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 [16], 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 [16]. 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, 8, 2, 5, 5, 17], 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

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." [20]

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. 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 Well known data models

- 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" [21]. 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

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" [22]. 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" [22].

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, how it can be interpreted, and how the consumes use the data. 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.

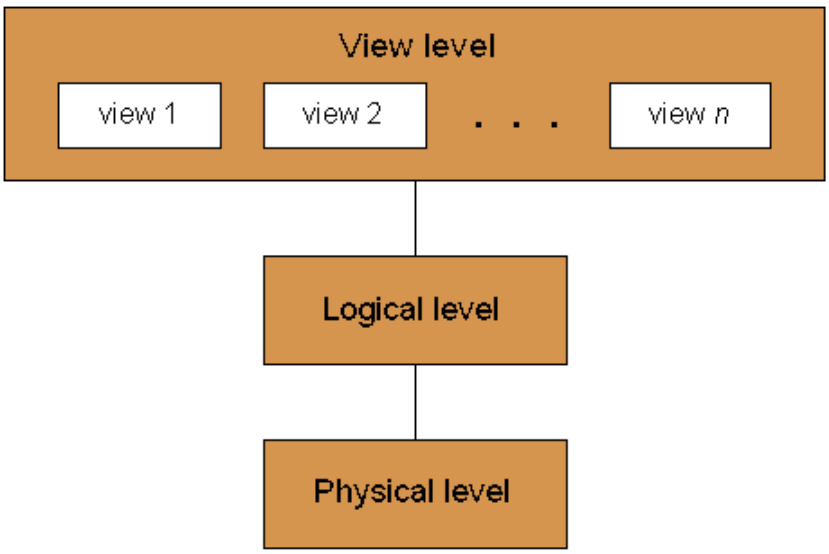

Figure 2.1: Different layers of data abstraction in database systems [23]

- 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.

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" [9]. 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" [6]. 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 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.

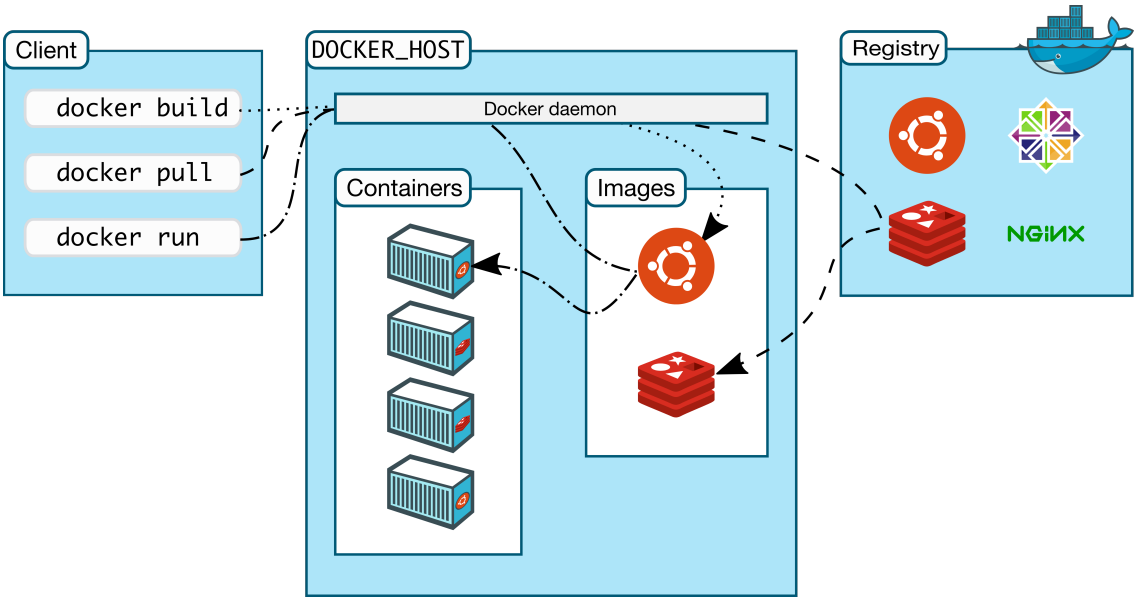

Figure 2.3: Docker architecture [7]

3

State of the art analysis

3.1 Semantic Data model

Su et al. [18] 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 be not excluded 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. [4] 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.

