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

Your abstract

Acknowledgements

Thanks to ....

Contents

List of Figures

List of Tables

1

Introduction

Robots generate a large amount of data from different types of sensors attached to it and also from its hardware components. In our previous research work [10], 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 [10]. 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. 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 a 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 [1, 4, 2, 3, 3, 11], 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, components used in their architecture. This project mainly focuses on defining semantic based 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, there is a single black box component has been developed to simulate the robot test cases. During the simulation black box stores the data produced by the sensors as dumps into a single MongoDB instance locally.

The first problem here is since the sensor data stored as dumps which makes the consumer's[2] inability to make queries against the data.

And the second problem is missing contexts and the entity-relationship model. For example, if a consumer tries to query the data from dumps, it will be unsure that which sensor produced this data from which robot/black-box at which location and time, and who triggered this test case. What we mean "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 the context about the data should be shared somewhere globally.

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

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

The final problem is, what if we have a situation where multi-robots tries to share data or human controller wants to do fault-diagnosis on data shared on multi-robots.

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 x concisely describes the background knowledge of the topics which are relevant to this work.

2

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 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 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 2.1: 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
double orientation/w
```

For example in the black box, geometry_msgs/Pose ROS topic will be flattened to a simple JSON document which has the data structure mentioned above.

In the above format, '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.

3

Approach

