats mentioned above, robot one cannot understand what robot two is saying, because they do not talk in the same language.

This is where JSON-LD plays a significant role to solve this interoperability issue. Let solve the issue discussed above with the help of JSON-LD.

Figure 5.15: Twist message created by robot one in JSON-LD format

What has been changed in the first robot twist message? We have added a "@context" keyword to the existing message. This means that, whoever consumes this message, they have to understand the complete message in the context of "http://example.com/". Now we transform the second robot message format like this.

```
{
  "@context": {
      "@vocab": "http://example.com/",
      "linear_velocity": "linear",
      "angular_velocity": "angular"
  },
 "linear_velocity": { "x": 3, "y": 43, "z": 73.2 },
 "angular_velocity": { "x": 0.6, "y": 13, "z": -0.5 }
}
```


Figure 5.16: Transformed twist message created by robot two in JSON-LD format

In the new transformation, we added one more attribute called "@vocab" which means that apply the "http://example.com/" IRI to all keys with the specific key mapping. Now one may think that how this solves the interoperability issue? Still, both messages look exactly different with few extra @context and @vocab information.

JSON-LD does not offer only the semantic representation, but also offers two other important features called Expansion and Compaction algorithm.

5.2.4.1 Expansion algorithm

Expansion algorithm takes an object as an input along with its context and applies the context into the data attributes and expands all compact IRIs to absolute IRIs. Also, during the expansion, it expresses all JSON-LD values in the expanded form. Finally, it removes the "@context" information from the result data since all information is propagated to the real data. Let us apply the expansion algorithm on the twist messages created by robot one and two, and see the result after applying the expansion algorithm.

Now both the messages in figure 5.17a, 5.17b look exactly the same without any difference in keys. This is the power of JSON-LD expansion algorithm. One can probably think now, who creates the common vocabulary list which can be understandable by all the robots. So far, there are many vocabulary lists which have been created for common things in the universe, but there is no vocabulary list composed for the robotic ecosystem. For example,

- https://schema.org/ - schema.org has a vast collection of most common definable things in the universe.

```
[
  {
      "http://example.com/angular": [
          {   "http://example.com/x": [{ "@value": 2 }],
              "http://example.com/y": [{ "@value": 3 }],
              "http://example.com/z": [{ "@value": 23.2}] } ],
      "http://example.com/linear": [
          {   "http://example.com/x": [{ "@value": 23 }],
              "http://example.com/y": [{ "@value": 33 }],
              "http://example.com/z": [{ "@value": 0.5 }] } ]
  }
]
```


(a) Transformed twist message created by robot one after applying expansion algorithm

```
[
  {
      "http://example.com/angular": [
          {
              "http://example.com/x": [{ "@value": 3 }],
              "http://example.com/y": [{ "@value": 43 }],
              "http://example.com/z": [{ "@value": 73.2}] } ],
      "http://example.com/linear": [
          {   "http://example.com/x": [{ "@value": 0.6 }],
              "http://example.com/y": [{ "@value": 13 }],
              "http://example.com/z": [{ "@value": -0.5 }] } ]
  }
]
```


(b) Transformed twist message created by robot two after applying expansion algorithm

Figure 5.17: Twist messages from robot one and two is same after applying JSON-LD expansion algorithm

- http://thingschema.org/ - thingsschema.org has just initiated a motivation to define things (smart things) in the environment.
- https://iot.schema.org/ - iot.schema.org defines the vocabulary list and relationships for IoT devices.
- https://wiki.dbpedia.org/ - dbpedia.org have a collection of person record to identify each person information in the world uniquely.

As part of this research work, we are taking the initiative to prepare a set of vocabs for robots, sensors and its working environment. Then, we use the vocabularies to represent robot generated messages with more precise context so that any other robots in the world can understand each other even though different vendors are manufacturing them.

5.2.4.2 Compaction algorithm

Compaction algorithm takes two inputs, one is the real data object 5.18 and the second is an object 5.19 which has only context information and generates a human-readable final result 5.20 which includes both data and context injected into it.

As it is stated above, ”@context” is injected into the real data object. Now the result object can contextually understandable by other robots.

Figure 5.18: First input to the compaction algorithm

Figure 5.19: Second input to the compaction algorithm

Figure 5.20: Result produced by compaction algorithm

5.2.4.3 Why JSON -LD instead of JSON?

JSON is a widely used format to exchange data between systems and storing persistently in databases. Also, there are many famous document databases(e.g., MongoDB, CouchDB, etc.) evolved which supports handling JSON objects. Any system can easily parse JSON objects, but the JSON object itself is meaningless. The context of each key can be shared manually with any form of documentation, but it is not easily accessible every time.

For example, a developer worked on a system and named the keys by his own interest. Now if a new developer wants to understand what a specific key means, then he/she needs to check the document or contact other developers. It is even more difficult for machines, to interpret what each key means in a JSON object. JSON-LD solves this problem by encoding IRI's in the JSON object to uniquely identify each key.

5.3 Comparison between GraphQL & Falcor

In this section, we are going to analyze the potential features of GraphQL and Falcor, also the possibility of adapting these frameworks as a base for our mediator. Before getting into GraphQL and Falcor, we should know what REST API is and how it transformed the way of communication between systems for many years.

5.3.1 REST API

Primarily API stands for Application Programming Interface, and it allows any software to talk with each other. There are different types of API available, but here in the context of GraphQL and Falcor, we consider REST API (REpresentational State Transfer API). Fundamentally, REST API works in the same way as how websites work. A client sends a request to the server over HTTP protocol, and the client gets an HTML (Hypertext Markup Language) page as a response and browser renders the page. In REST API, the server sends a JSON (Javascript Object Notation) response instead of HTML page. The JSON response might be unreadable to humans, but it is readable by machines. Then the client program parses the response and performs any actions on the data as they wish.

This architecture looks perfect for fetching the data from the server but what are the limitations in this architecture that give space for the emergence of frameworks like GraphQL and Falcor?

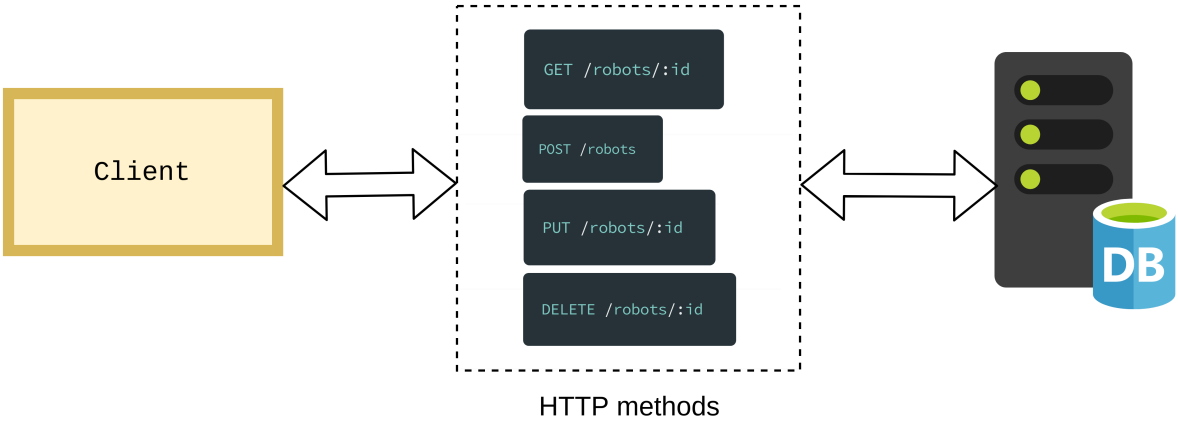


Figure 5.21: Traditional REST API Workflow

5.3.1.1 Limitations

- The client has to make recurring round trips to the server to fetch the required data.
- Various endpoints for different resources which make complication between developers and challenging to manage them on server and client side in big projects.
- Over Fetching, means there is no way to control the response to include only a subset of fields which bloats the response size and may cause network traffic.
- No static type validation on data sent or received.

In the later sections, we see how GraphQL and Falcor address the limitations in their approach.

5.3.2 GraphQL

GraphQL is a Query Language developed by Facebook to fetch the data from the database unlike the traditional way of making REST API requests. Technically, GraphQL replaces the use of REST API calls with a single endpoint on the server. Single endpoint architecture solves various communication difficulties between client and server side team members.

5.3.2.1 Single roundtrip

GraphQL helps to fetch all the data we required in a single request. For example, consider a scenario in the below figure where we need to get top 10 robots and sensors attached to it.

