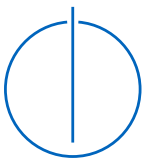
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Ontology Based Practical Rule Engine for Internet of Things

Hakan Uyumaz



DEPARTMENT OF INFORMATICS

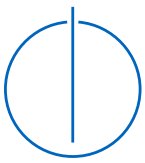
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Ontology Based Practical Rule Engine for Internet of Things

Ontologie Basierte Praktische Regel-Engine für das Internet of Things

Author:	Hakan Uyumaz
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Vladimir Podolskiy, MSc.
Submission Date:	15.03.2019



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.03.2019 Hakan Uyumaz

Abstract

In the current technology state of the world, the internet acts as the central hub for all kinds of communication between different nodes. Regardless of the subject, all types of devices now can directly connect to the internet or can connect indirectly by using gateways. All the improvements in connectivity between devices emerged a new concept called the Internet of Things.

There is a vast amount of data generated by millions of different devices. These data contain different information about both the environment and the state of devices. The data processing is required to make them understandable or to aggregate new information by combining them.

The requirements in any data processes are similar. They only distinguish between each other regarding use-case. However, these differences can be parameterized. Therefore, the programmers only require to configure their system from a high-level perspective. These configurations consist of the information about the user devices, protocols, how data is stored and how data is processed.

In this thesis, the configurable rule engine with semantic capabilities is presented. The thesis also introduce how semantic ontologies can be used for the classification of the IoT data. The rule engine uses ontologies and flows to process and to analyze any data that is received. These flows are also self-adaptable with the caching and optimization features. The rule engine can apply flows either on data streams to derive the active state or the data batches to process and to analyze historical data of a device or devices. The resulted rule engine provides an easy-to-use development environment with a high-level configuration in the domain of IoT.

Moreover, users can use their use-case specific semantic ontologies to handle the data classification. These high-level configurations can either be done in a user-friendly web interface or through an application program interface. Therefore, a developer can define functionalities regarding the process and the analysis of the data. Also, the challenging gap between high-level design and actual low-level implementation has been covered with the flows and ontologies that are configured by the users to match their design.

Contents

1 Introduction

1.1 Background

Internet of Things (IoT) has become an emerging technology with the power of low priced computational units and cloud technology. There are thousands of different examples in the real world which targets both the end-users and the industry.

The problem with the current state of the art is the requirement of tremendous time investment by the developers. Developers need to spend this time to develop both embedded software and web services to achieve their goal. Most of this time is spent on functionalities that are already implemented in different projects with slight differences.

In most cases, the applications have similar capability requirements to process their data with some little customization. The most basic needs during a data process flow can be listed as;

- Retrieving the data,
- Checking whether a data point in a data stream satisfies a condition,
- Checking whether a data batch in historical data satisfies a condition,
- Forwarding the data to another data process flow,
- Manipulating the data,
- Classifying the data,
- Triggering defined events.

There are requirements and challenges while building a system which can process data. Availability, the need of time, integration of a new process into the system and the vast number of different devices can be considered as some examples of these requirements and challenges.

The required functionalities to define data process flows in the area of IoT diverge from each other with only a little customization. Business logic, interoperability of the sensors and actuators, machine to machine (M2M) communication and context-aware services are the divergency items and the primary concern for each IoT system

[1]. Therefore, any abstraction to define functionalities shall also allow fundamental freedom not to lose these divergency items.

To point out the challenges in the current state of the art within an example, a company which delivers IoT solutions for the factory automation systems can be taken into account. This company customizes its system to match with the needs of their customers. In the scope of data processing, this customization may be adding the support of the different type of devices, introducing new methods into the data flow, implementation of brand-new feature or integration of features that are already implemented in the past for another customer.

All of these customizations may lead to challenges which may end up with hiring new developers or losing the potential customers. A generic and practical rule engine can be defined to implement and to maintain the functionalities of the necessary system in the IoT domain. The predefined and parameterized building blocks within a functional user interface (UI) and application program interface (API) can sustain a fast and practical environment to ease and erase these challenges.

1.1.1 Hardware Abstraction Layer (HAL)

In the IoT practical course offered in the Technical University of Munich, the hardware abstraction layer for single board computers is built. With the help of this layer, a single board computer can be configured easily. In the user-friendly web GUI, users can choose between Odroid or Raspberry Pi for their single board computers. Then, they can choose which sensors are attached and which communication protocols are used to send the gathered data. There are currently 23 different supported sensors and 3 different communication protocols. The deployed script in the single board computer can configure itself according to the user configuration made in GUI. It sends all data coming from sensors to the servers using configured communication protocols.

This layer has been extended to a platform with IoT core which covers required web services functionalities to store and access the data of the devices with multi-protocol support. In the following semesters, the development of this platform will be sustained. The goal of this platform is to solve the challenges and problems that developers encounter concerning the need of technical background and time by serving a configurable platform with an easy-to-use user interface. This generic and practical platform can be extended with the easily configurable rule engine to process live data streams or historical data batches of the devices. Therefore, this thesis also aims to integrate with the current platform. The integration details are given in the following chapters.

1.2 Proposed Solution

This master thesis aims to build an ontology-based rule engine to define data flows needed for web services in the area of IoT within a generic and practical development environment. An abstraction of all functionalities that are necessary for any data flow would help to achieve this generic rule engine. The predefined and customizable building blocks, such as ontology models to classify the data and the rules to process the data are crucial for a generic and practical platform with the help of the abstraction. On the other hand, the generalization of the system would create an applicability trade-off. So to maintain practicality, the resulting platform shall be definite and comprehensive enough to support any IoT project and the other areas rather than IoT shall not be a concern. In other words, the building blocks shall not be too broad to support any domain. Additionally, all customization concerning each building block shall be done in easy-to-use interfaces that are supplied to the users to maintain practicality.

While designing a system for data processing that is used in different use cases in IoT as well as others, some design concerns must be taken into account by the developers. The techniques and the architecture to enable communication between all necessary nodes, the format for the data that is stored, how and when data is served can be considered as the primary design concerns [1]. After the design phase of the system has been finished, the design shall be easily implemented to make the prototypes live and to run them. The resulting platform shall support the developers during the implementation process. It should be available to be used with any other services that are already present.

The resulting platform provides solutions in different degrees for the practicality challenges and problems in the IoT domain. These challenges and issues can be listed as follows;

1. The requirement of the technical background,
2. The need for the time investment,
3. The maintenance of the system,
4. The reimplementation of already implemented functionality,
5. The deployment of the system,
6. The different technology backgrounds of the developer in the team.

The rule engine provides solutions to the developers by simplifying the working environment. It solves the problem of the requirement for vast amount of time investment

and technical background, and maintenance challenge by providing practical tools. The platform shall render a practical and easy to learn web UI for hobby users with a less technical background. The data and the rules, which are owned by the platform user, are easily accessible and mutable by the platform user in the UI. The platform users can quickly adapt even if they have little technical background since the system provides building blocks by using the most common user experience (UX) elements such as drag-drop objects, wires, and forms.

The rule engine provides its tools in different levels to keep solutions in a readable way. These tools can be listed as;

- Nodes that are responsible with a single task to apply on data.
- Flows that contain nodes and wire them to each other. They represent a use-case in the entire solution.
- Ontology models to classify data according to business logic
- Ontologies that contains both the ontology model and flows represent an entire solution to realize a business logic.

M. G. Kibria et al. proposed that a semantic ontology which declares every device as objects on the gateways can be used for the energy efficiency [2]. N. Lee and H. Lee also suggested an IoT service architecture that provides services with an object gateway [3]. The semantic ontologies can also be used to classify data with the help of the existing reasoners in the field. Nevertheless, the management of ontology model according to business logic and how a data is represented in an ontology model are the topics that need to be covered to build a generic and practical platform that serves any needs of IoT.

To build business logic and to define data evaluation methods on the ontologies; the flows, which are in if-then form decision-making tools, must be introduced in the platform. The flows function like a chain of blocks, where each does a small fraction of the functionality while interpreting the logic. There are different types of nodes that each has a different function in flow handling. The limited toolbox of generic nodes also preserves the practicality for the users since they do not need to master a vast pool of different block types. Different types of nodes are introduces in following chapters.

In addition to these simplifications, the platform also supports reusability of any functionality or data hierarchy to avoid the reimplementation challenge. Therefore, the developers such as in the company that is given in subsection 1.1 do not need to reimplement functionalities that are required by a new customer. Instead, they can reference the model or the flows that are already implemented for another customer.

The deployment of the system is easier for the most basic user with the existing UI. However, there are more tools for power users to handle their deployment challenge. The power users such as working in a company are provided with an application program interface (API) to use any functionality of the platform through HTTP requests and Internet of Things Easy Query Language (IoTeQL). By using these, the power users can manipulate their models and flows as well as the UI for their basic operations on their system. With the help of API and IoTeQL, existing systems such as the system of the company can be transferred to the platform quickly with the help of a script. Moreover, when the company needs to provide their services with little customization for a new customer, they can promptly achieve it with the help of reusable components.

Since the platform is technology independent and focuses on the construction of the logic, different technology background of the team members is not a concern for the developer teams. Business logic can be simply implemented in the system like drawing a flow chart. They can easily adapt to the development environment of the platform with any background.