To find the best data models for robotic applications and build a scalable mediator, we begin with SOA analysis 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 [10] 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. Finally, to visualize the integrated data from the mediator in a meaningful way, a GUI application will be developed based on Node.js stack.

4

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.

4.1 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 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?

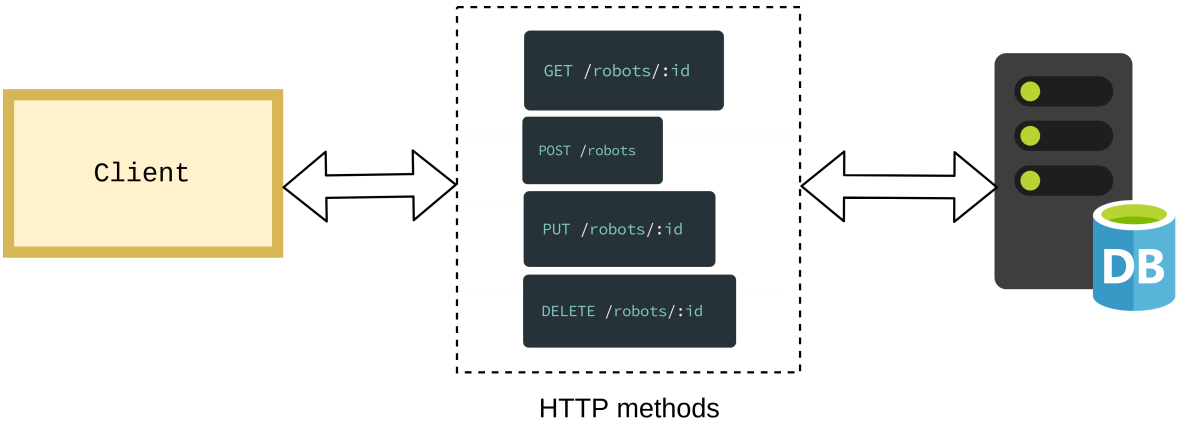

Figure 4.1: Traditional REST API Workflow

4.1.1 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.

4.2 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.

4.2.1 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.

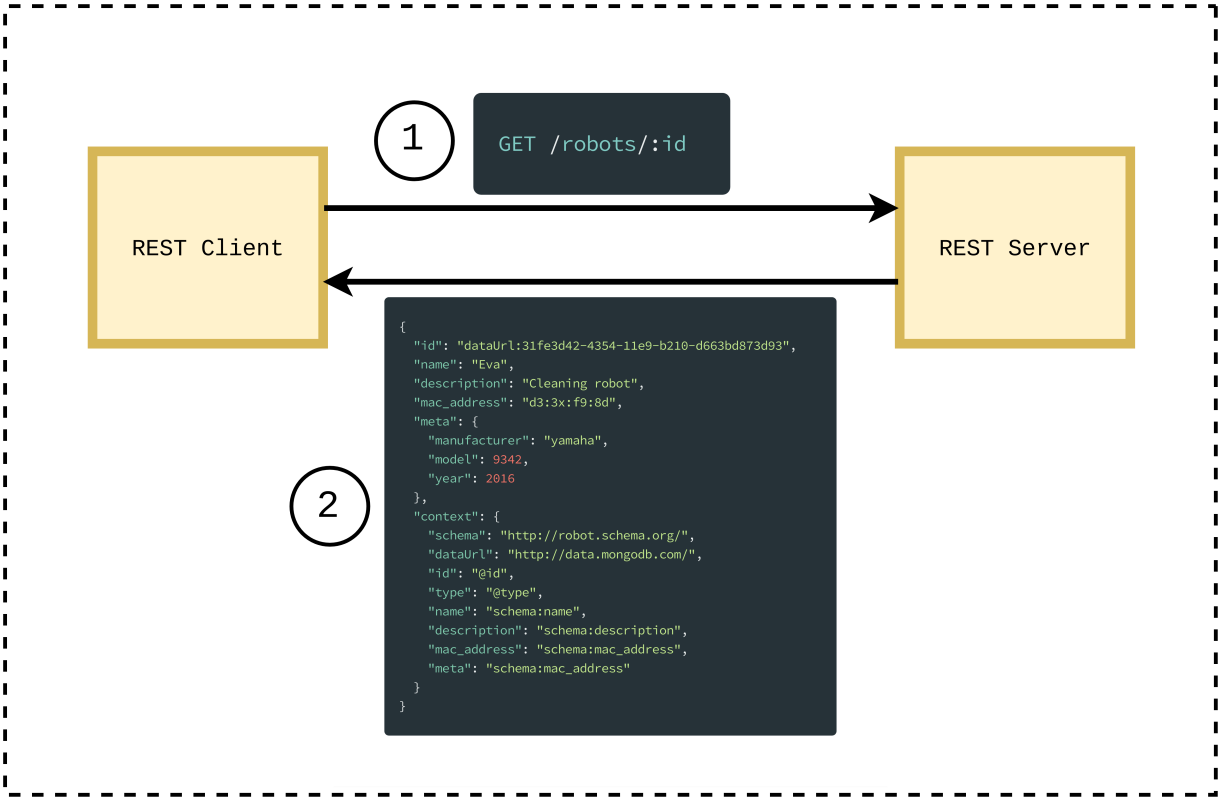

Figure 4.2: Multiple round trips in REST API (a)

Figure 4.2 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 4.3 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.

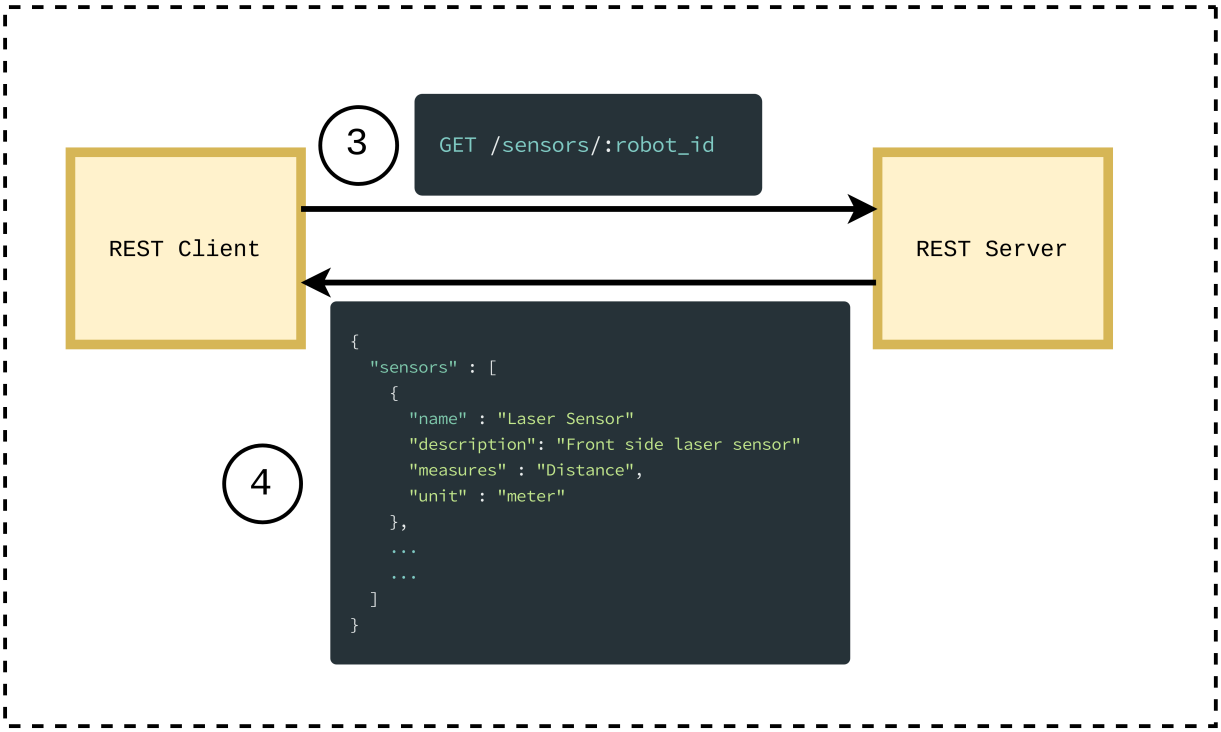

Figure 4.3: Multiple round trips in REST API (b)

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 4.4 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.