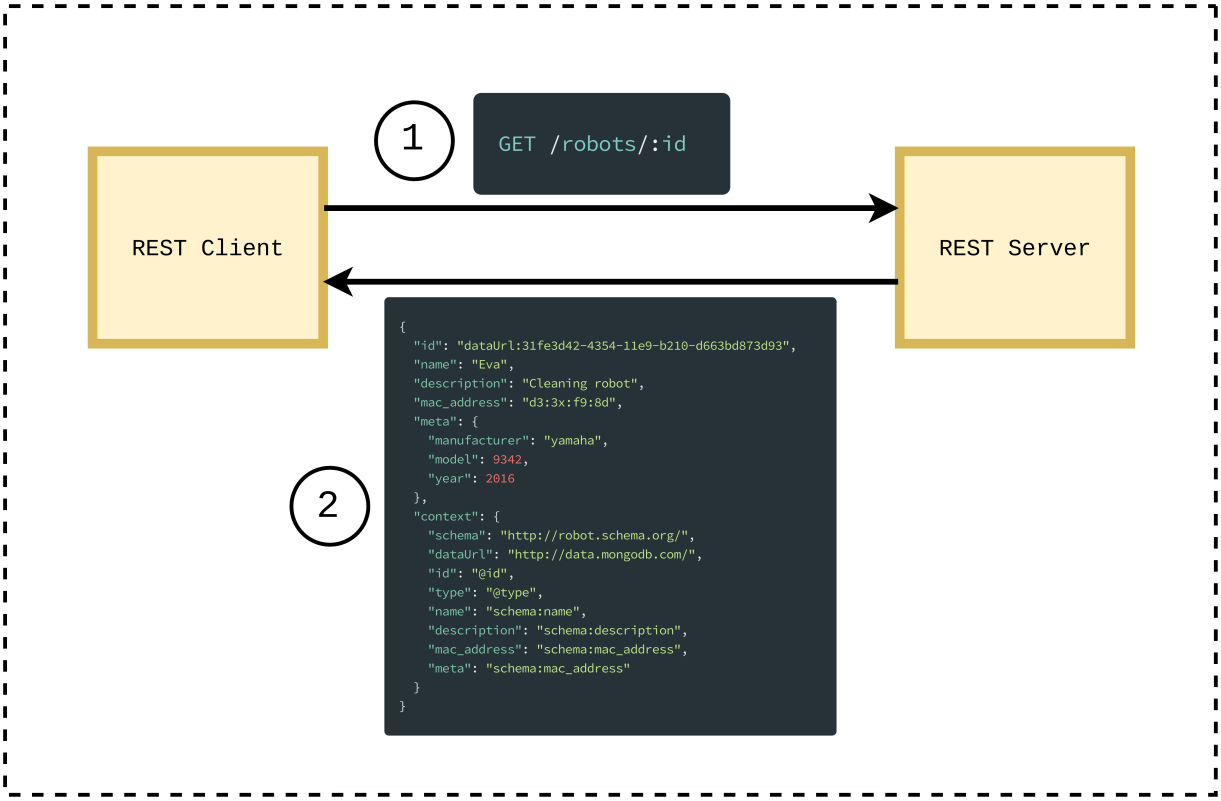


Figure 5.22: Multiple round trips in REST API (a)

Figure 5.22 shows the traditional REST approach, step 1 shows single GET request to fetch a piece of robot information and step 2 shows the JSON response that includes the robot information which is requested. Note that, server spits out all information regardless of what client is going to use in their application.

In figure 5.23 step 3 shows the next GET request from the client to fetch all the sensor information belongs to the robot_id which client received in step 2 response. In step 4, the server responds with all sensor details for the specified robot.

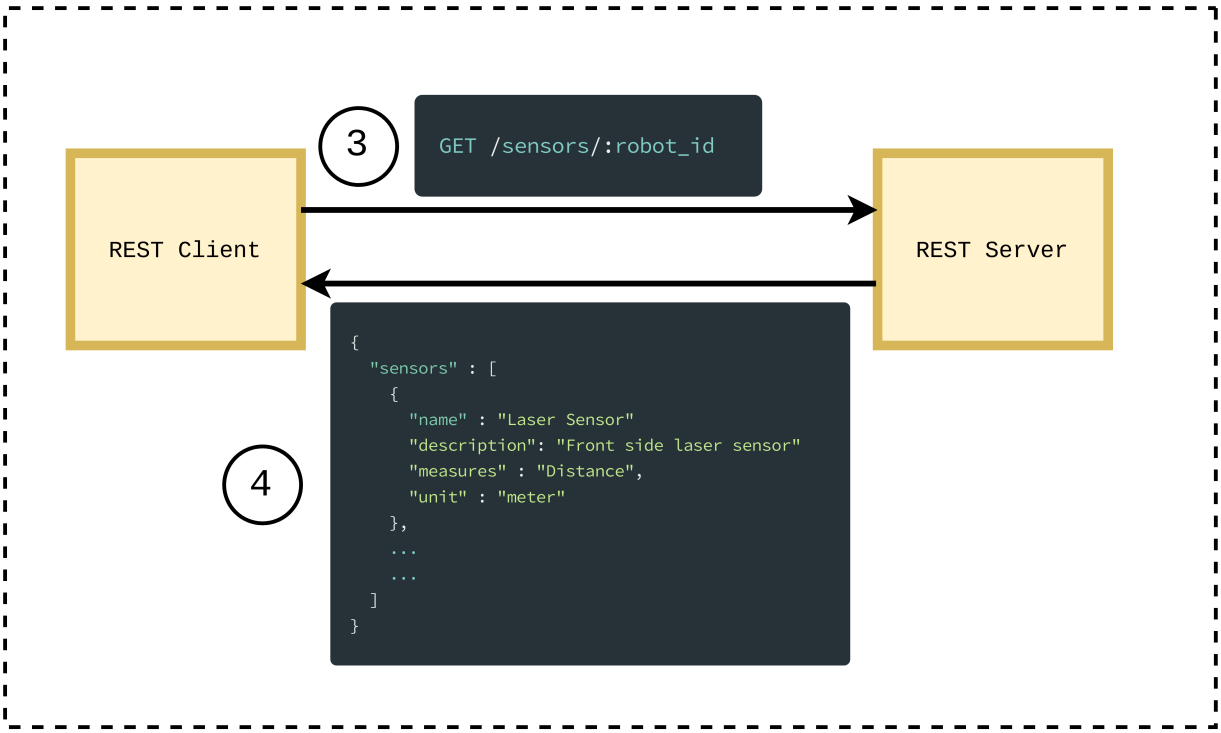


Figure 5.23: Multiple round trips in REST API (b)

To get all 10 robots information, the client has to make a series of individual GET requests to fetch all robots information and later client makes another round of requests to get the sensor data. This consumes too much network resources by making multiple roundtrips.

On the other hand, figure 5.24 shows that GraphQL client attaches the required fields and their additional related fields in a single request to fetch the complete information in a single roundtrip which reduces the usage of network resources

tremendously.

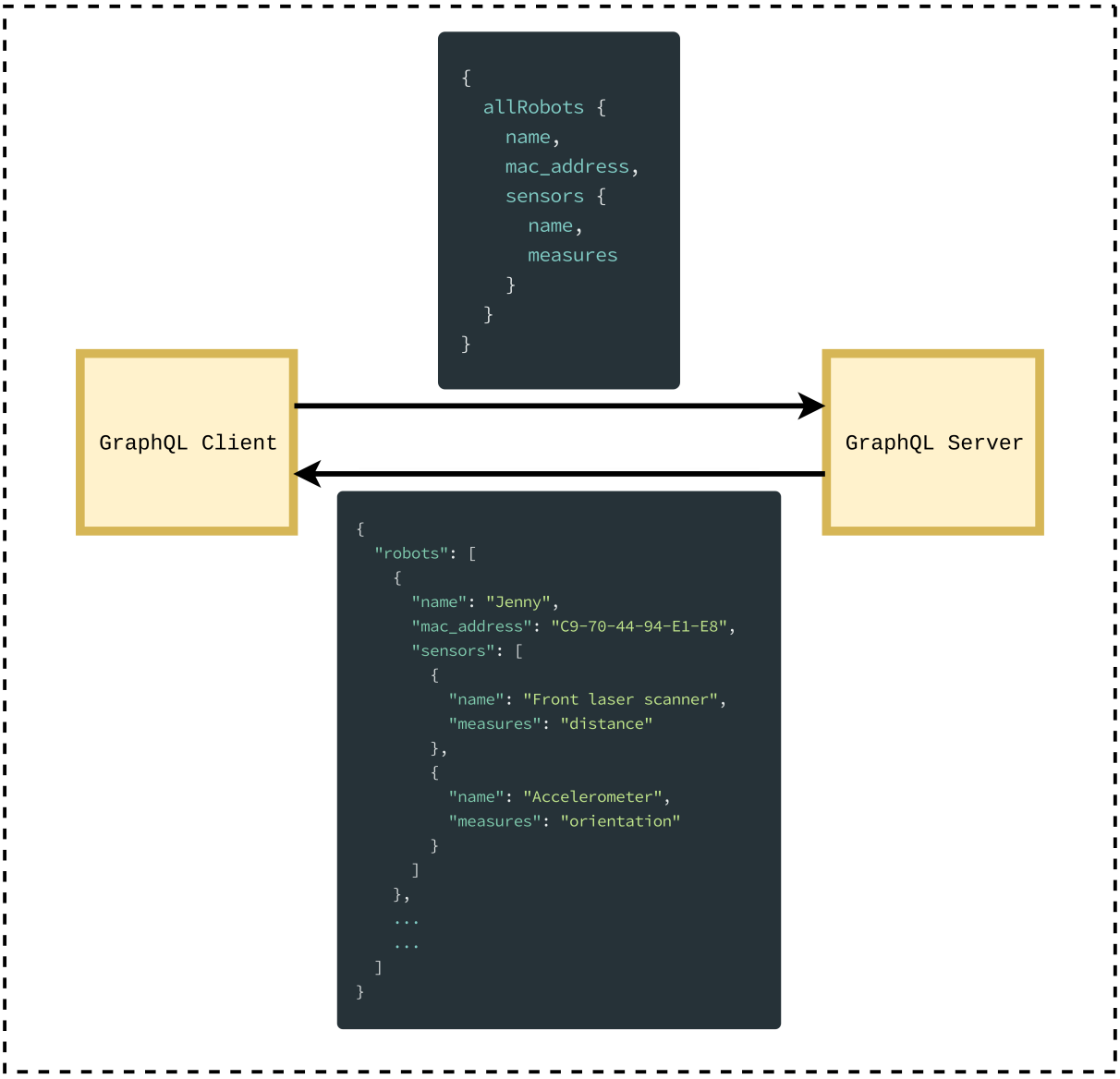


Figure 5.24: Single roundtrip in GraphQL

5.3.2.2 Declarative

The client decides the fields that should be available in the query response. GraphQL doesn't give less or more than what the client asks for. Declarative approach solves the over fetching issue in REST API. Also, we can say that GraphQL follows under fetching approach.

Let's consider an example to see the variance between these two approaches. Imagine we have a database that contains a list of robot information such as name, description, mac_address, context, manufacturer information, type, weight, etc. and we would like to query only for name and mac_address of all the robots.