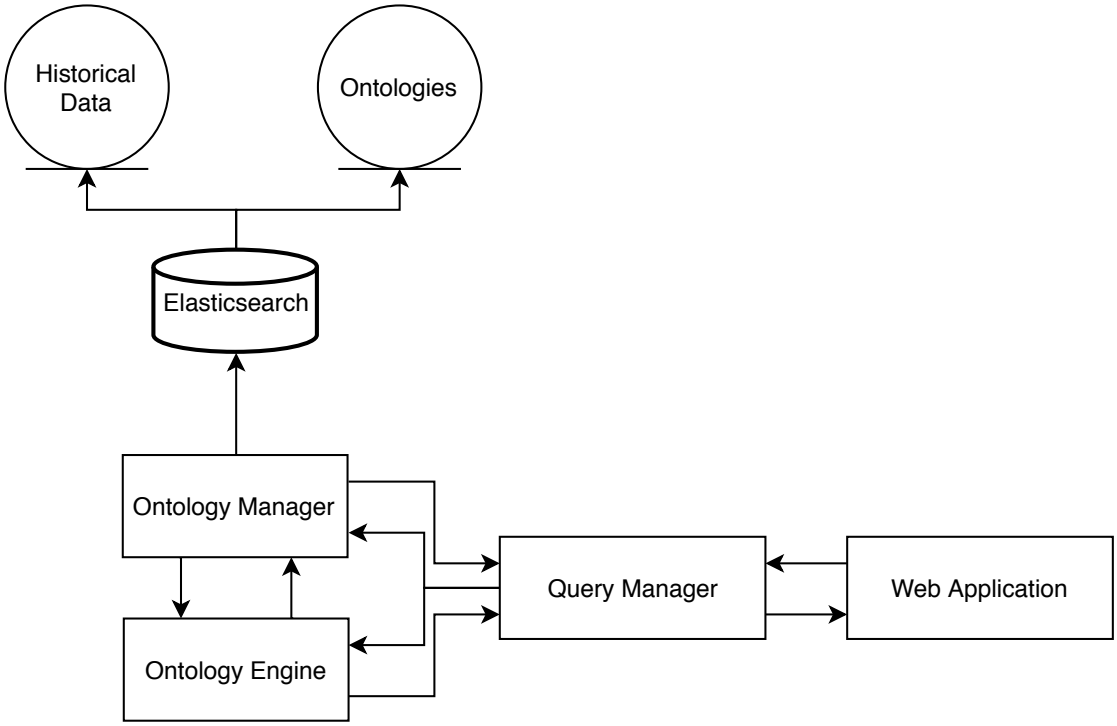


Figure 1.1: The high level architectural diagram of the platform

In figure 1.1, the high-level architecture of the rule engine is shown. The task of the

ontology manager is to run the flows of ontologies on the data when required. The management of the ontology models and classification of data is the responsibility of the ontology engine. The query engine is a high-level application programming interface to manage both the ontology manager and the ontology engine. Since ontology engine and ontology manager are isolated and loosely coupled with each other, they can scale individually with the help of load balancing capabilities of the query manager. Moreover, the query manager can autoscale by using generic autoscale and load balancing services of cloud vendors. Consequently, only the query manager horizontally scales when the number of queries is increased, or just the rule manager scales horizontally when the massive amount of flows is needed to be handled at the same time. A platform user can access all of the subsystems by their APIs, or the query manager to manage all the system at once. Therefore, any change in any subsystem does not lead to a change in the overall architecture.

In figure 1.2, a simple ontology model for classification basic connected car data is shown. Any data can be classified depending on their sourcing sensors or their fields. These ontology models should be defined depending on the use-case. If the classification should be done in a more broad sense, the ontology model should also be extended to match with needs.

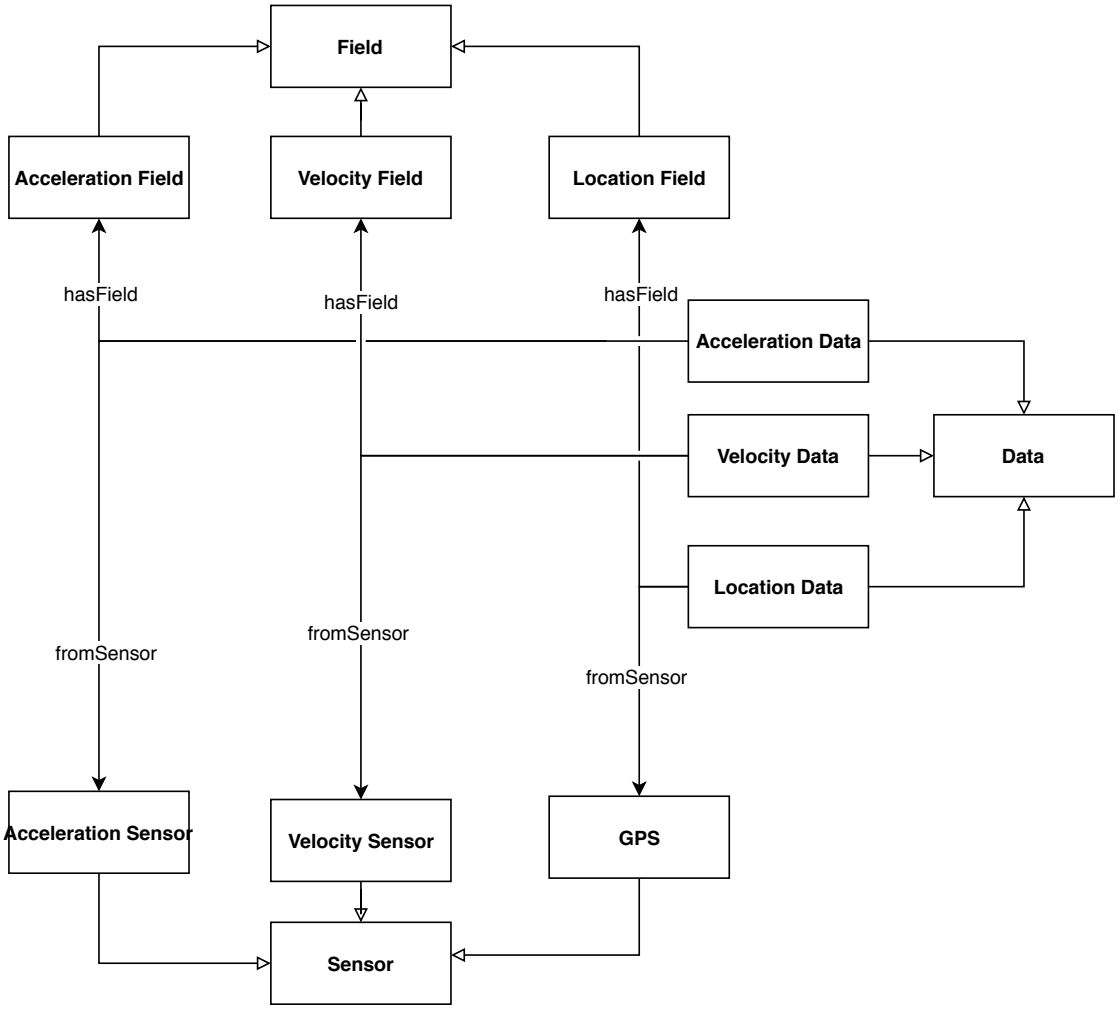


Figure 1.2: Simple ontology of the connected car data

2 Data Management

The data management requirements can differ by the use case. Every requirement is addressed with different tools supplied by the rule engine. Therefore, developers benefit from different tools to realize their use cases by creating components. All the components needed is encapsulated within the ontology of the use case. They all interact with each other to serve the needs of business logic.There are three different components encapsulated in ontologies. These are flows, nodes and models. To keep ontology management simple, any components that owned by an ontology cannot be shared. However, they can be duplicated to other ontologies.

The flows defined by the developer process the data. The flows are the chain of functioning blocks which are called nodes. These flows can be easily implemented by using query engine API directly or the user interface.

The nodes are simple function blocks which are only obligated with one operation in flows. They can be wired to each other inside of a flow. They are configurable according to the requirement.

The classification of data is done in a completely independent application which is called ontology engine. Ontology engine is loosely coupled with the rest of the system. It classifies data by using Web Ontology Language(OWL) and HermiT OWL Reasoner. HermiT uses ontology models defined in OWL and classify data using inference logic. In the entire system, when to classify data and what to do with the classified data is defined in the flows. Flows communicate with the ontology engine whenever it is needed by using correct MQTT topics.

Currently, there are three ways to handle with resulted data. They can be stored in ElasticSearch database, published in defined topics of any MQTT broker or stored in the nodes of the flows for further calculation.

2.1 Ontology Management

As described before, an ontology can have different components to manage different needs. In figure 2.1, the components belong to an ontology is shown. An ontology can have as many flows as it needs to manage the data process. Each flow is consist of various nodes. These nodes can send or receive data to each other inside the flow. However, they cannot be connected with the nodes in different flows. Therefore, the

encapsulation of the data traversing inside a flow is sustained. Ontologies also own an OWL document with ontology model to be used in the classification of data. This document stored and managed by Ontology Engine while all the flows are managed in Ontology Manager.

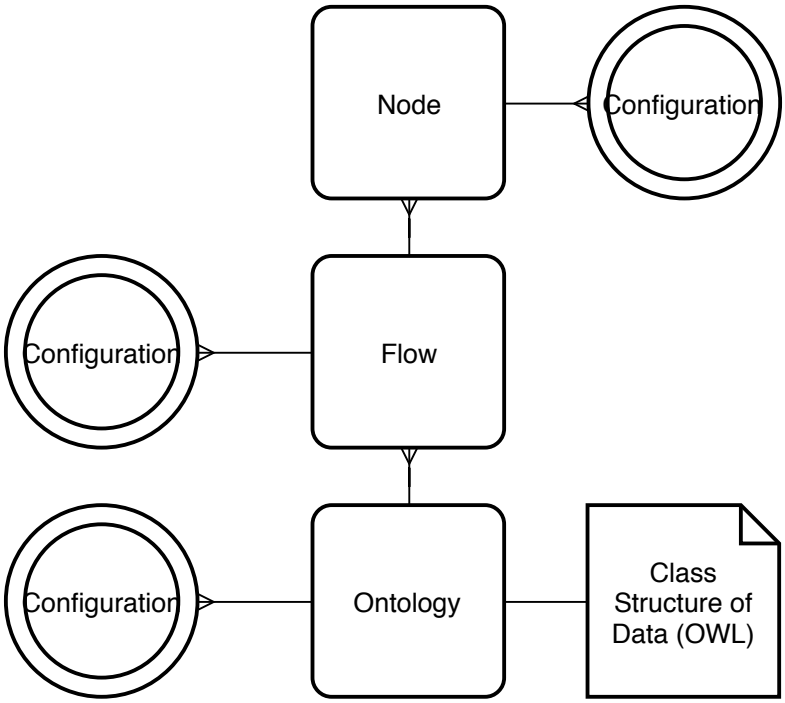


Figure 2.1: Components of an Ontology

All these subsystems can be located in different machines in the cloud without any common resource. Nevertheless, they have to be aware of where each subsystem is located and how they can communicate with them. Most of the communication between components is done in a structured way. The flows send data to classify to Ontology Engine by using "/ontology/query/<Ontology ID>" MQTT topic. Then, Ontology Manager returns after classification the received data back to flows by using "/ontology/classified/<Ontology ID>/<Class Name>" MQTT topic. If a received data belong to more than one class, the same data is published to topics of all matched classes.

2.1.1 Data Classification in Ontologies

The knowledge base of how to classify a received data is kept in Ontology Engine. This knowledge base is defined in an extendable way with OWL. Moreover, each ontology has its knowledge base to operate. Therefore, any user can extend their classification logic by extending the related OWL file. Thanks to these extensions, more complex and specific classification can be made to achieve the goals of business logic.