3.2 Mediator architectures

Fahl et al. [8] proposed an active mediator architecture to gather in formation 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.
- 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 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

[1]Active Mediators Object System
[2]Integration Active Mediators Object System
[3]Translation Active Mediators Object System

all the results. To query multi databases from IAMOS, IAMOS servers are mapped with TAMOS servers w+ith the help of Object-Oriented query language.

Ahmed et al. [1] developed a heterogeneous multi-database 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 sobject 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. [5] 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 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. During data integration process, Tsimmiss removes possible duplicates to avoid redundancy in the response. Also, Tsimmiss bundles a default browser tool to query data using OEM-QL language.

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

[4]Heterogeneous Object Structured Query Language

Rufus system proposed by Shoens et al. [17] 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. In terms of query language, Rufus can apply simple object predicates and finding text from the extracted information from the documents or files.

Papakonstantinou et al. [15] proposes a Mediator Specification Language 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
- Schematic discrepancy
- Schematic evaluation
- Structure irregularities

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. [2] 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.

3.3 Declarative data fetching frameworks

Cederlund [3] 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 [3] 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 fields 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 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 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 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
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.

5

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 [16] 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.

6

Data model analysis

6.1 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.

Figure 6.1 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.

```
[
    {"n":"Voltage sensor","u":"Volt","v":9.78}
]
```

Figure 6.1: Simple SenML observation structure

Figure 6.2 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.


```
[
    {"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 6.2: Complex SenML observation structure

To process or semantically query complex sensor data, measurements should have additional context about the data. However, it is not achievable directly with SenML, instead of as explained in this article [18] with the help of transformation from SenML to RDF one can explicitly specify the context of each measurement.

6.2 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.

6.2.1 Thing

It 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, de-

scription, 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 6.3: Structure of Thing entity

6.2.2 Location

It 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 6.4: Structure of Location entity

6.2.3 HistoricalLocation

It shows the thing to be in the location or vice versa, at a given timestamp. For example, consider a robot (thing) is running 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 6.5: Structure of HistoricalLocation entity

6.2.4 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 6.6: Structure of Sensor entity

6.2.5 Datastream

It 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 6.7: Structure of Datastream entity

6.2.6 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 6.8: Structure of FeatureOfInterest entity

6.2.7 ObservedProperty

It 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 6.9: Structure of ObservedProperty entity

6.3 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.

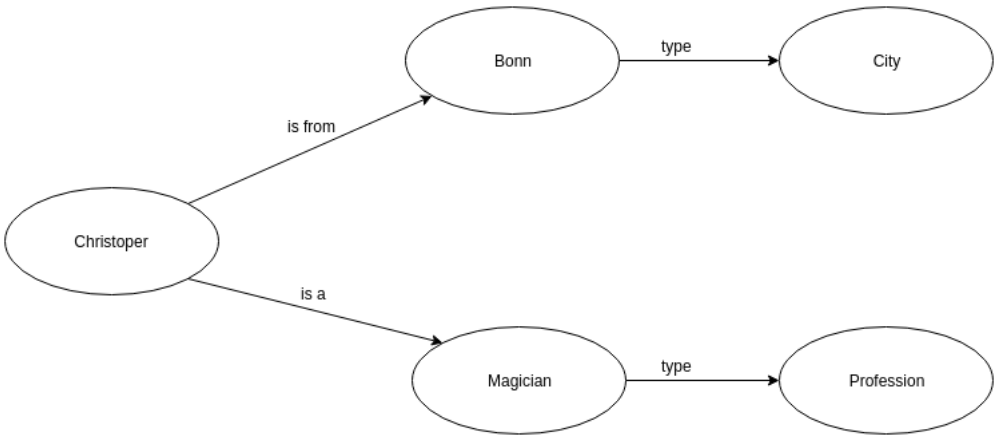

Figure 6.10: Illustrates a simple scenario in the form of RDF structure

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.

6.3.1 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 6.11 'xmlns:rdf' attribute represents the namespace of the current RDF document 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 6.11: Basic skeleton layout of Document

Statement is a individual building block to represent a single triple. In the given example figure 6.12, RDF statement starts with subject description about Christoper and in the next level it have 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'.