In the figure 5.25, the REST client requests to the server to get all the robots and server returns with a list of robots as a response but each robot object in the response consists of every information belongs to it. Now the client has to handpick the only required fields from the response. Adding $include=name,mac_address queries along with the GET request might solve this issue. However, this is overburden in terms of often rewriting code in the server for every change from the client.

GraphQL solves this over fetching issue with zero configuration in the server side. In the below figure 5.26, GraphQL client request for name and mac_address of all robots exclusively, and GraphQL server automagically responds a list of robots only with name and mac_address. In the end, it saves time on rewriting code on the server, and most importantly reduces the load on the network layer.

Figure 5.25: Fetching all robot details via REST API includes all the attributes related to each robot.

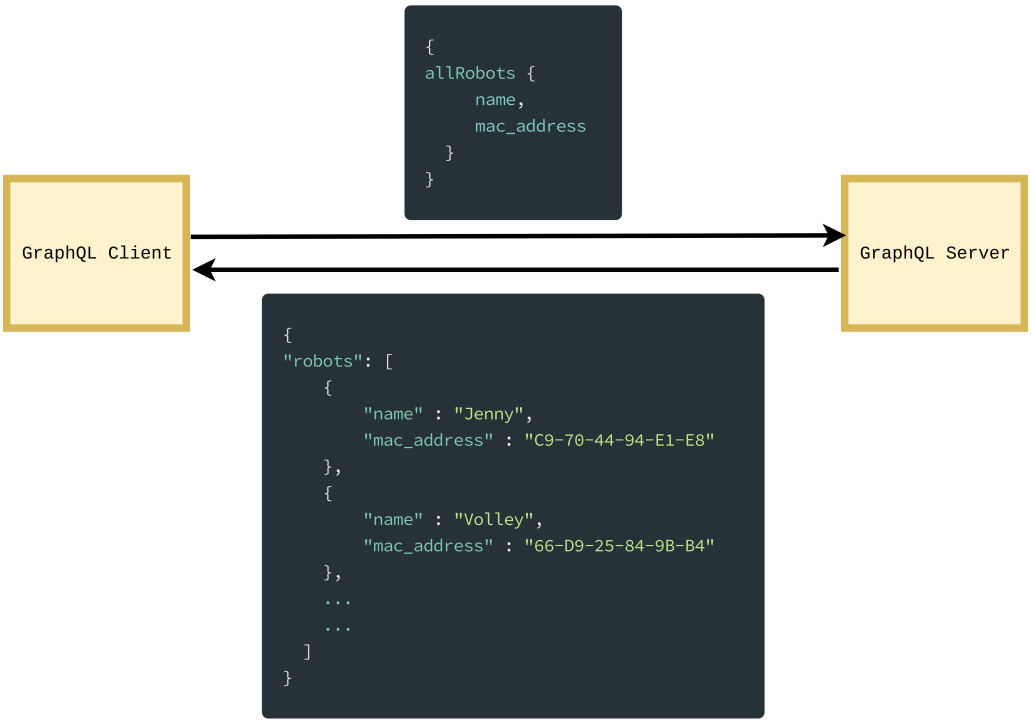


Figure 5.26: Fetching only required robot details with GraphQL queries

5.3.2.3 Single endpoint

GraphQL exposes a single endpoint for all available services from the backend. This overcomes the REST API multiple routes and each exposes a single resource. For example, consider we need to get a list of robots and sensors in two different network calls. In typical REST API architecture, we would have two separate routes as shown below.

"/robots" & "/sensors"

In GraphQL world, we define only one route, and we send the queries to the server over the single URL.

5.3.2.4 Strongly typed validation

Strong type system in GraphQL validates the incoming query for data types and prevent prematurely even before sending the queries to the database. Also, it makes sure that the client sends the right data and also the client can expect the data in the same way.

5.3.2.5 Caching

Usually, the browser caches the responses for different routes, and if the client makes a similar request, the browser gets the data locally. However, it is not possible directly since GraphQL uses single route endpoint. Without caching, GraphQL would be inefficient, but there are other libraries in the community which handles the caching in the client side. The popular libraries are Apollo and FlacheQL, and they store the requests and responses in the simple local storage in the form of a normalized map [18].

5.3.2.6 Multiple data sources

GraphQL creates an abstraction layer between clients and the databases used in the backend as shown in the figure 5.27. This feature allows the service provides to use any number of data sources and GraphQL fetches the relevant data from all data sources and returns the required fields to the client side.

This is one of the primary reason why we considered GraphQL as a base to our mediator system. Because it is always not sure how the robots store the sensor data and which database is used. In a typical scenario, multi robots might use various databases to store similar sensor entities.

5.3.3 Falcor

Falcor is a framework similar to GraphQL developed by Netflix for their internal use, and later they made it available as open source. Unlike GraphQL, Falcor doesn't emphasize users to provide a schema. Instead, Falcor generates a schema from the given data as a single Virtual JSON object [20]. It uses "One Model Everywhere" [20] policy to model all the backend data into a single JSON file.

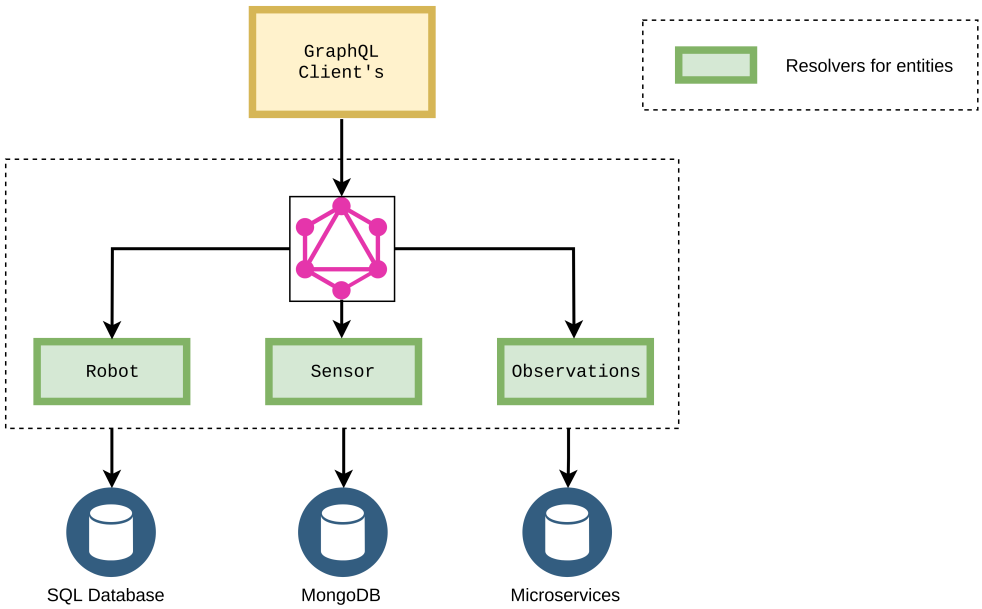


Figure 5.27: Connecting multiple data sources with GraphQL

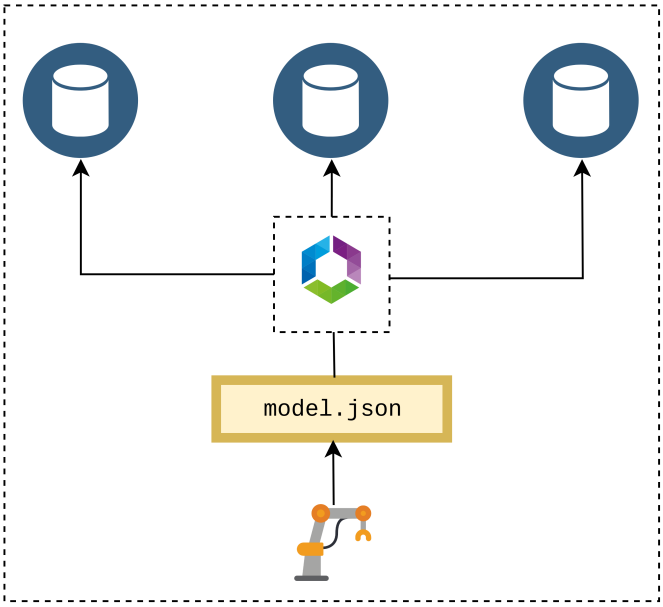


Figure 5.28: Falcor One Model Everywhere design [20]

Falcor and GraphQL share many similarities like data demand driven architecture, single endpoint, single roundtrip, and under fetching.

However, what makes Falcor differ from GraphQL and why limited people are using the Falcor framework compared to GraphQL. The major reasons are,

- Falcor is not a query language like GraphQL. Alternately, Falcor uses a javascript style of accessing attributes from the objects to select the necessary fields in the response.
- It works based on a single colossal JSON model.
- Falcor doesn't follow a strong type validation out of the box, and the users may achieve this functionality manually with one of the popular type systems like JSON schema or typescript.
- By default, GraphQL provides schema introspection which allows various tools to utilize the representation of schema beforehand. Though, Falcor says, "if you know your data, you know your API" [17] which is not true always since more often we don't know what fields and their names available in our data.
- GraphQL have a wide range of server implementation in C# / .NET, Clojure, Elixir, Erlang, Go, Groovy, Java, JavaScript, PHP, Python, Scala, Ruby [16]. Unfortunately, Falcor has their implementations in JavaScript, Java and C# / .NET. This hardly gives options for the developers to choose Falcor.
- It is possible in GraphQL to pass arguments to queries which allow the user to do advanced operations in the backend. However currently, Falcor doesn't support this feature.

Falcor also has few advantages over GraphQL.

- Easy learning curve.
- Simple to adapt with small range projects.
- Caching and query merging.
- Batching and Deduplication [19].

5.3.4 Conclusion

At the end of this research work, we develop a mediator component to interact with diverse databases running in multi-robot systems. For that, we need to decide to choose between GraphQL and Falcor as a base for the mediator component since they have the capabilities to fetch data from various data sources and return a single response over a single endpoint. However, for choosing one we have made a detailed comparison in the above sections based on their abilities and features. Most of the times, GraphQL supersede Falcor in case of flexibility, scalability, introspection, type validation, and more language support. So we conclude to use GraphQL as a base for our mediator component.