4.2.2 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

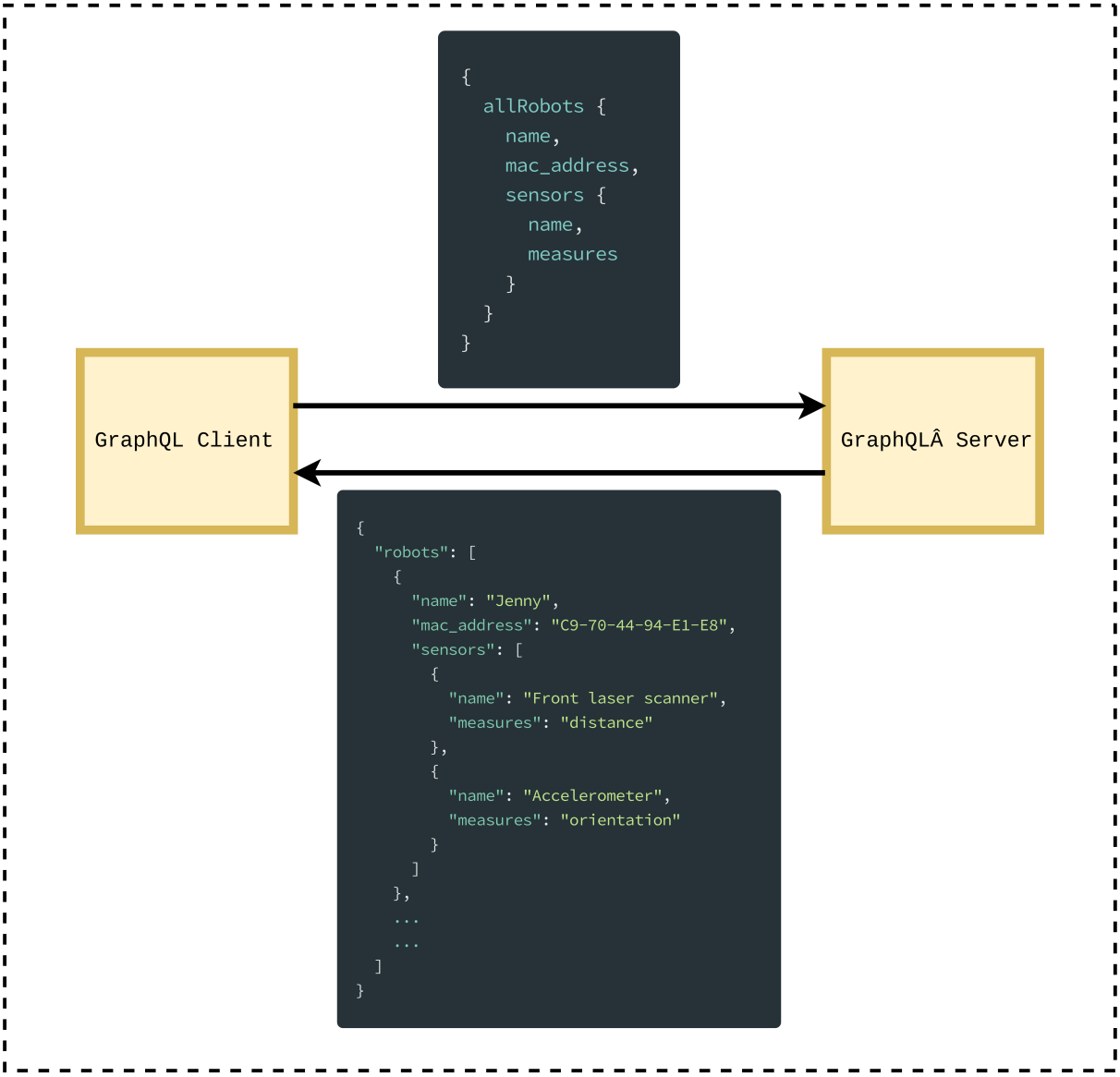

Figure 4.4: Single roundtrip in GraphQL

follows under fetching approach.

Let's consider an example to see the variance between these two approaches. Think 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.

Figure 4.5: Fetching all robot details via REST API includes all the attributes related to each robot.

In the figure 4.5, 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 4.6, 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.

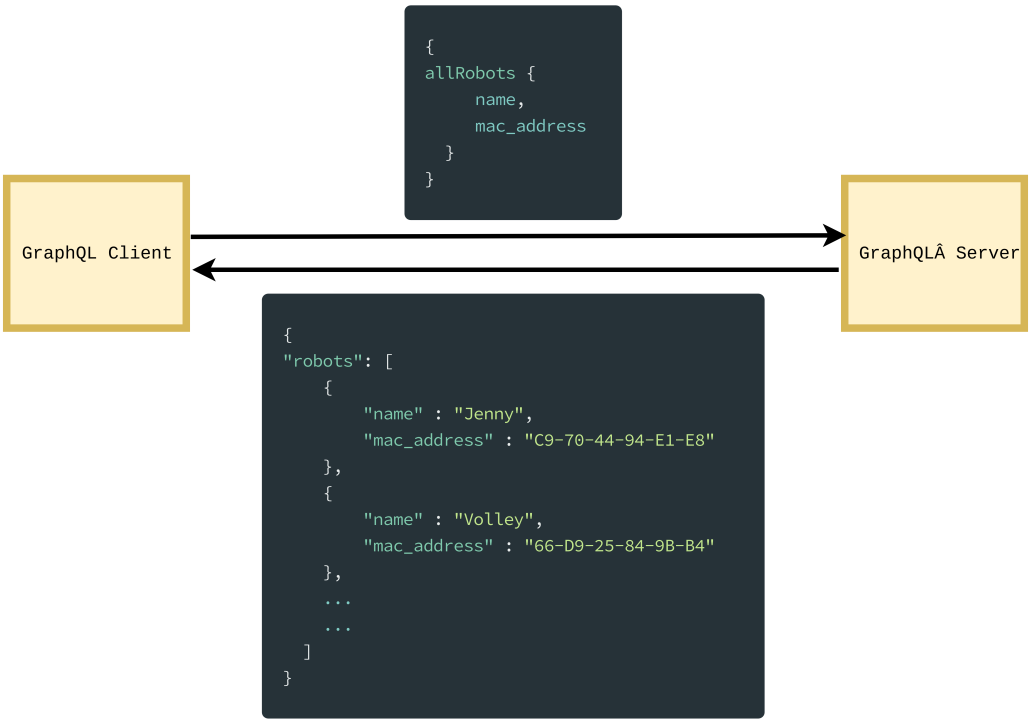

Figure 4.6: Fetching only required robot details via GraphQL

4.2.3 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.

4.2.4 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.

4.2.5 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 [7].

4.2.6 Multiple data sources

GraphQL creates an abstraction layer between clients and the databases used in the backend as shown in the figure 4.7. 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.

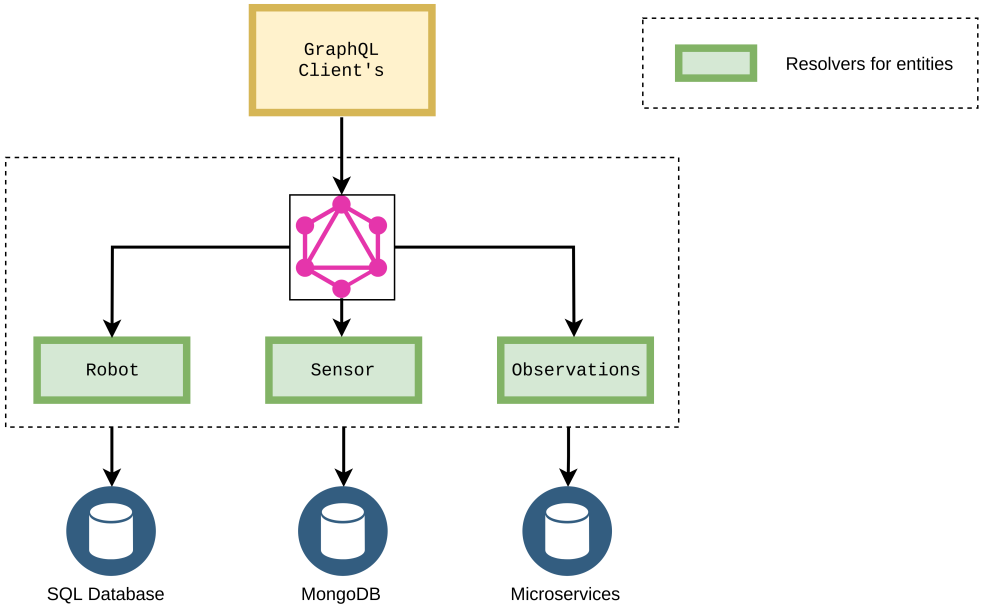

Figure 4.7: Connecting multiple data sources with GraphQL

4.3 Falcor