```
<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/Profession#Bonn"/>
       <person:age>26</person:age>
     </rdf:Description>
 </rdf:RDF>
```

Figure 6.12: Basic skeleton layout of Statement

The primary advantage of adding semantics to the data is interoperability and easy to query them for a meaningful answer. Let us see 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 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.

6.4 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 [19]. 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.

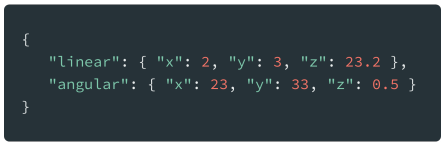

Figure 6.13: Twist message created by robot one

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

Figure 6.14: 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 6.15: 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.

Figure 6.16: 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.

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.


```
[
  {
      "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 }] } ]
  }
]
```


Figure 6.17: 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 }] } ]
  }
]
```


Figure 6.18: Transformed twist message created by robot two after applying expansion algorithm

Now both the messages 6.17, 6.18 look precisely the same without any difference

in keys and values. 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 has been created for common things in the universe but ain't one for robots ecosystem. For example,

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

Compaction algorithm

Compaction algorithm takes two inputs, one is the real data object 6.19 and the second is an object 6.20 which has only context information and generates a human-readable final result 6.21 which includes both data and context injected into it.

```
{
  "@context": "http://example.com/",
  "linear_velocity": { "x": 2, "y": 3, "z": 23.2 },
  "angular_velocity": { "x": 23, "y": 33, "z": 0.5 }
}
```

Figure 6.19: First input to the compaction algorithm

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

Figure 6.20: Second input to the compaction algorithm

Figure 6.21: Result produced by compaction algorithm

6.4.1 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 based 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.

7

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.

7.1 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?

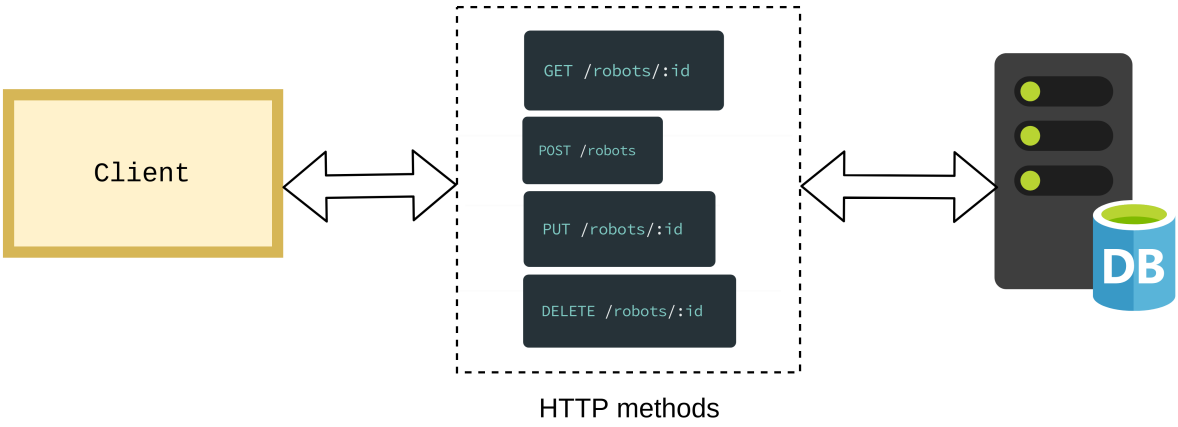

Figure 7.1: Traditional REST API Workflow

7.1.1 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.

7.2 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.

7.2.1 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.