6

Implementation

This chapter describes the implementation of the mediator component, tools involved in the mediator and their purpose, and the strategy followed to collect sensor observation data from different robots over the mediator.

6.1 Mediator design process

The mediator is a sophisticated software component which comprises various sub-components. Instead of directly start developing the complex mediator component we decided to find and set the standards for the development process. Many standards and approaches have been defined in the software engineering domain that should be followed by developers to achieve a stable development process and at the end finish the project with a successful working product. Since this research work implements a software component, we decided to take up two well-known software development models called Feature driven development from Agile and Component Assembly Model. It is a heterogeneous development model as we take the useful features from two different models.

Feature driven development was first introduced by the book "Java Modeling in Color with UML" [1] and first used for a huge bank project. The core concept of FDD is initially developing an overall model and define all possible required features as a list. The overall model for this research work is, a mediator component operates between multiple database instances which runs in a centralized server or in the robot itself. Now the possible feature list for our mediator component is,

- Finding the entities and define their attributes.
- Finding a suitable data structure which supports context.
- Schema registration for a new task, robots, and sensors.
- Creation of observation buckets and update GraphQL type definitions.
- Update GraphQL mutations and queries.

Once the overall model and feature list are defined, we decompose the feature into smaller reusable components. These components are developed individually and combined later back to a single feature on the basis of Component Assembly model. For each feature in the list, a planning schedule is assigned to keep track of time and finish developing the feature on time. Before start implementing a feature, proper design is made to avoid changes in the mediator component in the future. After preparing the plan and design, the next step is building smaller components which form the individual feature. The complete process is repeated until the final mediator component meets the proposed mediator requirements.

After each release, the mediator is cross verified with the user story and tested on real use cases which are described in the later section. Feedbacks are received from the user and improved the mediator stability and reliability on each version which is released on every week.

6.2 Architecture

This section gives an overview of the mediator architecture and the components involved in it. The overall architecture is divided into three major sections, Producer or Consumer, Mediator, Data sources as shown in figure 6.1.

6.2.1 Producer/Consumer

The leftmost section in figure 6.1 includes any robots or tools which consumes the data from data sources or produces data to the data sources via the mediator component. In the research work, we are not investing the types of consumer/producer involved in the process since the mediator component is being developed to solve general heterogeneous data sources problem with a global audience in mind.

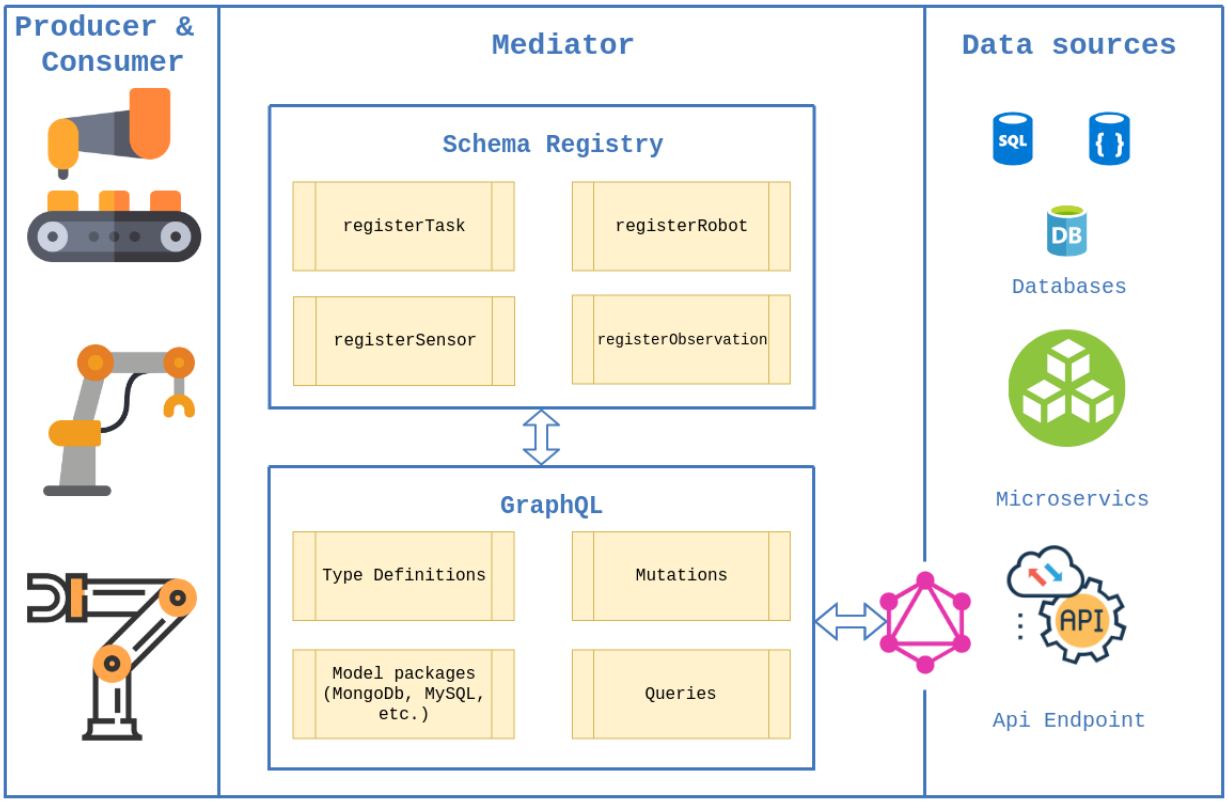


Figure 6.1: Mediator system architecture (Images used in architecture are cited here [11] , [25])

6.2.2 Mediator

The essential section in the architecture is the mediator component considering it manages most of the interactions between consumer/producer and data sources. Mediator component again divided into subcomponents called Schema Registry and GraphQL mediation. These two components are developed as individually isolated docker containers for modular and portability reasons. Each docker container runs on its own configuration and a small alpine Linux as the base image. The alpine base image is chosen for the following reasons,

1. Usually, Alpine flavored images are small in size which in turn shrinks the container size. For example, Fedora version 5 base image size is 231MB, CentOS

7 base image size is 193MB, Ubuntu 16.04 base image size is 118MB and Alpine 3.6 base image size is only 3.98 MB [21]. We can see that the Alpine base image is more than 90% smaller than other flavors.

2. Contain only the basic functionalities without GUI components.
3. Faster container creation and boot time. For example, Debian based container creation took around 28 seconds and Alpine based container creation took only 5.6 seconds [21]. Note that, this comparison is performed without cache.
4. Safe and secure. Fewer risks of attack if there are less number of packages and libraries available in the base system. A few years ago, a severe vulnerability was exploited called "ShellShock" which give access to hackers to execute bash commands on the server over an HTTP request. Alpine is safe from "ShellShock" attack because it does not have bash installed by default [21].

6.2.2.1 Schema registry

Schema registry is the entry point for registering the tasks, robots, and sensors as a relational model. This step is mandatory to let the mediator component know the relationship between the entities and also the exact structure of sensor data. This structure of sensor data is represented in the form of JSON schema which is used for GraphQL schema transformation which is discussed in registering observation section.

Schema registry itself is an individual docker container which runs "express[1]" server to provide the schema registration service. Express.js is a modular light-weight web application framework developed to run with Node.js platform. In the schema registry, the express server opens an endpoint for the entity and schema registration.

Entity and schema registration is a sequential step by step process to store the task, robot and sensor registration, and creating new observation buckets in the real database. All the entities and their attributes are defined as follows,

Task- A task defines the activity that needs to be performed by the robot.

Robot - A robot entity holds the digital twin of the robot.

Sensor - A sensor entity defines the details of the sensor used in the robot.

[1] https://github.com/expressjs/express

Attributes	**Definitions**
id	Unique UUID for each task
name	Name of the task
@context	Context for the attributes
creator	Name or id of the person who created this task
creationTime	Time of the task creation
startTime	Time of the task that started
endTime	Time of the task that ended

Table 6.1: Task entity attributes description

Attributes	**Definitions**
id	Unique UUID for each robot
name	Name of the robot
@context	Context for the attributes
type	Type of the robot
id	Unique identified for the robot
macAddress	Mac address of the robot

Table 6.2: Robot entity attributes description

Attributes	**Definitions**
id	Unique UUID for each sensor
name	Name of the sensor
@context	Context for the attributes
type	Type of the sensor
description	Short description of the sensor
measures	What physical characteristic that the sensor measures from the environment
valueSchema	A JSON Schema represents the structure of the data that this sensor will generate.
unit	Unit for the measured observation
meta	Meta attribute allow users to add any additional information about the sensor which is missing in the given attribute list.

Table 6.3: Sensor entity attributes description

TaskRobotSensor - TaskRobotSensor entity doesn't store any information about other entities; instead it stores only the relationship between a task, robot, and senor. We introduce this pivot relation because we need a way to relate entities in non-relational databases too. This type of table is called a pivot table. Pivot tables can be used to represent the relationships between different entities stores in other tables/collections. Additionally, pivot tables can have their own attributes, and this TaskRobotSensor pivot entity has startTime and endTime as additional properties to identify at what time a robot has joined the task, or when a sensor is attached with this robot because involved sensors and robots can change over time.