Falcor is a framework similar to GraphQL developed by Netflix for their internal use, and later they make 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 [9]. It uses "One Model Everywhere" [9] policy to model all the backend data into a single JSON file.

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

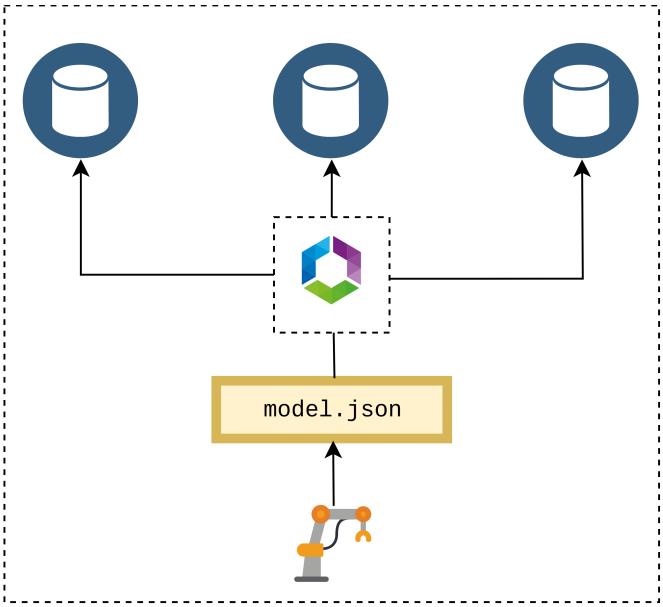

Figure 4.8: Falcor One Model Everywhere design [9]

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" [6] 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 [5].

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 [8].

A

Design Details

Your first appendix

B

Parameters

Your second chapter appendix

References

[1] 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.

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

[3] 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.

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

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

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

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

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

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

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

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