There are different definitions that can be made in OWL. Classes are used for grouping resources with similar attributes. Individuals represent any concrete resource. The relationship between any two different classes are given with the object properties. The actual values can be assigned to a class are represented by the data properties.

Ontology engine uses these definitions to classify received data. The most critical OWL class which is defined and used in the ontology engine is "IoTOntology:Data". For each received data, a new individual of IoTOntology:Data class of related ontology is created. For each field existing in data, a class named as "IoTOntology:<Field Name>Field", a new individual of that class and a data property named as "IoTOntology:has<Field Name>Value" are created if they do not exist in current ontology model. Then, an object property assertion is added to newly generated data individual with "IoTOntology:hasField" object property and field individual. A data property assertion is also added by using related data and the value itself.

Listing 2.1: Sample Received Data

```
{
    "id": "091798e5-9cbf-4036-98f0-26e13b3a744a",
    "temperature" :24.5,
    "flame": false
}
```

JSON object given in listing 2.1 can be taken as an example. Ontology engine creates a new individual in the ontology for this data. In figure 2.2, generated individual with its properties from this data is shown. This generated individual is used by the reasoner to classify data.

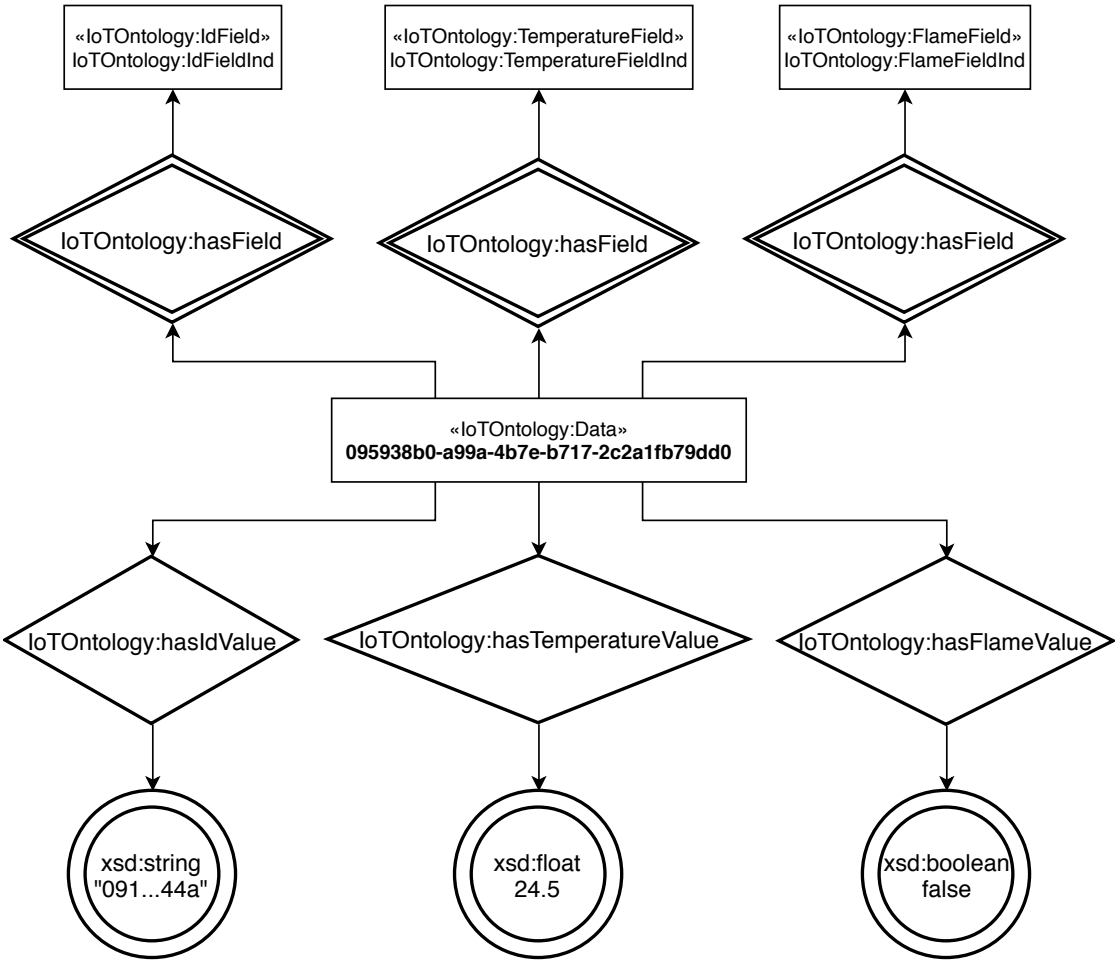


Figure 2.2: Generated Individual of Received Data

With the help of HermiT reasoner, classes that a data belongs to is inferred. When any device is created using Harware Abstraction Layer from TUM IoT practical course, an ontology model which is able to classify data of defined sensors in HAL is also created. By just using this as our knowledge base without any modification, Ontology Engine can deduct that the data belongs to all FlameData, TemperatureData, DataWithTemperatureField and DataWithFlameField classes. In OWL snippet in listing 2.2; OWL classes, properties and restrictions related to this reasoning process can be found.

Listing 2.2: Temperature Data Class from OWL File

```
<owl:Class rdf:about="IoTOntology:Data">
    <rdfs:subClassOf rdf:resource="IoTOntology:Object"/>
```

```
</owl:Class>

<owl:Class rdf:about="IoTOntology:Field"/>

<owl:ObjectProperty rdf:about="IoTOntology:hasField">
    <rdfs:domain rdf:resource="IoTOntology:Data"/>
    <rdfs:range rdf:resource="IoTOntology:Field"/>
</owl:ObjectProperty>

<owl:Class rdf:about="IoTOntology:TemperatureData">
    <rdfs:subClassOf rdf:resource="IoTOntology:Data"/>
</owl:Class>


<owl:ObjectProperty rdf:about="IoTOntology:hasTemperatureField">
    <rdfs:subPropertyOf rdf:resource="IoTOntology:hasField"/>
    <rdfs:domain rdf:resource="IoTOntology:Data"/>
    <rdfs:range rdf:resource="IoTOntology:TemperatureField"/>
</owl:ObjectProperty>

<owl:Class rdf:about="IoTOntology:DataWithTemperatureField">
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource="IoTOntology:hasField"/>
            <owl:someValuesFrom rdf:resource="IoTOntology:TemperatureField"/>
        </owl:Restriction>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource="IoTOntology:TemperatureData"/>
</owl:Class>
```

In figure 2.3, the deduction process for the reasoning of the received data can be found. By the definition of IoTOntology:DataWithFlameField and IoTOntology: DataWithTemperatureField classes, every IoTOntology:Data individual which has at least one temperature or flame field is belong to IoTOntology:DataWithFlameField or IoTOntology:DataWithTemperatureField classes. Therefore, IoTOntology:Data individual is also an individual of both IoTOntology:DataWithFlameField and IoTOntology:DataWithTemperatureField classes. Moreover, it also belongs IoTOntology:FlameData and IoTOntology:Temperature classes since they are super-classes of IoTOntology: DataWithFlameField and IoTOntology:DataWithTemperatureField classes. Hiearchy of

IoTOntology:FlameData and IoTOntology:Temperature classes can be seen in figure 2.4.

$id =$"095938b0-a99a-4b7e-b717-2c2a1fb79dd0"
$D = Data$
$TDf = DataWithTemperatureField$
$TD = TemperatureData$
$T = TemperatureField$
$T* = TemperatureField(TemperatureFieldInd)$
$FDf = DataWithFlameField$
$FD = FlameData$
$F = FlameField$
$F* = FlameField(FlameFieldInd)$

$$\frac{TDf(x) \rightarrow TD(x) \qquad \dfrac{hasField(D(id), T*) \qquad hasField(D(x), T) \equiv TDf(x)}{TDf(id)}}{TD(id)}$$

$$\frac{FDf(x) \rightarrow FD(x) \qquad \dfrac{hasField(D(id), F*) \qquad hasField(D(x), F) \equiv FDf(x)}{FDf(id)}}{FD(id)}$$

Figure 2.3: Reasoning of Received Data

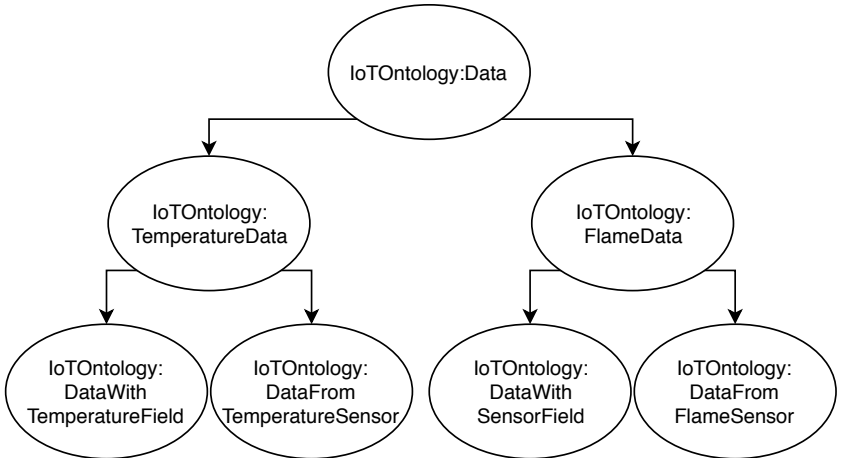


Figure 2.4: Hierarchy of Flame and Temperature Data Classes

2.2 Process of Data in Ontologies

Ontology manager manages every step in the data process. The flows and their nodes handle the data processing in ontologies. The flows manage and encapsulate their nodes. There are three types of nodes with their sub-types. These types are sink nodes, middle nodes, and source nodes. Data is gathered, processed and sunk by these nodes of flows. The source nodes are the entry point for all kind of data. They redirect data that they have gathered to their proceeding nodes. The middle nodes handle all the required process on the data to generate resulting data. They execute their proceeding nodes with their resulting data. The proceeding nodes cannot have information about what is the data like in previous steps. They can only reach data generated by their prior node. In sink nodes, the resulted data is sunk from flow and Ontology Manager to any other system. This system could be Ontology Engine, MQTT broker or Elasticsearch database.