Attributes	**Definitions**
id	Unique UUID for each TaskRobotSensor relation
task	Unique task UUID
robot	Unique robot UUID
sensor	Unique sensor UUID
startTime	Time shows when this relation has been made
endTime	Time shows when this relation has been ended (if this value is not provided, then the Task endTime will be considered as TaskRobot-Sensor endTime)

Table 6.4: TaskRobotSensor entity attributes description

Observation - Observation is a generic entity, and it can be extended to any sensor that generates data. For each new sensor entity, a new observation bucket will be created in the name senor. The naming convention of each bucket should be unique to avoid conflicts, and the name is derived from the sensor name, and "Observation" keyword will be attached at the end. For example, consider we create a new observation bucket in MongoDB for the sensor "command velocity". Schema registry component will create a new collection in MongoDB in the name of "CommandVelocityObservation". Multiple robots are allowed to have observation buckets with the same name, but a single robot should not have duplicate observation buckets to avoid conflicts (By default, databases do not allow users to create tables/collections with the duplicate name). Each observation contains the real observed data from the sensor and also the relations with the task, robot, and sensor.

The attributes described in the entities table 6.1, 6.2, 6.3, 6.4 can be modified based on the domain needs.

Attributes	Definitions
id	Unique UUID for each observation
name	Name of the observation
@context	Context for the attributes
featureOfInterest	Tells the important feature from the observed value
value	Data generated by the sensor
task	Unique task UUID
robot	Unique robot UUID
sensor	Unique sensor UUID
phenomenonTime	Time of observation created by the sensor
resultTime	Time of observation stored in the database

Table 6.5: Generic observation entity attributes description

6.2.2.2 Schema registration workflow

Figure 6.2 shows the broad outline of schema registration workflow and also the essential subtasks such as registerTask, registerRobot, and registerSensor. Registering schema with the mediator is a necessary process because initially, the mediator component has no prior assumptions on the used robots/sensors. Schema registration docker component exposes an endpoint "/schema-registry" that allows users or robots to send a complete schema registration over HTTP POST request and the schema configuration A contains the task information, array of robot objects and each robot object consists of an array of sensor object. After the starting point in the flow chart, the *parse schema config* function starts parsing the given schema configuration and checks the validity of the configuration. If the configuration is valid, then execution moves to the next step and checks for a task configuration. If task config exists, then it proceeds to the very first subtask called registerTask and all the subtasks are outlined in section 6.2.2.3. If there is no task configuration or the task registration is unsuccessful, then the execution will be aborted. After the successful task registration, the process starts iterating the robots array.

For each robot in the array, the system checks whether it is a new robot or existing one. For existing robots user gives the UUID of the robot, and for new robots, the system expects complete information about the robot as mentioned in the robot entity section. If the robot information is available, then the operation proceeds to the registerRobot subtask. If the subtask is not successful, then the

system will abort the execution, else it starts iterates the sensor list which is attached to the robot information. Like new robot check, the system checks whether the given sensor contains an already registered UUID or new sensor information. If it is a new sensor, then the system proceeds to the last subtask called registerSensor.

In any case, after entity registration failure, the system will abort the schema registration execution. However, this workflow can be configured in such a way that, ignore the registration failures and continue the execution until the end of the schema config. After each sensor registration, the system verifies whether the current sensor is the last item in the sensors array or not. If not, then the loop continues until the last sensor in the array, else it checks whether the current robot is the last item in the robots array. If not, then the loop continues until the last robot in the array. If the system finishes the last robot in the array, then the system stops the execution with a success flag. Later, an HTTP response will be sent back to the user/robot with the status of the schema registration and additional messages if required.

6.2.2.3 Workflow of subtasks in schema registration

Major schema registration consists of three subtasks called registerTask, registerRobot, and registerSensor. registerTask 6.3a and registerRobot 6.3b subtasks execution follows a similar workflow but for two different entities. At first, the system checks for the valid task/robot configuration and DB configuration respectively. If it fails, then the system aborts the execution, else system creates task/robot record in the appropriate databases. But, the system should identify the specific databases and additional information beforehand. This database entity mapping is done with the help of media config file, and it is explained in section 6.2.2.5. The last subtask called registerSensors 6.3c follows the similar workflow like registerTask and registerRobot, however, after each successful sensor record creation, the system should carry out another major subtask called "Create observation bucket" which is explained in section 6.2.2.4.

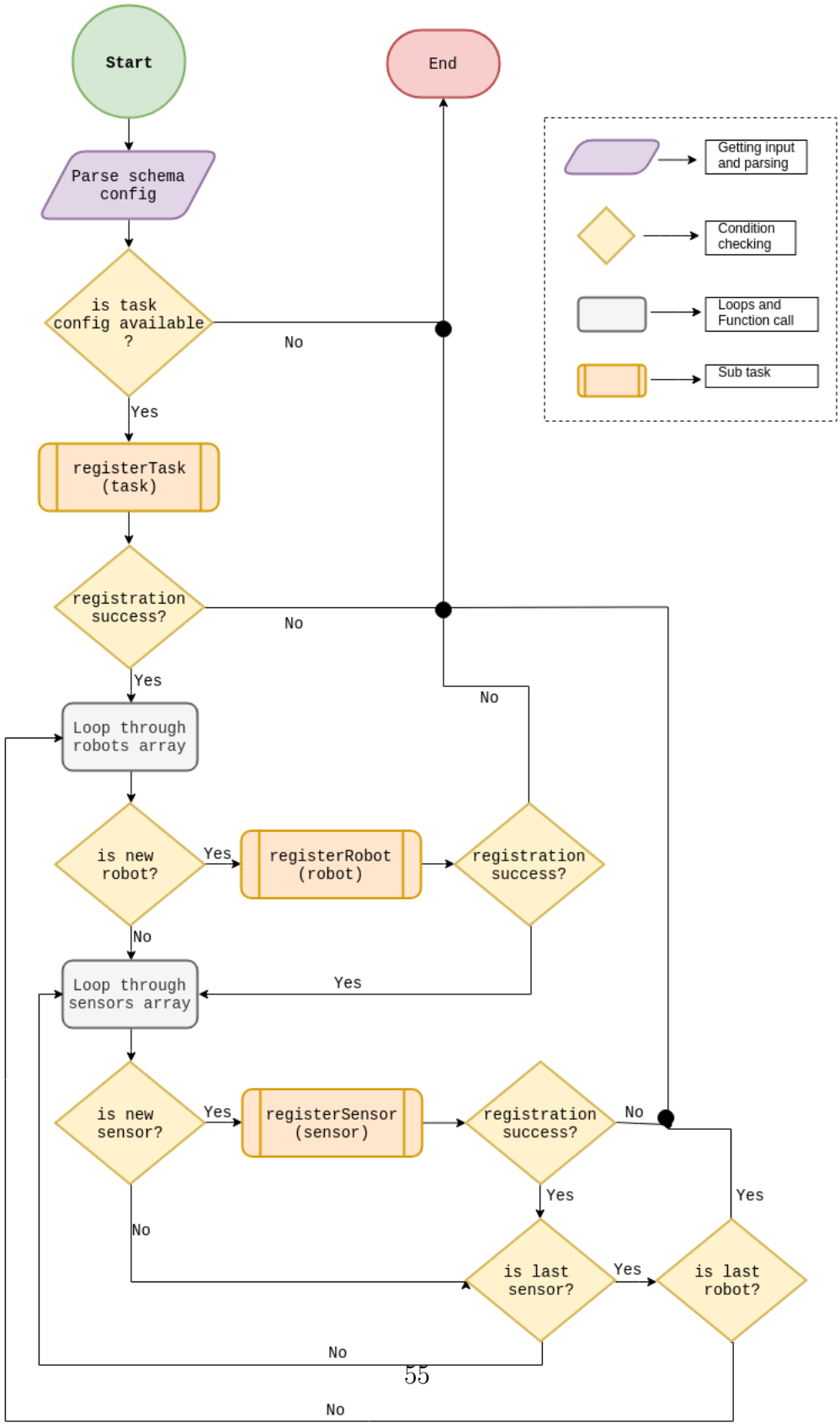
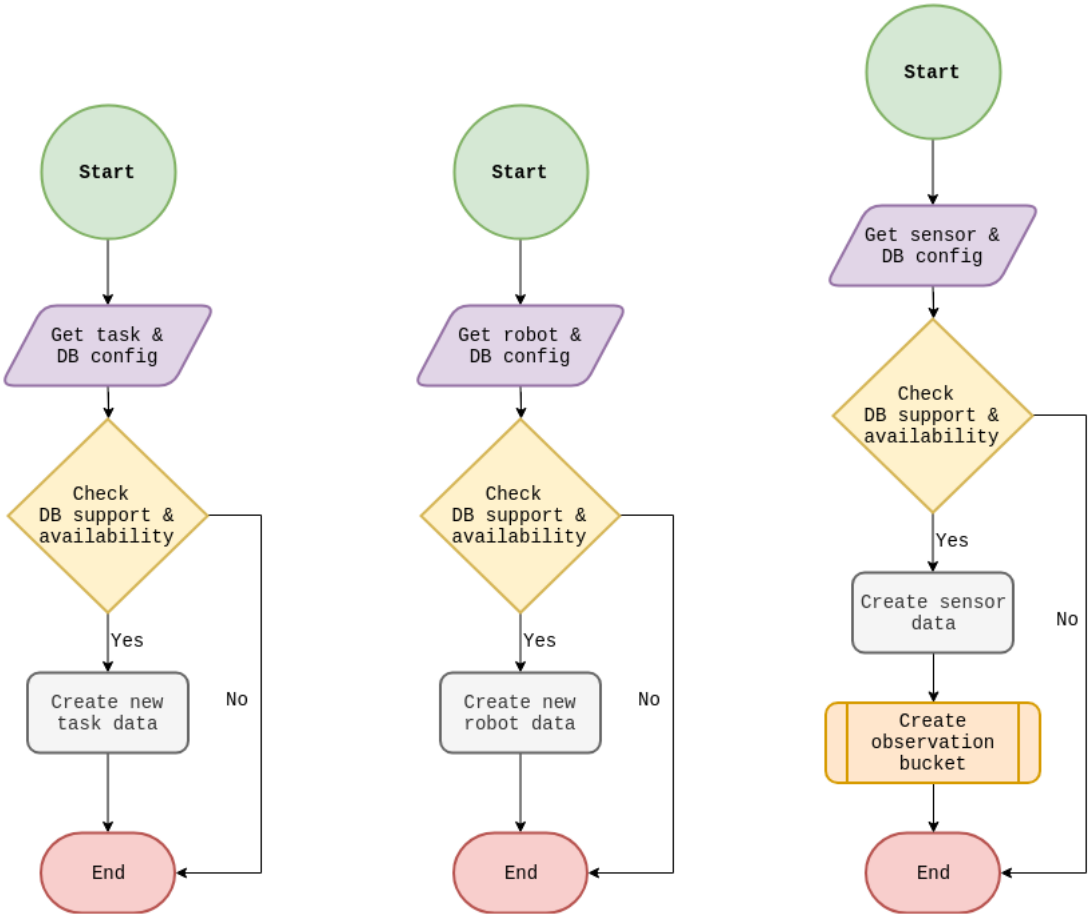