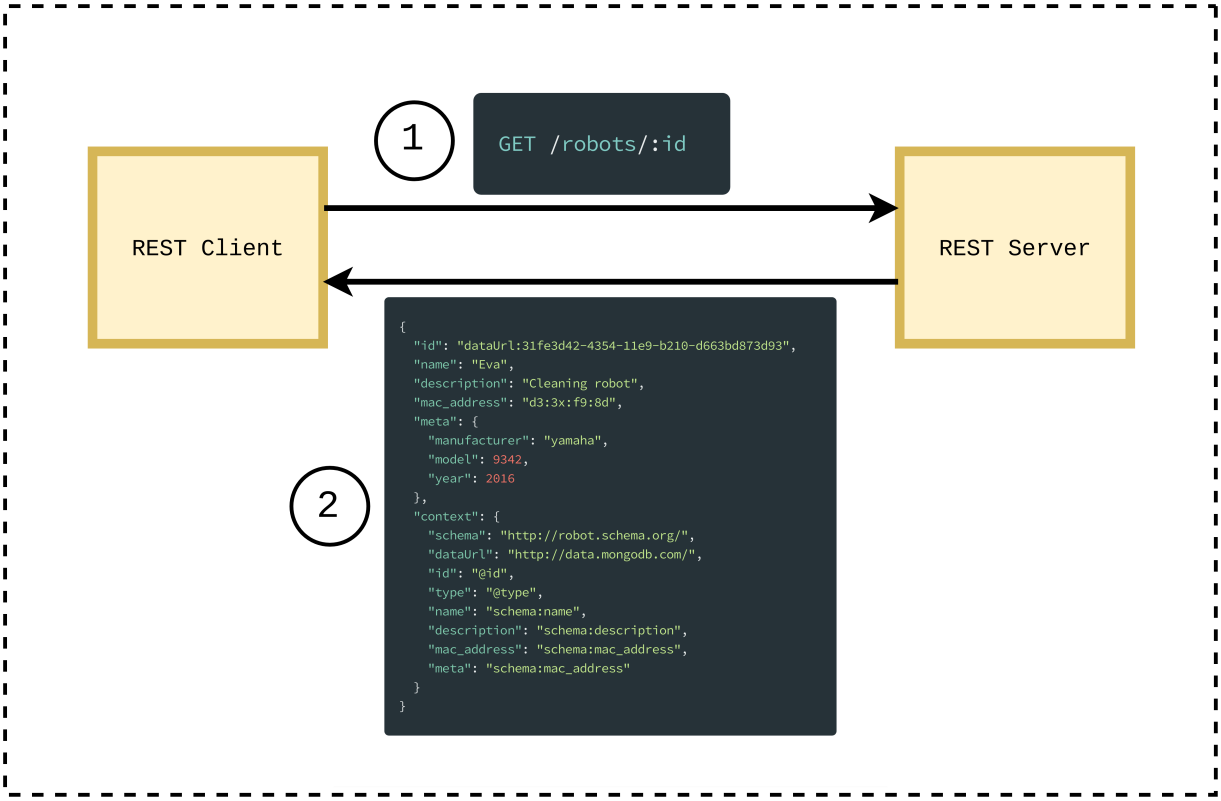

Figure 7.2: Multiple round trips in REST API (a)

Figure 7.2 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 7.3 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.