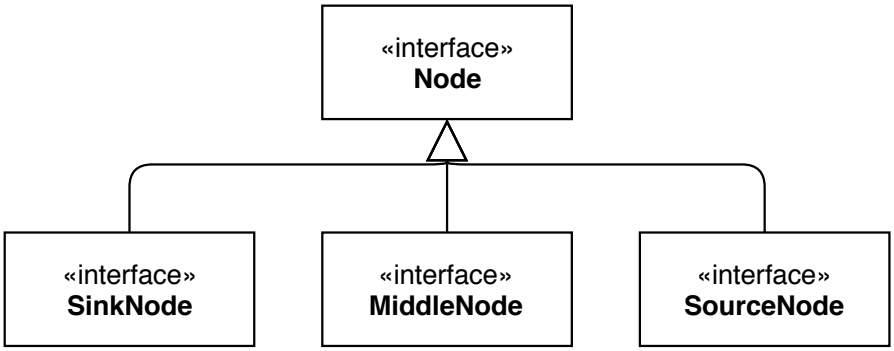


Figure 2.5: Three Node Type

An ontology can have more than one flow with each doing their processes on data. The flows cannot share a node or cannot pass data to each other directly. Nevertheless, it is possible to pass data by making a source node to listen to a message coming over MQTT topic and sinking the data from a sink node of another flow to the same MQTT topic.

The flows also manage the caching functionality and self-adapt accordingly the data reached to their source nodes and resulted in data in sink nodes. When a data reached to any source node of a flow, the flow checks whether this data is cached previously. If it is cached in the cache of the flow, the sink nodes are executed with cached data directly bypassing middle nodes. In figure 2.6, the activity diagram for a flow with multiple nodes is shown.

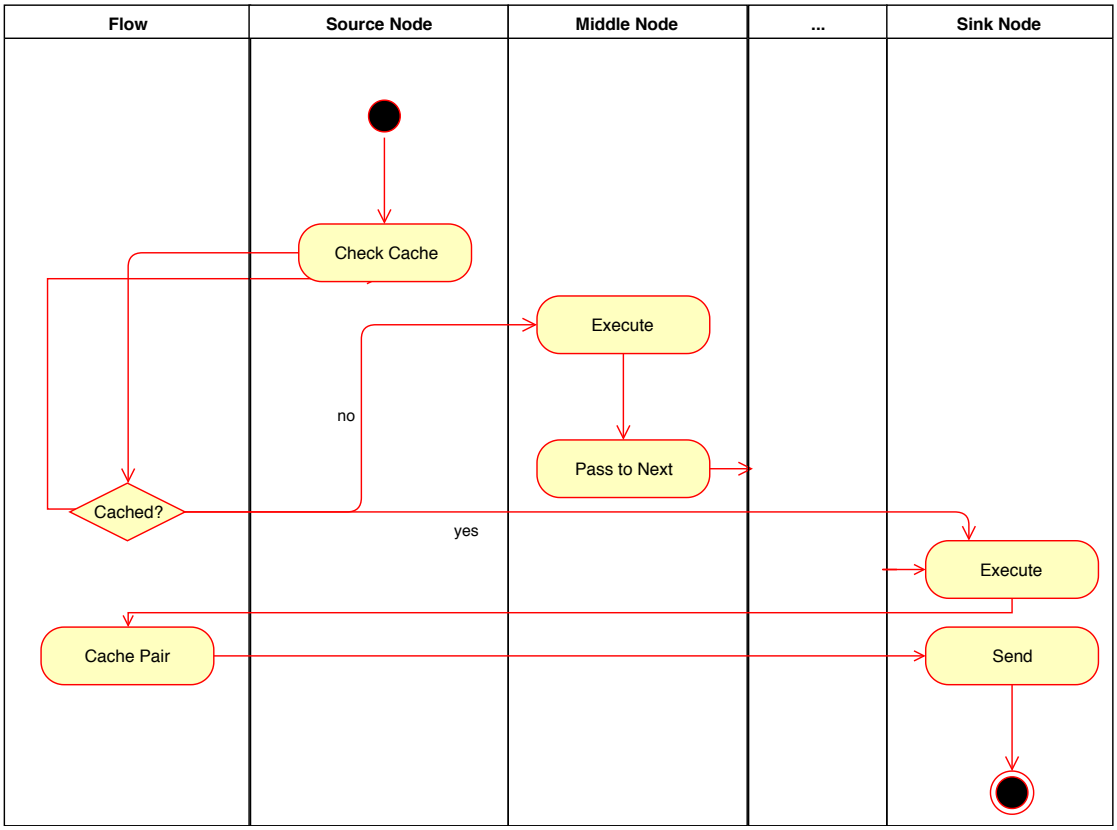


Figure 2.6: Flow Activity Diagram

2.2.1 Nodes for Data Processes

As mentioned in section 2.2, there are three main types of nodes. These are the sink, middle and source nodes. Moreover, each of these types has its sub-types. Depending on the needs of users, they can easily extend these sub-types since they have all their configuration and the implementation of the system serves all kind of abstraction to achieve this.

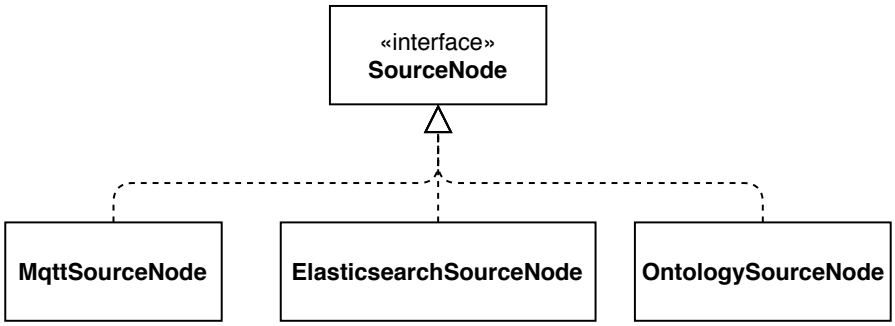


Figure 2.7: Source Node Types

As it can be seen in figure 2.7, there are three different sub-types of the source node type. MqttSourceNode can be configured with the address of any MQTT broker and any topic with wildcard support. When they receive any message from the topic, they parse it into a JSON object and execute their proceeding node or nodes with it. OntologySourceNode can be configured with an ontology class. Therefore, they listen for data that belongs to the same ontology and is classified as its class. They are the entry point for any data classified by using the Ontology Engine. ElasticsearchSourceNode is diversified from the other source nodes by not sourcing data from streams. It is also cannot be set as a starting node of any flows. When ElasticsearchSourceNode executed by a node, it requests the Elasticsearch database according to its configuration and data from the prior node. Then, it passes the gathered data to the next node in the flow. It can be used when a flow requires historical data.

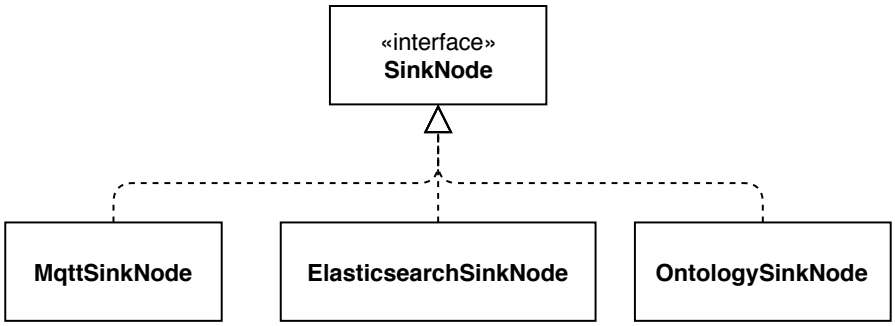


Figure 2.8: Sink Node Types

Sink nodes can be considered as mirrors of source nodes. Instead of sourcing data to be processed in the flow, they sink the resulting data to required endpoints. There are three different sub-types of the sink node type. MqttSinkNode is used to publish the resulting data to any MQTT topic. It can be configured with the address and port of the

MQTT broker and the topic that data should be published. It can also have configured to publish only some fields of the resulted data to the topic and to add the timestamp field to the data just before it publishes. Therefore, any system that is subscribed to the topic can distinguish when the data is generated without any further process.

OntologySinkNode is used to classify any data by using OntologyEngine. It redirects any data to the topic with ontology name it belongs to the MQTT broker located in OntologyEngine. They can be used in anywhere in a flow. Therefore, there is a chance for users to classify the data just after they receive them in flow or after the processing in a flow. ElasticserchSinkNode is very similar to MqttSinkNode. They are used to sink data to the Elasticsearch database instead of an MQTT broker. They can be configured whether to update existing data in the database or to create a new data object in the database. The data stored by ElasticsearchSinkNode can be further used by ElasticsearchSourceNode to process historical data.

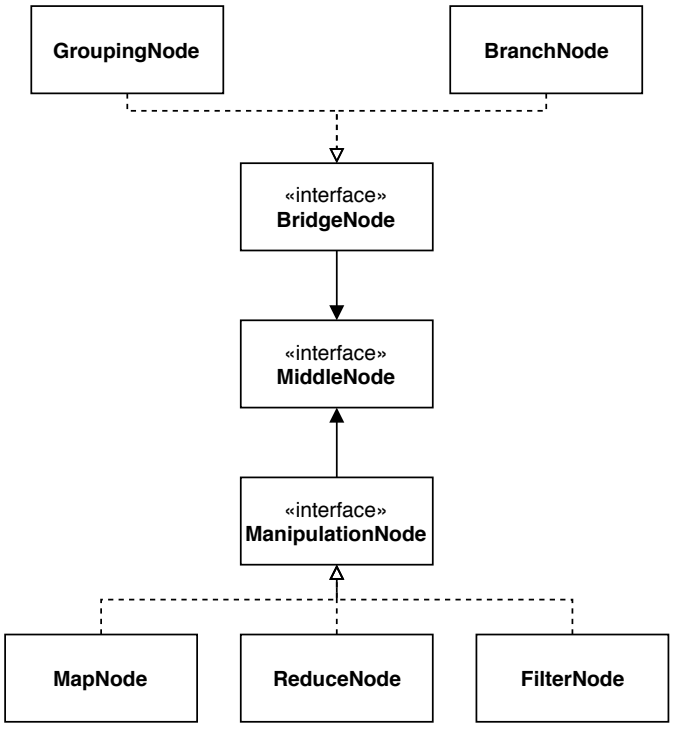


Figure 2.9: Middle Node Types

Middle nodes are grouped under two different interface as can be seen in figure 2.9. The nodes that implement ManipulationNode interface are responsible for manipulating any data according to their configuration. This manipulation is done on the defined

field or fields of the data rather than the data as a whole. Therefore, any field that is not configured in the node is safe and not lost. The manipulation on nodes is done by the user-defined simple JavaScript functions to be executed on fields. The resulting field value can be kept in a newly created field as well as source field. When the result of a manipulation node recorded in a new field, it is an option for the user to keep the sourcing field.