Figure 6.2: Detailed schema registration work flow



(a) Workflow that represents creation of new task entity during schema registration

(b) Workflow that represents creation of new robot entity during schema registration

(c) Workflow that represents creation of new sensor entity during schema registration

Figure 6.3: Decomposition of major entities creation work flow

6.2.2.4 Observation bucket creation workflow

This module includes a series of actions that need to be performed to make sure the new observation buckets are available after the creation of the new sensor, in all possible databases/data-sources which is defined in mediator config file. The naming convention for new buckets is already discussed in the "Observation" topic under section 6.2.2.1. Once after successfully created the buckets, schema registry should update the changes to GraphQL mediation component (second docker container). Three things need to be updated in the GraphQL mediation component as follows,

1. GraphQL type definition - GraphQL frameworks identify all the data attribute types via type definition file. So we need to update this type definition with the newly created sensor output structure. Each sensor object consists of an attribute called "valueSchema" which shows how the observation data will look like from this specific sensor. This schema is nothing but a JSON Schema file with all attribute types and constraints imposed on them. This JSON Schema to GraphQL schema conversion is done by a transformer module which is explained in section 6.2.2.6.
2. GraphQL queries - GraphQL queries consist of all possible queries that can be executed on the GraphQL mediation server. Here, the system should update new queries to fetch the new observation data from various buckets.
3. GraphQL mutations - GraphQL mutation helps users to create new records in the database. But, it is not mandatory to create records via the GraphQL mediation component, because robots can generate the data and store them directly to their local database instances. Later the person who wants to make fault diagnosis or to debug, they can configure the DB's with the mediator and fetch data via GraphQL queries.

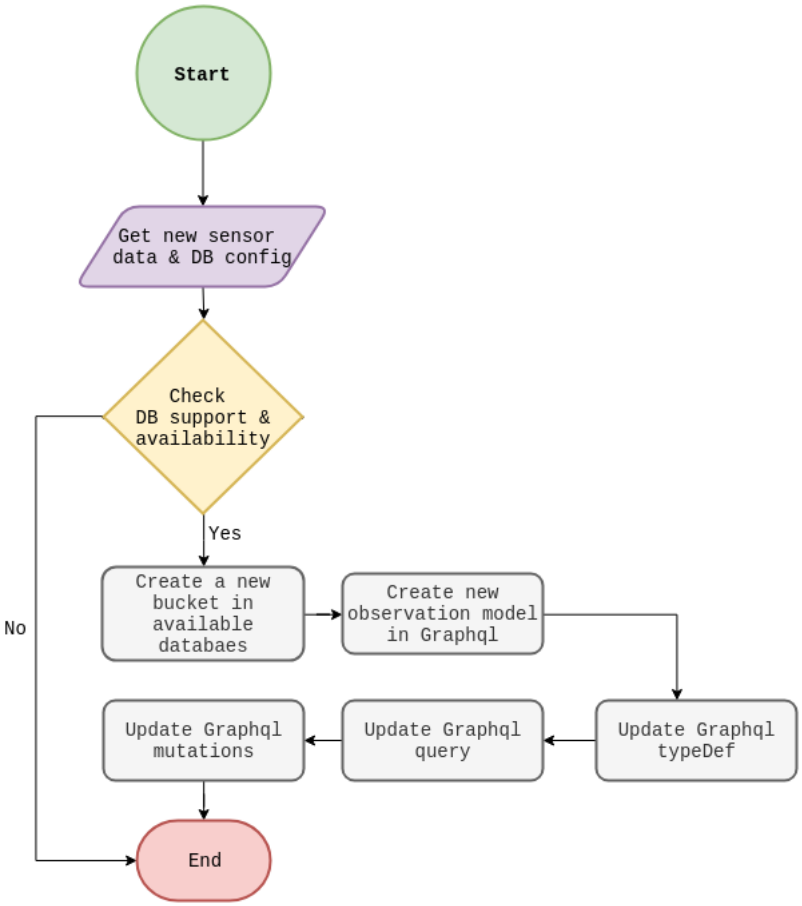


Figure 6.4: Observation bucket creation work flow

6.2.2.5 Entity database mapping

For creating new tasks, robots, and sensors data during schema registration, schema registry should have an idea where to create these records. We propose an entity database mapping strategy to identify the databases for each entity. To make it more flexible, we assign each entity to a database name, and the corresponding database configuration is described in the same config file under "db" attribute which is shown in figure 6.5. The "db" attribute contains a list of db configuration and the attributes in each db object is explained in table 6.6.

In the given example 6.5, under "entityDBMapping" attribute, each entity is

Attributes	Definitions
name	Unique name for each database and this name is used to identify the database configuration in the mediator system internally. Don't confuse this attribute with "dbName".
type	Type of the database. Currently, the mediator system supports mongodb and mysql.
url	URL to access the database.
dbName	The actual name of the database.
userName	Username to access the database (not mandatory).
password	Password to access the database (not mandatory).

Table 6.6: Single db object attributes description

mapped to a name of the database which is described under "db" attribute. The "observations" entity consists of an array of database names, because the sensor generated data may be available in multiple robots/datasources. With the help of this mapping, the schema registry gets the information about where to create new records. This same mediator config file is also shared with GraphQL mediation component.

```
{
    "db": [
        {
            "name": "mongodb_blackboxone", "type": "mongodb", "url": "localhost:27018",
            "dbName": "test", "userName": "", "password": ""
        },
        {
            "name": "mysql_central_server", "type": "mysql", "url": "localhost:3308",
            "dbName": "db", "userName": "root", "password": "password"
        }
    ],
    "entityDBMapping" : {
        "task": "mysql_central_server",
        "robot": "mysql_central_server",
        "sensor": "mysql_central_server",
        "taskrobotsensor": "mysql_central_server",
        "observations" : [
            "mongodb_blackboxone",
            "mysql_central_server"
        ]
    },
    "format": "json-ld",
    "graphqli": true,
    "port": 3085,
    "url": "http://localhost"
}
```

Figure 6.5: Entity database mapping representation in mediator config file

6.2.2.6 JSON Schema to GraphQL Schema transformation

The mediator component is using GraphQL framework as a base to utilize the features 5.3 offered by GraphQL out of the box. GraphQL needs a type definition document to work on even before starting the server. Through schema registration, user/robot provide a JSON Schema which represents how the sensor data would look like in the observation bucket. JSON Schema 2.5 is a well known schematic representation for data, so we decided to use this as a generic data schema format to represent the sensor observation structure. Later the JSON Schema to GraphQL transformation module in "Schema registry" component translates the JSON Schema to GraphQL type definition through a series of steps as shown in the figure 6.6. There are three steps involved in the transformation as follows,

1. Get the JSON Schema from the valueSchema attribute in sensor object and pass it to the next step.
2. In this step, we use an open source library "djvi [2]" to generate the model prototype from the JSON Schema. This model prototype resembles precisely the same as the observation that will be generated by the sensor in the future.
3. Finally, based on this model prototype, the schema registry system creates an equivalent GraphQL schema (type definition), and the new schema is appended to the current GraphQL type definition file.

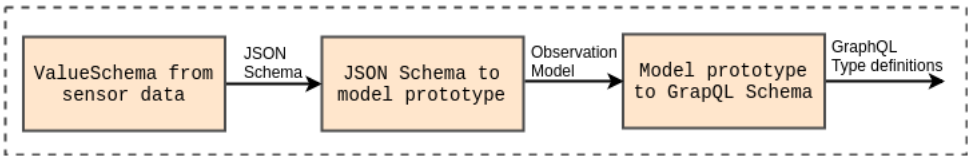


Figure 6.6: Process of JSON Schema to GraphQL Schema (type definition) conversion

In the end, GraphQL will consume the new updated file, and users/robots can write/read new sensor observations from different datasources.

[2] https://github.com/korzio/djvi

6.2.2.7 GraphQL mediation

Mediation component is a separate docker container which is built based on express server and Apollo GraphQL wrapper. Apollo GraphQL wrapper is implemented based on facebook GraphQL specification and designed their system as a wrapper to run on top of the express server. To run the Apollo GraphQL server, it requires four mandatory modules such as,

1. Type definitions - It consists of data type definition for all the entities which users are going to query or write into the database. In the context of our research work, this module consists of schemas for Task, Robot, Sensor, TaskRobotSensor and all type of Observations. Additionally, it may have the queries and mutations outline but not their definition.
2. Mutations - This module consists of required functions to receive the incoming data and write/update in the database.
3. Queries - This module includes necessary services to fetch data from the database.
4. Model packages - This is a custom module which comprises of model definitions of each entity for all the supported databases. Currently GraphQL mediation component supports MongoDB and MySQL. The system uses two open-source libraries called "mongoose [3]" and "sequelize [4]" to make the connection and communication between the GraphQL mediation component and the respective databases. Both modules are designed to follow ORM (Object Relation Mapping) pattern to let developers write their code in a class structure which is convenient and easy to maintain the code base. The mongoose module targets only MongoDB and sequelize can be used to interact with Postgres, MySQL, MariaDB, SQLite, and Microsoft SQL Server.