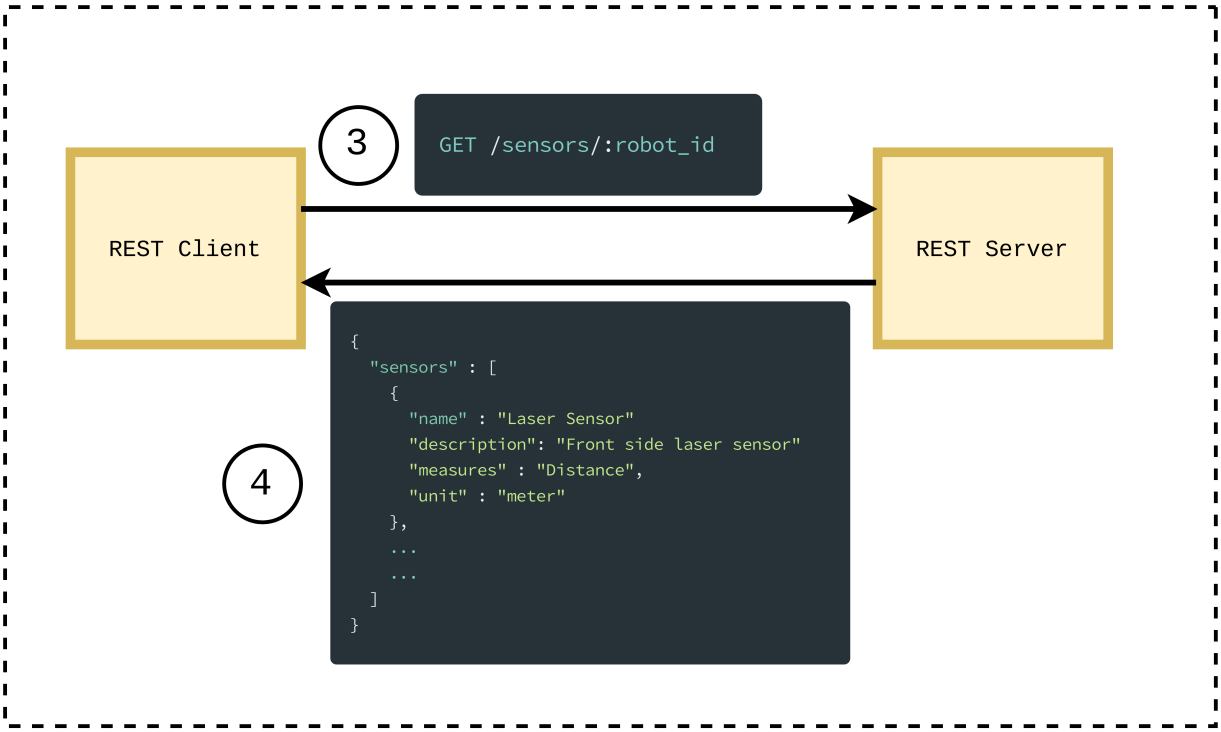

Figure 7.3: Multiple round trips in REST API (b)

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 7.4 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.