There are three different manipulation nodes which are inspired by one of the most common ways to iterate over a sequence of data. MapNodes are used to change a single value or sequence of values programmatically. They apply a unary function to all values in the field. ReduceNodes use a binary function to combine a sequence of elements to produce a single value. FilterNode applies a unary predicate and filters out the values that do not satisfy this predicate.

The BridgeNodes are responsible for data flow. They decide which path a data should take and when it should take this path. There are two different BridgeNodes which are GroupingNode and BranchNode. GroupingNode groups received data by configured field and create sequences in non-grouping fields with values in the data sequence. It passes a group when the group size reaches the limit configured in the node. Therefore, proceeding nodes of a GroupingNode are only executed when a group reaches the limit. BranchNodes can be configured with a predicate and a field. It checks whether the defined field of data satisfies the predicate. Then, it passes the received data to its proceeding nodes only when the predicate is satisfied.

2.2.2 Self-Adaptation of Flows

The flows can self-adapt to work faster and to have better readability. There are two different self-adaptation methodologies used in the flows. The first one is caching the data as received and resulted in pairs. The second one is the optimization of flows by their nodes.

There exist two different types of caching strategy currently implemented in the flows. Users can make the selection of caching strategy in the configuration of flow. The selection depends on diversity and frequency of data received by the flow. HashCachingStrategy and HashlessCachingStrategy are two different types of caching strategy implemented in the system. The only difference, HashCachingStrategy apply the MD5 message-digest algorithm on data to generate a 128-bit hash value. Then, it uses this hash value as storing addresses of data in both cache and pre-cache. It also uses it for comparison of resulted data in pre-cache. The detailed flow chart of the caching process for both strategies can be seen in figure 2.10. Both of them is configured with a limit and fields. The cache pair for a data pair is only generated when the limit is reached with the same data pair. However, the caching for received data stops

whenever they produce different data in the flow. Nevertheless, some fields depending on use-case can be ignored during caching in both received and resulting data. These fields can be an identification of a sensor, timestamp or any field has no effect on the calculation. Therefore, the flows that are not involved with timestamp or identification can also be created with caching capabilities.

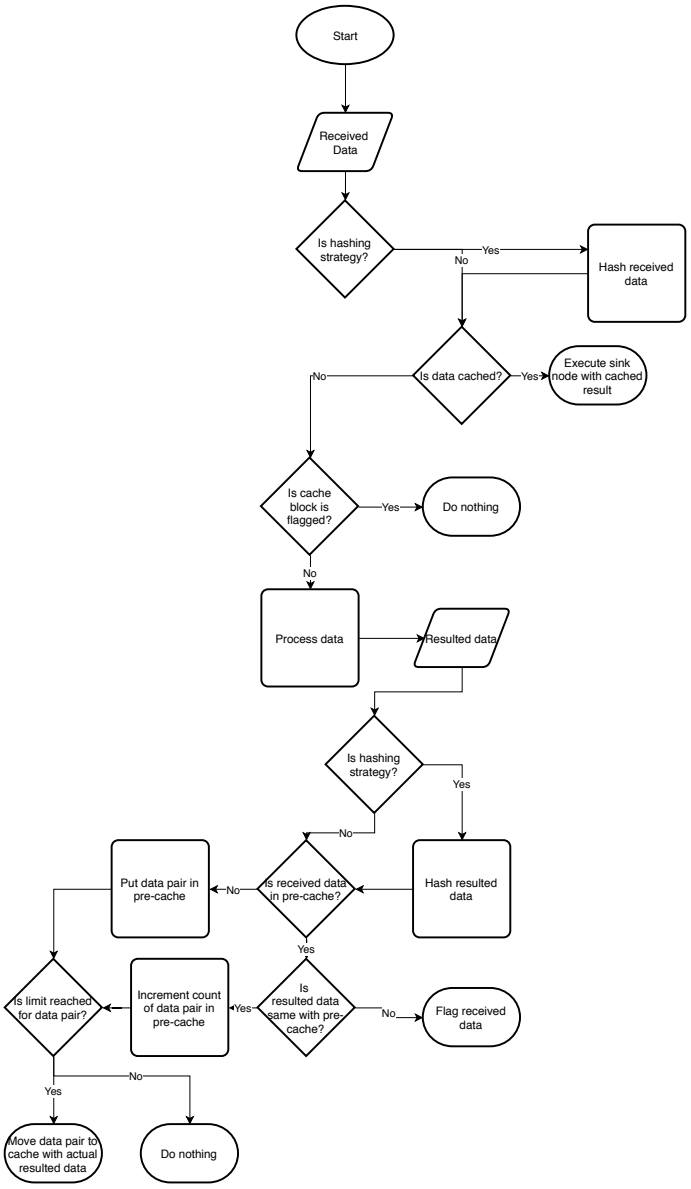


Figure 2.10: Caching Flow of a Data

The selection of whether to use HashCachingStrategy or HashlessCachingStrategy should be selected by the user depending on data frequency and diversity generated by the devices. If data generated by devices are expected to be too divergent, then HashlessCachingStrategy could require a huge amount of memory space. Because of every received data should be stored in cache or pre-cache to keep track of every data. If devices generate data too frequent, then HashCachingStrategy could add an overhead cost for computing MD5 hashes with time complexity of $O(n)$. Therefore, using which caching strategy should be decided concerning this trade-off. Even using no caching strategy might be considered if the user believes that generated data is both too divergent and frequent.

There are also two different optimization strategies which optimize flows simply checking as pairs of subsequent nodes. The BasicBranchMergeStrategy and the BasicFilterMergeStrategy work in a very similar way. They check if two subsequent BranchNode or FilterNode have the same configured fields. Then, they replace them with only one node by merging predicates of the nodes using a logical and operator.

2.2.3 Responsiveness to Further Needs of IoT

IoT is a newly emerged field in computer science. Therefore, new technologies, systems, and standards are introduced every day.

The rule engine is designed modular enough to respond to any need in the future. There are different sub-systems which are loosely coupled with each other. Also, the changes made in any sub-system does not affect all system completely. In other words, the flows do not need to be adapted if the management or the usage of ontology models is changed. Thence, the required changes for adaptation can be made quickly when any language more powerful or popular than OWL arrives.

The users integrate their ontology models to their ontologies. Any change in the system is not required to inherit a new ontology model. So, users who use the open-source ontology models can quickly change to a newer version of the ontology model. This change can be made easily by using APIs of the rule engine. Thus, users can handle models with the most recent classification techniques.

Moreover, the specialized and decoupled nodes decrease the amount of time investment to introduce a new technology or system. For any new system or technology, a new node type can be implemented with parameters that are needed by the new system or technology. Therefore, users can use the new node to integrate the new technology into their business logic.

3 Software Tools and Techniques for Rule Engine

The rule engine can be considered as a Platform as a Service solution. It provides a practical environment to handle data in different degrees for developers. As mentioned before, this practical environment is supplied by its different sub-systems. Each sub-system has its interfaces to interact. Nevertheless, the platform also provides higher-level interfaces to give access to features of the platform as a whole.

By providing interfaces for both sub-systems and the whole platform, the usage of sub-systems as separate components is achieved. Therefore, any user, that own rule engine and wants to use ontology-based classification functionality of the rule engine, can use the ontology engine separately with its interface.

Each sub-system has its API to be interacted. These APIs are designed to match with the demands of their corresponding sub-system.

3.1 Ontology Engine API

Ontology engine manages classification of any data depending ontology definition written in Web Ontology Language (OWL). It is designed to publish all classified data to all interested parties. Therefore, if more than one OntologySource node points at the same ontology class, they should receive classified data. The API of ontology engine is implemented in MQTT to achieve many-to-many communication. Since MQTT is already in the communication stack of ontology manager, the integration with it can be made directly.

The management of ontology definitions in ontology engine is also handled over MQTT to keep the technology stack in ontology engine as simple as possible. OWL files belong to any ontology can be read, created, updated or deleted over matching MQTT topics.

Ontology engine can also be used as a separate component to classify data sent by any IoT system. Therefore, classification functionalities of ontology engine can be integrated into any existing IoT system. These kinds of systems can use the MQTT topics that are used to integrate the ontology manager. Ontology engine subscribes and publishes to predefined topics depending on the ontology.

The topics that the ontology engine subscribes are;

- ontology/create - Used to create a new ontology. It creates ontology based on the id field in the message.
- ontology/update/<Ontology ID> - Used to update an existing ontology.
- ontology/delete/<Ontology ID> - Used to delete an existing ontology.
- ontology/read - Used to request the current OWL file of ontology. It reads ontology based on id field in the message.
- ontology/query/<Ontology ID> - Engine classify data published through messages with ontology definition belongs to id.

The topics that the ontology engine publishes are;

- ontology/read/<Ontology ID> - The topic where response to a read reques is sent. Engine publish OWL files based on read requests.
- ontology/create/<Ontology ID>/<Ontology Class Name> - Engine publishes classified data with ontology id and class name.

3.2 Ontology Manager API

Ontology manager has a REST API to manage every ontology, flow, and node. Any component located in ontology manager can be managed by using this REST API. Ontology manager rearranges its components, runs or stops some of its components based on HTTP requests made into its REST API. It is possible to use the ontology manager as the only system by its API if a user does not need the classification capabilities.

3.3 Managing the Entire Platform

When the user needs the functionalities that satisfied by different subsystems, problems may occur in the management of all subsystems. Therefore, the rule engine provides two different interfaces. These interfaces can be used by the user to manage all subsystems at the same time.

Query manager and the web application manage the entire rule engine. Both of them handle communication between the API of ontology manager and ontology engine. Therefore, a user can create, update, read or delete their components in both sub-system with only one high-level interface.

3.3.1 Query Manager

Query manager is a high-level API to be used in management all platform. Any other system can programmatically use it. It handles the read, create, update and delete requests for all ontologies and their sub-components. When a request received by the query manager, it makes correct requests to ontology engine, ontology manager or both. It only has one HTTP endpoint with the POST method to reduce usage complexity. This endpoint only accepts requests with the Internet of Things Easy Query Language (IoTeQL) payload. Therefore, it is possible and easy to use query manager over existing command-line tools (like cURL). Moreover, any future command-line tool explicitly written to manage the rule engine can be achieved easily. The advantages of using IoTeQL are covered in proceeding sections.