6.2.2.8 Connection pool management

Both the schema registry and GraphQL mediation components initiate their database connection pool before the server starts. The system creates different database

[3] https://github.com/Automattic/mongoose
[4] https://github.com/sequelize/sequelize

connections based on the "db" configurations mentioned in the mediator config file 6.5 . And this database connection pool is shared across the component such that other services like queries and mutations can use it to read or write data into databases.

6.2.3 Data sources

The rightmost block in the architecture 6.1 represents the data sources. A data source can be any databases, microservices, or an API endpoint. GraphQL component serves as a bridge between the robot and data sources. In the current work, we used only MongoDB and MySQL as data sources. However, the mediator can be extended to support other databases with little modifications in the GraphQL Queries and Mutations module.

6.2.4 Summary

In the above sections, the three major subdivisions of the mediator architecture and their responsibilities are explained elaborately. And the next section evaluates the current mediator design with the requirements and existing black box data storage design.

7

Experiment

To verify the workflow of the mediator component, we set up different docker containers with individual database instances. Since the current design support only MongoDB and SQL databases, we built the containers based on these databases. The scenarios are formulated from the current ROPOD black box system. During the robot experimentation, data loggers in black box continuously record various sensor information in its MongoDB. Later, a script dumps all the data into a centralized server. For our experimentation, we collected the dumps from ROPOD team and analyzed what sensor data is being recorded and how it is stored. We found that the sensor data from ROS topics are flattened and stored in the database under different collections. From the sample dumps, we selected four data collections such as command velocity, joint states, odometer, and scan front. These four variants are chosen based on their data types and complex nested object structure in the data. Note that not all four has a nested structure, only joint states and Scan front values hold nested objects. With this mixture of samples, we wanted to investigate the significant features of the mediator given below,

- Schema registration workflow.
- Translation of JSON schema that belongs to these sensor data to GraphQL schema.
- Dynamic schema update in GraphQL component.
- Multiple database connection pool management.

- Declarative data fetching the sensor data from different database docker containers using single GraphQL query language.

For the schema registration process, we prepared a sample schema config which defines a sample robot task conducted in an indoor environment with two robots, and each robot has two sensors. Odometer and Command velocity sensors are attached with robot one and Joint states and Scan front sensors are associated with robot two. Also, the robot information such as name, macAddress, and description are added to each robot object in the configuration. This schema config is sent to the schema registry component over HTTP POST request. Now schema registry creates the buckets if not available to store the sensor observations in the databases mentioned in the mediator configuration. In our mediator context, bucket means a space to store data from individual sensors. If the database is MySQL, then a "table" represented as a bucket, or if the database in MongoDB then a "collection" represented as a bucket. At the end of schema registration, the system transforms the JSON schema of each sensor to a GraphQL schema, and this transformation process is explained in section 6.2.2.6.

After successful schema registration and buckets creation in databases, robots will receive the unique UUID that is created for the robot and sensor. Later robots add this UUID along with the observed values before storing them in the databases.

For quicker experimentation, we filled our buckets in all databases with data dumps which we collected from ROPOD team. Consider each of our docker containers as a black-box/robot and each running with its own database. The database could be either MongoDB or MySQL. We executed various GraphQL queries against the mediator to fetch sensor data distributed in multiple systems. The relevant queries which we used are described below,

- Get all robots involved in a specific task. This variant also includes fetching all sensors which are used in a particular robot, all tasks that the robots/sensors engaged.
- Get all observations of type command velocity which returns the observed command velocity values from all black-boxes/robots.
- On-demand context fetching for all the query via the mediator.

- Declarative data fetching queries to fetch only the required fields from the query response.

All the queries executed for the experimentation is available in our source repository, and the links are given in appendix A. The evaluation of the mediator system is presented in the next section.

8

Evaluation

In this section, we evaluate the proposed mediator architecture design based on whether it fulfills the requirements and solves the problem statement discussed in the section 4.

8.1 Evaluation through requirements

Context based data model - As we discussed in section 4, in a multi-robot scenario the data generated by different robot sensors should be interpreted by other robots in the group. But the system in the robots might be developed by different developers and may use different attribute names. So there is a chance for misinterpretation of data during calculation also humans understand the context by seeing the data, but it is difficult for the machines. Our proposed system supports delivering contexts along with the response data. To achieve this, we created our own list of general vocab collection in terms of robot applications with all the meanings predefined in it. Also, the proposed data models relationship is improved over time to support secure data retrieval from the multi-databases with minimal effort through GraphQL queries.

8.1.1 Generic query language

Our mediator architecture is designed with GraphQL frameworks as a base, and the explanation for why we adopted this framework is discussed in section 5.3.4.

GraphQL offers it's own query language to write/read data, and the learning curve is fast to understand this query language. So users/robots may use this generic query language to interact with datasources regardless of what type of database is running in the system.

8.1.2 Type validation

The current system is strongly bounded with types to avoid the problems or system failures during any necessary calculations in the robot to decide an action to perform. So the system doesn't allow to store any data with a type mismatch.

8.1.3 Scalability

One can configure multiple databases with the current mediator with a single change in the mediator config file.

8.1.4 Configurable data format support

By default, the mediator system returns the response in JSON-LD format. But some legacy systems or the tool which consumes the data might need the response data in JSON or RDF format. In this case, consumers can directly transform JSON-LD data to other context based format such as RDF using external tools. However, one can set the response format in the mediator config and implement their logic for transformation in the mediator.

8.1.5 Graphical User Interface

Apollo GraphQL server is packed with an efficient GUI system to make queries or mutations on the database. This GUI system can be exposed over "/graphiql" endpoint on demand. This tool is beneficial for users to visually debug the sensor generated data or mock a dummy set of data in the database. Also, this tool offers schema introspection so that the users do not require to check the schema in the database whenever they query.

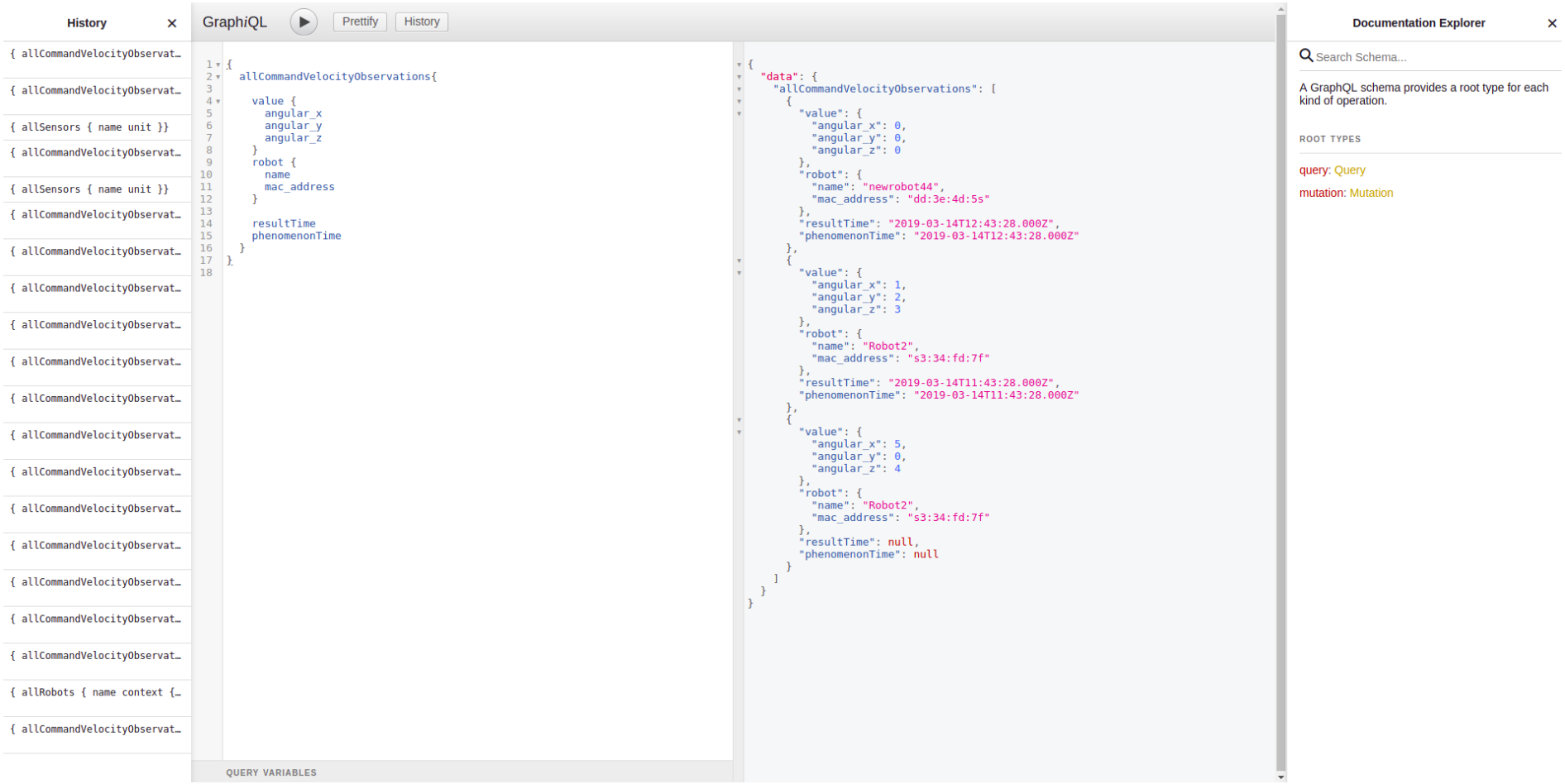


Figure 8.1: GraphiQL tool to mutate or query data visually from a browser

Figure 8.1 shows the complete sections in the graphiql web-based user interface. There are four sections in the application. The first section shows a list of previously executed GraphQL queries, and it saves users time by not writing the same queries repeatedly. Next section gives space for the users to orchestrate GraphQL queries and it provides autocompletion feature to select the attributes under the specified data entity. The third section shows the result which is returned from GraphQL server. The last section provides an option for users to do introspection on schemas for queries and mutations.