7.2.2 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

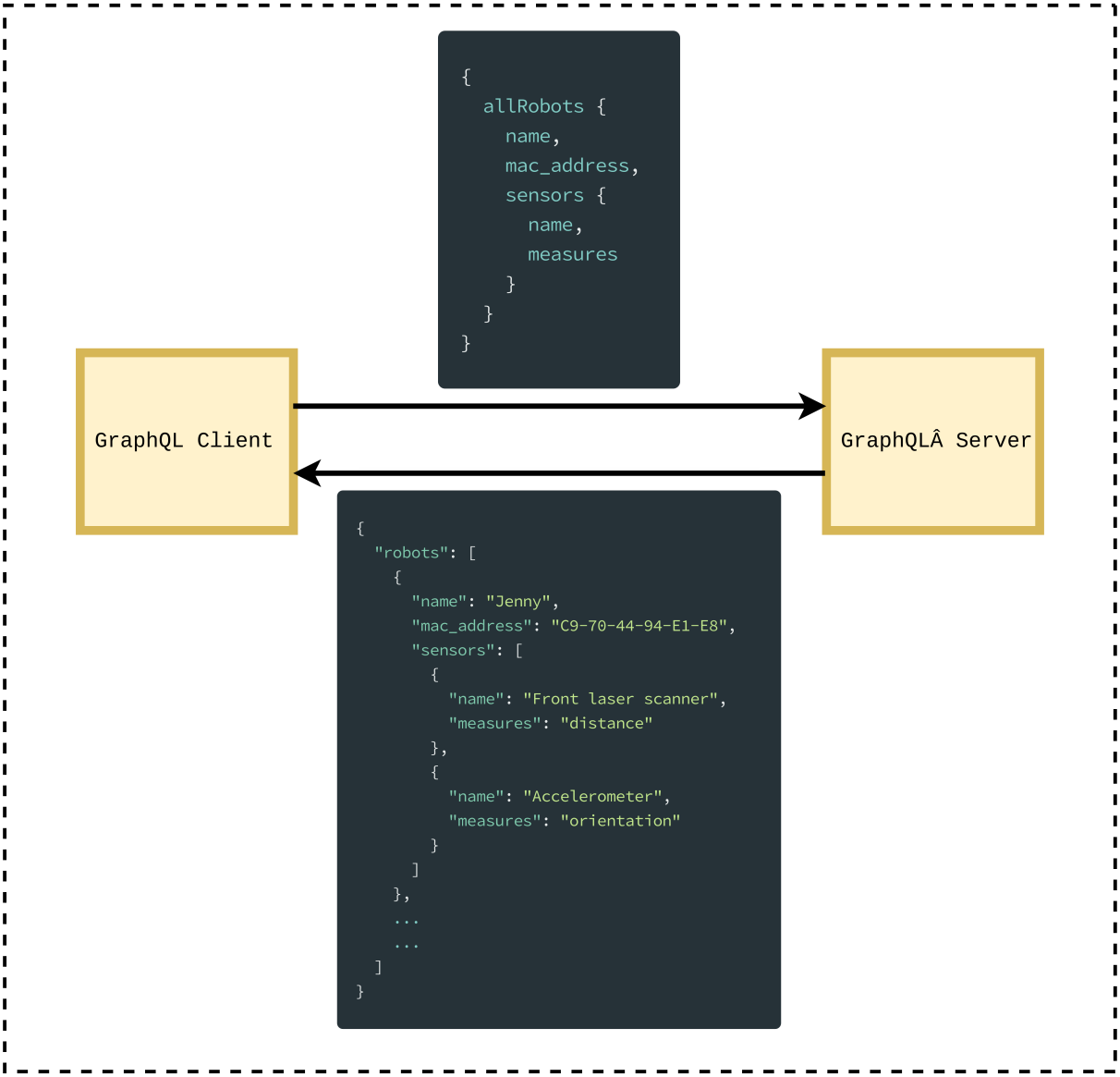

Figure 7.4: Single roundtrip in GraphQL

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

In the figure 7.5, 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 7.6, 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.

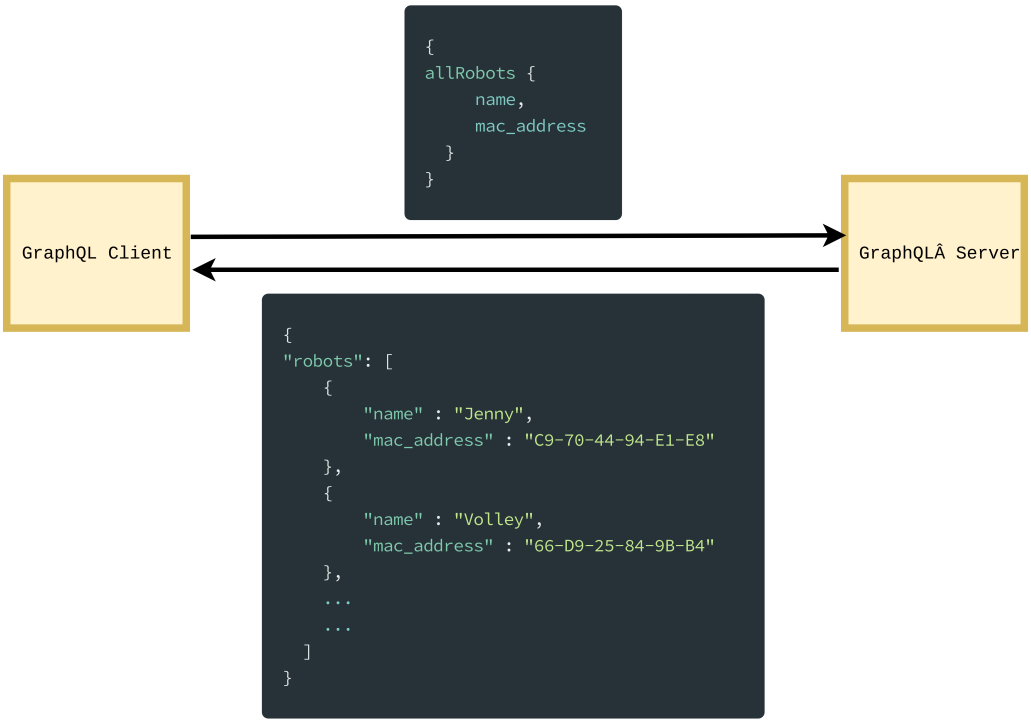

Figure 7.6: Fetching only required robot details via GraphQL

7.2.3 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.