Internet of Things Easy Query Language - IoTeQL

IoTeQL is a language created to manage the entire rule engine. It has read, create, update and delete (CRUD) functionalities. It is used by the query manager to take actions depending on the requests of the user. It can be used to manage ontologies, flows, and nodes.

Each query in IoTeQL has command, type, header, and JSON-like body. It is possible to operate more than one component by using query sequence The command section in a query defines which CRUD operation should be used for that query. The type section defines what kind of component it should handle. Ontology, flow or node can be used for the type field. The sub-type of the component and the id is declared in the header section. If the query is made with a create command, then id field is not required in the header. Id is returned as a result to the user instead.In the body, the rest of the configuration about the component is defined. Nevertheless, users have the freedom to place the body configurations into the header to keep query cleaner.

IoTeQL also has referencing capabilities to other components. These components can be created in the same query sequence or have been already created. For already created components in the ontology manager, ids should be referenced. For referencing components that are being created in the same query sequence, their names can be used since they are still not assigned with an id from the ontology manager. If the same name is used for the different components, then an attribute called as reference id can be added to the header which is used for distinguishing components. This reference id can be used to distinguish components in future.

3.3.2 Web Application - Graphical User Interface

For better usability of the entire system, a web application with an user-friendly graphical user interface (GUI) is implemented. Using this GUI, users can manage all of their components. The web application serves a practical GUI to use every function of the query manager. A user can view, create or edit any component they have using this GUI. The web application translates actions of the user to IoTeQL and makes queries to the query manager.

In figure 3.1, the sequence diagram for a user tries to create a new ontology using the web application interface is shown. The web application serves for usability purposes with a simple user interface. It is also possible for a user to surpass the web application and use query manager directly within their needs. Initial communication configuration of components is set by the Query Manager in their initialization process. It balances and pairs each ontology to existing Ontology Engines and Ontology Managers. Besides to initial configuration, reallocation of a component to different machine or fault in an existing system might cause reconfiguration of the components. For this purpose, the Query Manager also notifies and re-configures each component to how to communicate with their fellow component when needed.

In the web application, creating and viewing flows can be intuitively comprehended. The nodes in a flow are shown as a graph with only one-directed edges. These edges represent proceeding nodes that data should be forwarded. Addition of new edges or new node can be done effortlessly by using simple buttons in the UI.

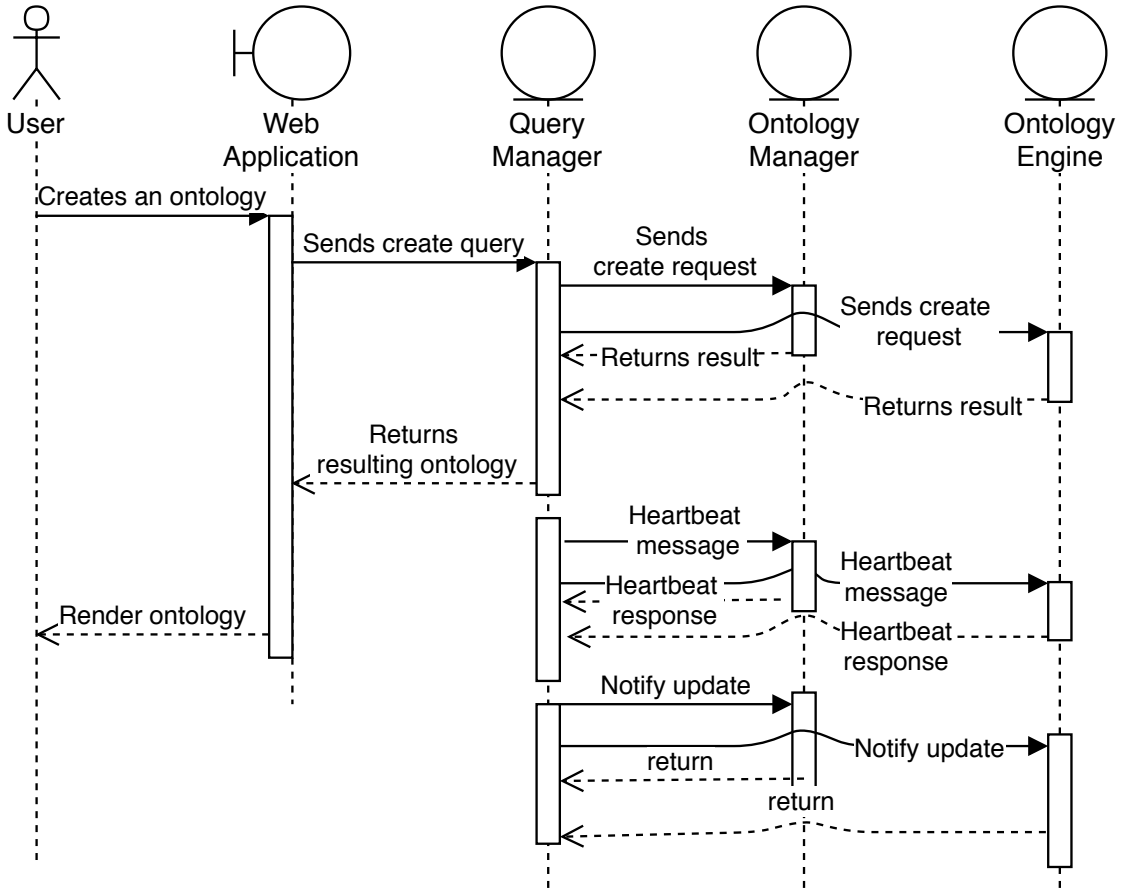


Figure 3.1: Creation Diagram of an Ontology

4 Literature Overview

There are different approaches from the researchers around the world to build semantic ontologies in the field of IoT. The built ontologies are used for different purposes. However, they converge in terms of how they are used. They are mostly used to derive more information from sensor data. The derived information is used for classification, visualization, and interoperation of sensors and sensor data.

There also exist particular implementations to construct a rule manager. Shukla and Sornalakshmi stated that building a rule-based engine to process and integrate logic has several advantages [4]. They described these advantages at different levels for each user. They require less time to re-code and to test when the business policy changes. They are easier to learn and understand than procedural code. Therefore, it can be a good bridge between business analysts and developers. They are less complicated since they are consistent representations of business rules. Moreover, they can be reused by changing the only necessary parts.

This chapter presents an overview of the related researches. It also covers the connection between researches and the thesis. Moreover, how to combine the results of the researches with the resulting platform of the thesis is discussed.

Analyzing and representing raw sensor data is hard since they are generated in different patterns and continuously. Seo, Chun, Oh and Lee proposed and sensor data processing architecture for modeling semantic data from sensor streams [5]. They analyzed raw sensor data with various machine learning strategies. Then, they used this analyzed data and semantic ontologies to process their semantic data to form a semantic knowledge base. Finally, this semantic knowledge base can be used to build a higher level application.

With the help of ontology-based rule engine, the generated knowledge base can be used in the ontology engine to classify data streams. Business logic can be created on top of it by defining flows which use this knowledge base. Therefore, the aim of building higher level applications can be achieved easily in the rule engine.

Park, Bang, Pyo, and Kang built a semantic open IoT service platform [6]. This platform serves a semantic repository to reach IoT data in a structured way. It collects data from sensors. They use ontology models to process this data and gather more meaningful information. Afterward, all the gathered contextual information is stored in a semantic repository. In this repository, the realation between all data is also stored.

Therefore, semantic interoperability between different IoT devices can be achieved. Also, the service can provide this knowledge to involved parties by querying the derived contextual knowledge in the semantic repository using SPARQL.

The results of semantic open IoT service platform can be used together with the ontology-based rule engine. In any use-case, the sensor data can be forwarded to the service platform from a flow in the rule engine. Therefore, a semantic repository with the derived contextual knowledge can be gathered. The derived contextual knowledge can be translated into OWL format. Therefore, users can improve their ontology models in rule engine with the derived contextual knowledge that are coming from open IoT service.

Research has been held for using Semantic Sensor Network (SSN) Ontology to provide visual data service for heterogeneous sensor data. Zhang, Zhao, Wang, and Pan used SSN ontology to derive meaningful information and visualize it from raw sensor data [7]. To achieve this, they mapped the generated sensor data to RDF format. Then, they used SSN ontology to make ontology-based inference on the mapped data. With the help of SPARQL, they managed to query the inferred data with the requirements of users. The visualization of diagrams and maps is achieved by using ECharts with observation values.

Kaed, Khan, Berg, Hossayni, and Saint-Marcel presented a semantic rules engine for the industrial IoT gateways [8]. They designed the rules engine explicitly for embedded devices and other resourceful devices. The primary purpose of the engine is to provide a simple environment to create rules within an abstraction of heterogeneity of devices, protocols, and data. It manages sensors and the data directly coming to the gateway from sensors. The rules that are defined in Lua scripting language are used to manage them. It also provides endpoints to communicate with applications which are located in web servers or cloud. The semantic queries can also be made through this endpoints. The rules engine interact with the semantic engine (SQenIoT) to resolve these queries. With the help of the SQenIoT, semantic queries such as a request for temperature information on a specific floor can be made to the gateway. The evaluation of a semantic query is based on the co-located ontologies and stored annotated data.

The semantic rule engine which is introduced by Kaed et al. [8] has similarities with the rule engine that is presented in this thesis. Nevertheless, the design choice of being specific for the embedded devices, using a scripting language and using semantic ontologies for queries are three distinct differences. In thesis rule engine, the ontologies are used only for classification directly with ontologies, and the annotated data are not kept in anywhere. Nevertheless, processed historical data can be annotated again for classification. The semantic rule engine does not provide a practical way of defining rules such as flows. Therefore, any user should know Lua scripting language to implement a new rule for their use-case.

5 Related Works

In this chapter, the IoT platform from TUM IoT Practical course and widely used technologies which provides similar functionalities for users are reviewed in terms of the relationship with the rule engine.

5.1 IoT Platform from TUM IoT Practical Course

In the scope of IoT practical course held in TUM, participants developed an IoT platform which supports users to build IoT use-cases practically. IoT platform has different layers to help its users.

Hardware Abstraction Layer can be used to configure single-board computers within the needs of use-case. This configuration can be done in a user-friendly web application. Therefore, the single-board computers can read data from its configured sensors and send them to the required endpoints.

Hardware Abstraction Layer (HAL) is integrated with the rule engine. The initial ontology model defined in the rule engine has all classes to identify data that may be received from any sensor supported by HAL. For every single-board configuration, this ontology model is created in the ontology engine. Moreover, the current web application of the rule engine is an extended version of the web application of HAL.