8.2 Evaluation against the existing system

Existing black box system flattens the sensor generated data and stores them directly in its local MongoDB instance. And the problems in dumping the sensor data without proper modeling them is already discussed in section 4. Also, the black box exposes a data query_interface to apply filters and query data from the dump. However, the person who wants to debug the logs from multiple robots, they have to switch the database in the configuration before working on the query interface system. Our system design solves all these problems. For example, with the help of the current mediator system, one can configure multiple database instances and apply a single query to fetch data from all of them. Also, users can add filters and choose the data only from selected robots. The current system also provides a well-defined data model to connect specific task with robots and sensors which are involved in that task. In the end, by giving the JSON-LD context in the response solves the interoperability issue from existing black box system.

9

Conclusion

9.1 Summary

The primary objective of this thesis work is developing a mediator component to make fault diagnosis convenient with multi-robot systems. Also, this mediator component gives additional advantages such as unique query language to communicate with various databases, declarative data fetching which is useful if the robots are working under low network access areas. The complete architectural design and descriptions for each component in the mediator are explained in section 6.2. Finally, proper data modeling has been done to identify the possible entities from the real-world scenario and relationships has been made between them. A list of vocabularies[1] created to define the context for each attribute specified in the entities such as Task, Robot, Sensor, and Observations.

9.2 Limitations

The current implementation supports only MongoDB and MySQL but can be extended to additional databases. Even though mediator makes parallel requests to fetch data from multiple data source, the responses will be slower if the tables are not indexed correctly. Also we assume that mediator can reach all databases, otherwise a protocol has to be added to deal with caching and resending data/queries. Because, for making parallel requests, the current system uses JavaScript Promises array and

[1] All the vocabularies mentioned in appendix A.

tries to resolve all the promises using Promise.all() method. The limitation of using Promise.all() method is if one promise fails then the Promise.all() method will ignore the other promises and returns an error message to the user. Lacking of undoing or updating the schema registry. But this can be improved by adding additional functionalities in the schema registration component. We are not concerned with efficiency in this mediator implementation. If higher performance is required, the mediator could setup a direct stream between sensor and the local DB on that robot.

9.3 Future work

There is still room for improvements in the proposed mediator architecture. There are missing User and Location entities in the current entity eco-system to identify who initiated a specific task and what are the locations traveled by robots during the experiment. The current mediator system implementation is dynamic and configurable to accept new enhancements over time. So the robotic community should come up with a collection of new entities with vocab contexts at any time that supports robot applications.

JSON-LD provides @context system to give meaning for the heterogeneous data from different robots. But when we use GraphQL architecture as a base, heterogeneity problem is solved partially but not completely. Because GraphQL requires the structure and attributes of the data beforehand so that the user can send queries along with the attributes. But if there are two different attribute names are used for the same property, then GraphQL should know about both the name. However, to solve this we can introduce a view system to map all the differently named attributes into a single attribute name and use them in GraphQL type definition file. Object-oriented View Systems are already discussed elaborately in many research works. Most specifically, Drosten et al. [10] proposed a technique to achieve combining heterogeneous information from different sources by using object-oriented views.

Furthermore, we can analyze the possible ways to live update of new databases to the mediator system. Also, we can investigate whether it is feasible to apply the expansion algorithm on the response object on server or client side to expand the @context in the result. A comparative analysis can be carried out to benchmark the overall performance of this mediator system with existing or new mediators.

A

Appendix

As part of this research work, we have created a vocab list for entities such as Task, Robot, Sensor, and Observation and made publicly available to refer the IRI in JSON-LD @context field. This vocab list is initially created for this research use case. However, this can be modified or extended according to other project specification. Current vocab list is available in the following links,

- https://github.com/rubanraj54/master-thesis/blob/develop/reports/thesis/contexts/task.json
- https://github.com/rubanraj54/master-thesis/blob/develop/reports/thesis/contexts/robot.json
- https://github.com/rubanraj54/master-thesis/blob/develop/reports/thesis/contexts/sensor.json
- https://github.com/rubanraj54/master-thesis/blob/develop/reports/thesis/contexts/observation.json

A sample schema registration configuration is provided here https://github.com/rubanraj54/master-thesis/blob/master/mediator/schema_registry/toy_data/test.json for testing purpose. This schema contains a simple test case with two robots, and each robot has two different sensors and its JSON Schema configuration.

The queries used for experimentation are maintained in a separate folder in the repository https://github.com/rubanraj54/master-thesis/tree/master/mediator/queries .

The source code and docker files to spin-up the mediator components is available under https://github.com/rubanraj54/master-thesis/tree/develop/mediator .

References

[1] agilemodeling. Feature driven development (fdd) and agile modeling. http://agilemodeling.com/essays/fdd.htm [Online; accessed 19-March-2019].

[2] Rafi Ahmed, Philippe DeSmedt, Weimin Du, William Kent, Mohammad A. Ketabchi, Witold A Litwin, Abbas Rafii, and M-C Shan. The pegasus heterogeneous multidatabase system. *Computer*, 24(12):19–27, 1991.

[3] Yigal Arens, Chun-Nan Hsu, and Craig A Knoblock. Query processing in the sims information mediator.

[4] Mattias Cederlund. Performance of frameworks for declarative data fetching: an evaluation of falcor and relay+ graphql, 2016.

[5] Victor Charpenay, Sebastian Käbisch, and Harald Kosch. Towards a binary object notation for rdf. In *European Semantic Web Conference*, pages 97–111. Springer, 2018.

[6] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The tsimmis project: Integration of heterogenous information sources. 1994.

[7] André Dietrich, Siba Mohammad, Sebastian Zug, and Jörg Kaiser. Ros meets cassandra: Data management in smart environments with nosql.

[8] Docker. Docker overview, . https://docs.docker.com/engine/docker-overview/ [Online; accessed 15-March-2019].

[9] Docker. Docker architecture, . https://docs.docker.com/engine/images/architecture.svg [Online; accessed 18-March-2019].

[10] Klaus Drosten, Manfred Kaul, and Erich J Neuhold. Viewsystem: Integrating heterogeneous information bases by object-oriented views. In *[1990] Proceedings. Sixth International Conference on Data Engineering*, pages 2–10. IEEE, 1990.

[11] Facebook. Graphql logo, 2016. https://upload.wikimedia.org/wikipedia/commons/thumb/1/17/GraphQL_Logo.svg/2000px-GraphQL_Logo.svg.png [Online; accessed 19-March-2019].

[12] Gustav Fahl, Tore Risch, and Martin Sköld. Amos-an architecture for active mediators. 1993.

[13] Alexander J Fiannaca and Justin Huang. Benchmarking of relational and nosql databases to determine constraints for querying robot execution logs.

[14] D. Fourie, S. Claassens, S. Pillai, R. Mata, and J. Leonard. Slamindb: Centralized graph databases for mobile robotics. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6331–6337, May 2017. doi: 10.1109/ICRA.2017.7989749.

[15] Gary Court Francis Galiegue, Kris Zyp. Json schema: core definitions and terminology, 2013. http://json-schema.org/draft-04/json-schema-core.html [Online; accessed 17-March-2019].

[16] GraphQL. Server libraries. https://graphql.org/code/ [Online; accessed 11-March-2019].

[17] Jonas Helfer. Graphql vs. falcor, 2016.

[18] Will Howard. Caching with graphql: What are the best options?, 2018. https://blog.usejournal.com/caching-with-graphql-what-are-the-best-options-e161b0f20e59 [Online; accessed 11-March-2019].

[19] Meteor. Graphql vs falcor. https://www.meteor.com/articles/graphql-vs-falcor [Online; accessed 09-March-2019].

[20] Netflix. One model everywhere. https://netflix.github.io/falcor/starter/what-is-falcor.html [Online; accessed 11-March-2019].

[21] nickjanetakis. The 3 biggest wins when using alpine as a base docker image, 2017. https://nickjanetakis.com/blog/the-3-biggest-wins-when-using-alpine-as-a-base-docker-image [Online; accessed 19-March-2019].

[22] Yannis Papakonstantinou, Hector Garcia-Molina, and Jeffrey Ullman. Medmaker: A mediation system based on declarative specifications. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 132–141. IEEE.

[23] Rubanraj Ravichandran, Nico Huebel, Sebastian Blumenthal, and Erwin Prassler. A workbench for quantitative comparison of databases in multi-robot applications. 2018.

[24] Kurt Shoens, Allen Luniewski, Peter Schwarz, Jim Stamos, and Joachim Thomas. The rufus system: Information organization for semi-structured data.

[25] eucalyp https://creativemarket.com/eucalyp prosymbols smashicons https://smashicons.com/, freepik https://www.freepik.com/. Mechanical arm free icon, industrial robot free icon, robotic arm free icon, api free icon. These four icons are being used in mediator component architecture diagram in this research work and the icons are downloaded from www.flaticon.com [Online; accessed 19-March-2019].

[26] Xiang Su, Hao Zhang, Jukka Riekki, Ari Keränen, Jukka K Nurminen, and Libin Du. Connecting iot sensors to knowledge-based systems by transforming senml to rdf. *Procedia Computer Science*, 32:215–222, 2014.

[27] W3. Linked data glossary. https://www.w3.org/TR/ld-glossary/#internationalized-resource-identifier [Online; accessed 13-March-2019].

[28] Wikipedia. Data model, . https://en.wikipedia.org/wiki/Data_model [Online; accessed 13-March-2019].

[29] Wikipedia. Database schema, . https://en.wikipedia.org/wiki/Database_schema [Online; accessed 14-March-2019].

[30] Wikipedia. Federated database system, . https://en.wikipedia.org/wiki/Federated_database_system [Online; accessed 15-March-2019].

[31] Wikipedia. Data abstraction levels, . https://en.wikipedia.org/wiki/File:Data_abstraction_levels.png [Online; accessed 17-March-2019].