7.2.4 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.

7.2.5 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 [12].

7.2.6 Multiple data sources

GraphQL creates an abstraction layer between clients and the databases used in the backend as shown in the figure 7.7. 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.

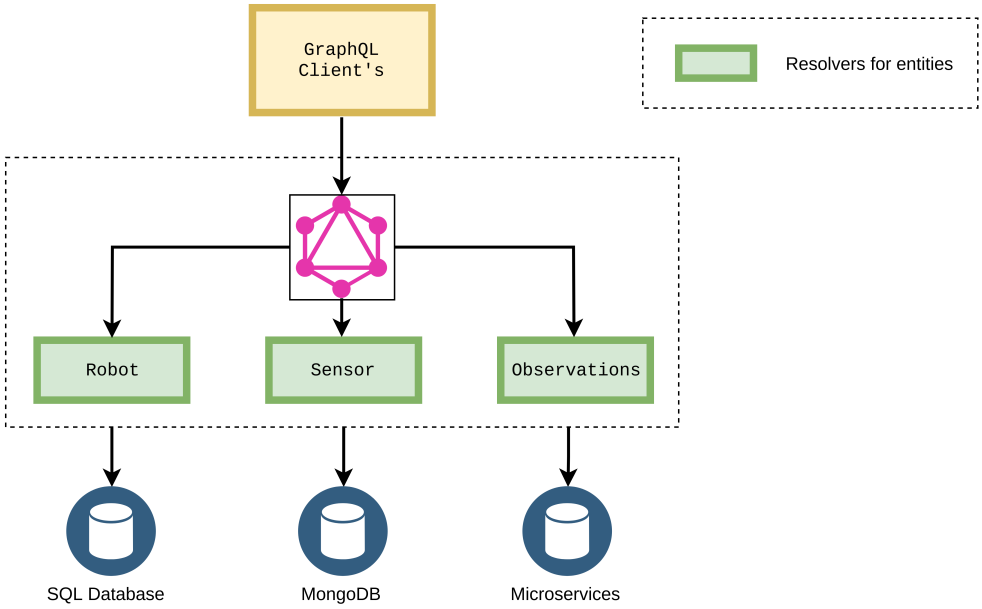

Figure 7.7: Connecting multiple data sources with GraphQL

7.3 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 [14]. It uses "One Model Everywhere" [14] 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.

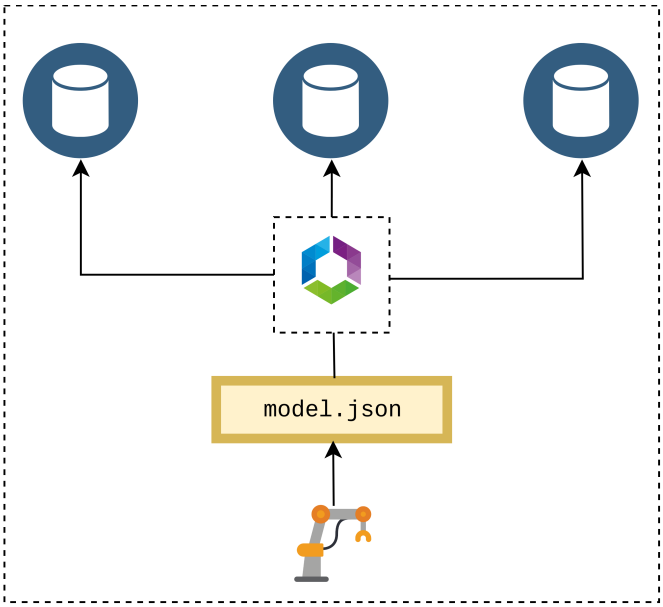

Figure 7.8: Falcor One Model Everywhere design [14]

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" [11] 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 [10].

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 [13].

7.4 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.

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

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

[5] 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.

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

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

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

[9] 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].

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

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

[12] 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].

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

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

[15] 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.

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

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

[18] 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.

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

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

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

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

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