When a new device is registered in HAL, it also creates a new ontology for it by using QueryManager. The required classes to identify any data based on their sensor id are also added to the ontology model whenever a new sensor is attached to the device. Therefore, the classification and process of data can be quickly done by setting the rule engine as a communication endpoint for the device.

Finally, the extension of the IoT platform can be achieved with the help of the rule engine. Any data generated by HAL can be processed and classified according to the needs of the user in the cloud. Moreover, the actions can be sent to HAL according to processed data. Then, the required actions can be taken in device by using actuator support of HAL.

5.2 Node-RED

Node-RED is a flow-based programming tool for IoT [9]. IBM's Emerging Technology Services team initially developed it. It is now supported and owned by Javascript Foundation.

Node-RED also inherits the flow-based programming approach similar to the rule engine. Users can create applications by dragging function specific nodes into their workspaces. The nodes can be wired to each other to create flow. Therefore, users can integrate their hardware devices, APIs and online services.

Node-RED can run any capable device. It can run on, including, but not limited to, Raspberry Pi, Cloud, and even Android [10]. A great community also supports it. There are current 1190 flows, and 1876 different nodes shared in the Node-RED library [11].

In comparison with the rule engine, Node-RED is a more general purpose tool. It is designed to handle any needs of programming in IoT. It can process data as the rule engine in the most basic level.

Nevertheless, the rule engine specialized in data handling. It serves data processing specific tools for its users. It provides built-in caching functionality in the flows. Therefore, fewer computation resources are used for handling the same received data. The flows in the rule engine are also aware of their nodes. Hence, they can optimize themselves with optimization strategies. They can change into more readable and faster types of flows without any operation by the user.

Moreover, the classification of any data is more practical in the rule engine. Semantic ontology-based classification can be made by using the ontology engine and user-defined ontologies. In the other hand, Node-RED does not provide any native practical way for the classification.

5.3 openHAB

openHAB is an open-source home automation software. It provides an environment for integrating different home automation solutions and IoT gadgets in the market [12].

openHAB is built-on pluggable architecture [13]. This architectural choice allows it to support new technologies, systems, and devices as they arrive in the market. There are currently more than 300 add-ons, which supports different technologies and systems, and more than 1500 different devices in its library [14].

Similar to the rule engine, openHAB uses rules-based approach for automating processes [15]. openHAB uses a scripting language called Rules-DSL. Users can define rules to manage all the devices and the systems they use. Rules-DSL creates an easy to

use environment since its syntax, likewise the rule engine.

On the other hand, the initial setup of openHAB is tricky. It requires technical knowledge to choose correct add-ons and use them [12]. The setup may require reading and understanding logs of openHAB. It also requires basic Java knowledge since the scripting language is built on top of Java. It is not designed for any person to use.

6 Use Case Implementation and Performance Analysis

In this chapter, the performance results of the ontology-based rule engine are presented and discussed. The rule execution performance is done by using a real-world example. The use-case based on the real world is presented in the following section.

The performance of the rule engine is analyzed in terms of the two different metrics. The first metric is the ontology creation time. The second metric is the flow execution time.

Different parameters are considered for each metric. The parameters for the ontology creation time can be listed as;

- The number of instances running an ontology manager,
- The number of flows that are owned by an ontology,
- The number of nodes that are owned by a flow,
- The number of virtual users sending ontology creation requests to the query manager in parallel.

The parameters for the rule execution time can be listed as follows;

- The number of virtual devices used in parallel
- Time interval of data sent to flows by the virtual devices.

The other variables remain unchanged during the analysis process. All sub-systems, which are ontology manager, ontology engine and query manager, are set-up in different instances. These instances are created with Amazon EC2 by Amazon Web Services. Each instance is launched with the configuration of 1 GB RAM, 8 GB SSD storage, and 1 virtual CPU. To handle IoTeQL queries, five different instances of query manager are set-up. Amazon Web Services Load Balancer is used for managing loads of these query managers.

The IoTeQL queries that are sent to the query manager are randomly generated. The number of flows inside of ontologies and the number of nodes inside of flows are

randomly selected. The random selection is made with a Gaussian distributed series of numbers with defined minimum, maximum, and skew values.

Two different parameter sets are used for the random IoTeQL query generation. These parameter sets can be seen in the table 6.1.

	Set 1	Set 2
Node Parameters		
Minimum	1	1
Maximum	10	20
Skew	1.5	1.5
Flow Parameters		
Minimum	1	1
Maximum	10	20
Skew	2	1

Table 6.1: The parameter sets used for query generation

The first set of the queries are more realistic and closer to ontologies that can be created during the implementation of a real-world use-case. On the other hand, the second set of queries are more overwhelming and consists of a tremendous amount of nodes.

Therefore, the generated results with the first set of queries give more meaningful information about how the rule engine performs in the real world. Nevertheless, the second set of queries creates an environment for understanding the performance of the rule engine under stress. In other words, the second set of queries still generates valuable information on how the rule engine acts in overwhelming situations.

The load balancing capabilities of the query manager have been used during the analysis. Therefore, queries can be distributed among a variable number of ontology managers. The analysis is done by using 1, 2, and 5 different instances of ontology manager.

The analysis is held for the different number of instances, and each parameter set given in 6.1. The number of virtual users is increased until the creation of an ontology took a few seconds. This can be defined as the limit of the rule engine with given resources. Because of longer response time for an ontology creation request affects the usability of the rule engine.

Open-Source technologies are also used for load generation, result storage, and visualization. For the load generation and the result generation, k6 is used. All generated data is stored in InfluxDB. Grafana is also used for the visualization of the results in a meaningful way. The graphs and the metrics that can be found later in this

section are created by Grafana.

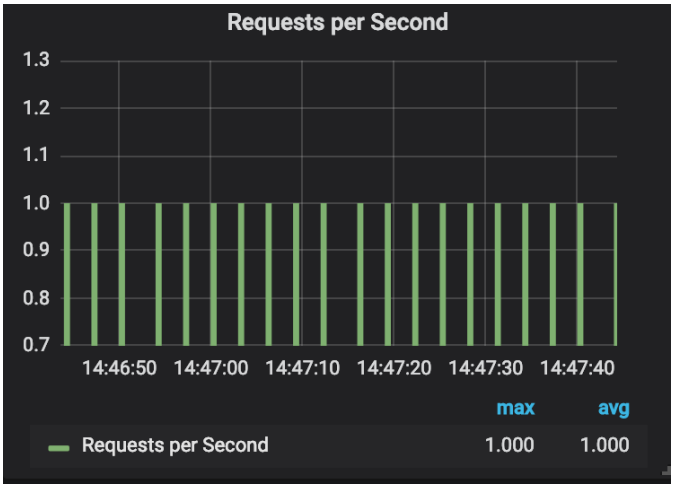


Figure 6.1: Request per second for 1 instance and 1 user with the query set 1

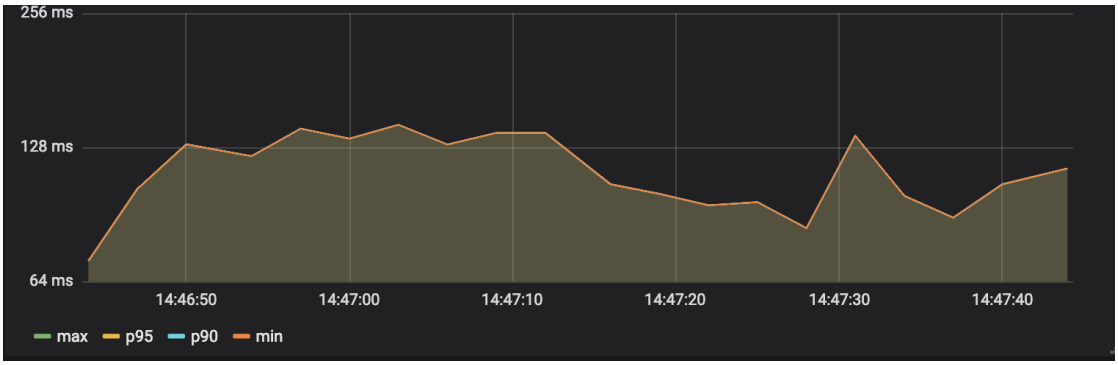


Figure 6.2: Request duration for 1 instance and 1 user with the query set 1

Mean	Median	Maximum	Minimum	90^{th} Percentile	95^{th} Percentile
113.388 ms	110.21 ms	143.73 ms	71.20 ms	137.92 ms	140.87 ms

Table 6.2: Results for 1 instance and 1 user with the query set 1

The very first test is done with only 1 virtual user on 1 instance with the first query set. k6 sent one request per second to the query manager as can be seen in figure 6.1. The ontology creation for this kind of environment can be considered as fast. With

the average time of 113.388 ms, it can actively create new ontologies. Therefore, an ontology would be already created when a new request arrives.

The response time of the rule engine only goes under the expectation when there are 50 different users. Also, these users send the ontology to create requests in parallel continuously. As it can be seen in figure 6.4, the response time for the requests is gone as high as 6.35 seconds. This would slow the work process of the users since this delay also affects updating their existing ontologies. The average requests that are handled in one second is 13.82 as shown in figure 6.3.

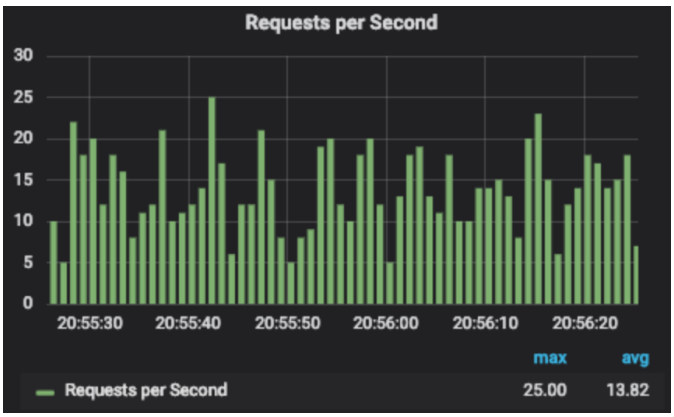


Figure 6.3: Request per second for 1 instance and 50 users with the query set 1

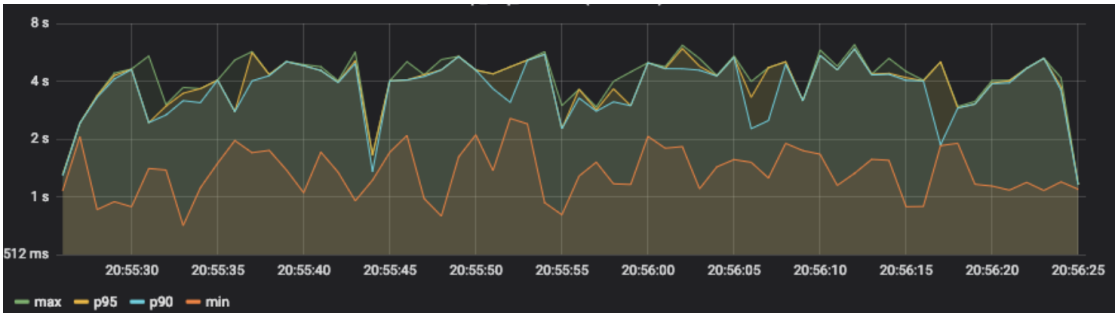


Figure 6.4: Request duration for 1 instance and 50 users with the query set 1

Mean	Median	Maximum	Minimum	90^{th} Percentile	95^{th} Percentile
2.52 s	2.11 s	6.35 s	724.74 ms	4.60 s	5.18 s

Table 6.3: Results for 1 instance and 50 users with the query set 1

In table 6.4, the performance results for one instance with very limited resources is shown. The ontology manager has no problem in terms of creating new ontologies until there are 50 different users in parallel. It is not expected for all users to create a new ontology or update their ontologies this frequent. The ontologies are components that are created once and updated only for maintenance. Therefore, the limit of 50 users can be considered as the limit for the expected number of users who creates ontologies in parallel. A corporate company with thousands of employees can even use the rule engine with only one instance. The rule engine can handle requests of thousands of users since they do not make modifications on ontologies frequently.

Number of Virtual Users	Mean	Median	Maximum
1	113.388 ms	110.21 ms	143.73 ms
10	144.69 ms	123.82 ms	464.12 ms
20	694.50 ms	597.84 ms	2.37 s
30	1.05 s	687.26 ms	3.98 s
50	2.52 s	2.11 s	6.35 s

Number of Virtual Users	Minimum	90^{th} Percentile	95^{th} Percentile
1	724.20 ms	137.92 ms	140.87 ms
10	62.18 ms	220.75 ms	271.39 ms
20	42.03 ms	1.47 s	1.69 s
30	90.41 ms	2.51 s	3.02 ms
50	724.74 ms	4.60 s	5.18 s

Table 6.4: Results for 1 instance with the query set 1

As described above, 1 instance of the ontology manager is enough to handle with requests of employees in a corporate company. However, it might still be used for providing a cloud solution that multiple companies are using. In this case, more instances should be launched, or the resources of existing instances should be increased. When there are two different active ontology manager instance, the response time gets slightly better. As shown in figure 6.5, the average of the requests per second increased to 16.37 from 13.82. The mean of request duration also decreased to 2.03 seconds from 2.52 seconds as indicated in table 6.5. The biggest differences between 1 and 2 instances are maximum, 90th and 95th percentile values. Therefore, the interval for ontology creation is much shorter than only one instance since maximum and minimum values are closer.

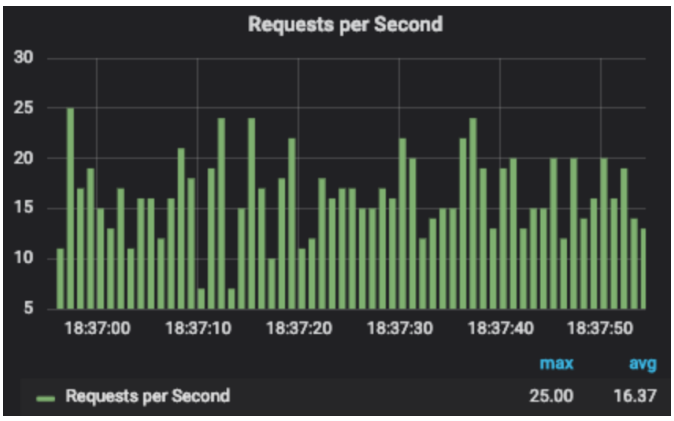


Figure 6.5: Request per second for 2 instances and 50 users with the query set 1

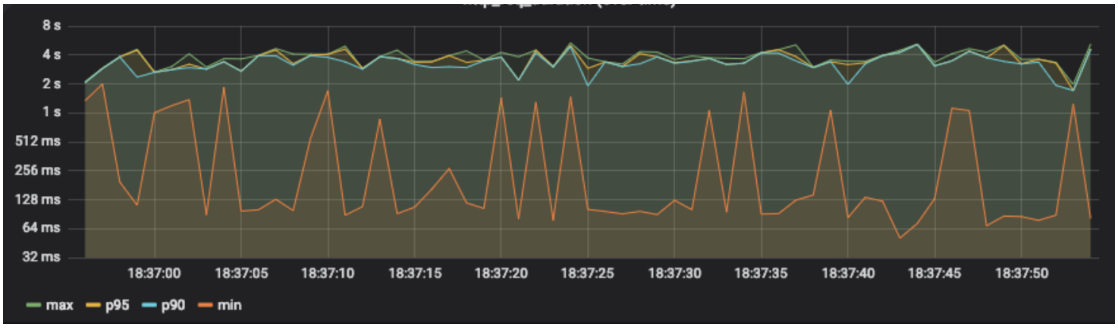


Figure 6.6: Request duration for 2 instances and 50 users with the query set 1

Mean	Median	Maximum	Minimum	90^{th} Percentile	95^{th} Percentile
2.03 s	1.75 s	5.47 s	51.00 ms	3.68 s	4.16 s

Table 6.5: Results for 2 instances and 50 users with the query set 1

If the number of ontology manager instances is increased to 5, then the request duration decreases completely. As shown in table 6.7, it has no problem handling 30 different users at once. It gives almost the same result when 2 instances handle 20 users or 1 instance handles 10 users. The mean request duration gets unacceptable value when there are 100 virtual users. This unacceptable amount is still not too high and only 3.19 seconds.

Number of Virtual Users	Mean	Median	Maximum
10	149.44 ms	128.45 ms	436.94 ms
20	276.82 ms	195.95 ms	2.00 s
50	2.03 s	1.75 s	5.47 s

Number of Virtual Users	Minimum	90^{th} Percentile	95^{th} Percentile
10	57.71 ms	239.23 ms	323.63 ms
20	49.63 ms	545.69 ms	707.83 ms
50	51.00 ms	3.68 s	4.16 s

Table 6.6: Results for 2 instances with the query set 1

Number of Virtual Users	Mean	Median	Maximum
30	259.69 ms	162.85 ms	3.18 s
100	3.19 s	3.14 s	9.79 s

Number of Virtual Users	Minimum	90^{th} Percentile	95^{th} Percentile
30	44.45 ms	488.71 ms	705.72 ms
100	60.82 ms	6.46 s	7.19 s

Table 6.7: Results for 5 instances with the query set 1

When used query set is changed to the second one, the request duration increases dramatically. The tremendous amount of nodes in each query causes this performance fall. As shown in table 6.1, the expected number of flows and nodes are much higher than the query set 1. The results that can be found below show performance of the ontology manager under a stress test rather than a real world use.

The performance of only one instance shows that it can still handle the request of 10 users as seen in table 6.8. However, it gets almost 4 times slower for 10 users. When the number of users changed to 20, it gets nearly 8 times slower. In other words, it reaches an unacceptable value. While it was able to handle requests until 50 users with the query set 1, the delay is too high for even 20 users with the query set 2.

Number of Virtual Users	Mean	Median	Maximum
1	113.388 ms	110.21 ms	143.73 ms
10	524.04 ms	468.04 ms	8.4 s
20	5.07 s	4.73 s	12.83 s

Number of Virtual Users	Minimum	90^{th} Percentile	95^{th} Percentile
1	724.20 ms	137.92 ms	140.87 ms
10	134.16 ms	628.41 ms	700.89 ms
20	658.48 ms	7.85 s	7.98 s

Table 6.8: Results for 1 instance with the query set 2

In table 6.9 the performance results for 2 instances is shown. The impact of having more instances is greater for the query set 2. The performance of two instances for 20 users is better with 1.6 seconds on average than one instance. It was only 500 milliseconds for 50 users when the query set 1 is used.

Number of Virtual Users	Mean	Median	Maximum
5	571.47 ms	492.23 ms	2.08 s
20	3.47 s	3.28 s	9.67 s

Number of Virtual Users	Minimum	90^{th} Percentile	95^{th} Percentile
5	193.10 ms	888.67 ms	1.11 s
20	180.02 ms	6.45 s	7.60 s

Table 6.9: Results for 2 instances with the query set 2

The performance results for 5 instances can be seen in table 6.10. Five instances can handle requests of 20 users in 1.34 seconds on average. However, the performance goes lower than the sufficient amount when there are 30 different users in parallel.

Number of Virtual Users	Mean	Median	Maximum
1	408.07 ms	401.11 ms	519.07 ms
10	504.00 ms	471.34 ms	1.70 s
20	1.34 s	824.88 ms	6.33 s
30	4.23 s	4.75 s	12.76 s

Number of Virtual Users	Minimum	90^{th} Percentile	95^{th} Percentile
1	279.82 ms	483.52 ms	507.04 ms
10	90.23 ms	718.51 ms	870.93 ms
20	134.80 ms	3.42 s	4.06 s
30	177.81 ms	7.85 s	8.69 s

Table 6.10: Results for 5 instances with the query set 2

6.1 Overspeed Warning System for Connected Cars

The automobile industry is developed to be smarter with the growth of connected mobility. Even mid-level cars have a connection to the internet over mobile services. They use this connection taking remote actions on cars or sending generated data to the internet for further processing.

As a use case example, the concept of connected cars is taken in the account. The cars in this use case are assumed with GPS integration. The speed and speed limit in the current road considered to be gathered from this GPS.

The use-case is about a warning system which informs users when they are close to the speed limit with their acceleration. There are two different warning levels. The critical warning is sent to users when they are expected to reach a speed limit within 5 seconds. The regular warning is sent to users when the prediction of reaching the speed limit within 20 seconds is made five times one after another.

An ontology with three different flows and an ontology model is created to match with the requirements of the use-case. The flows process the data received from cars and send them particular MQTT topics when they generate a warning. The ontology models are used to classify the processed data whether it is a warning or critical warning.

The first flow can be seen in figure 6.7. It first receives data from cars. Then, it combines every two subsequent data from the same car to find the current acceleration of the car. It uses this acceleration and current speed to find when the car reaches the speed limit if it keeps the acceleration. Finally, it sends the processed data to the ontology engine for classification. The node by node description of car data classifier

flow can be found in table 6.11.

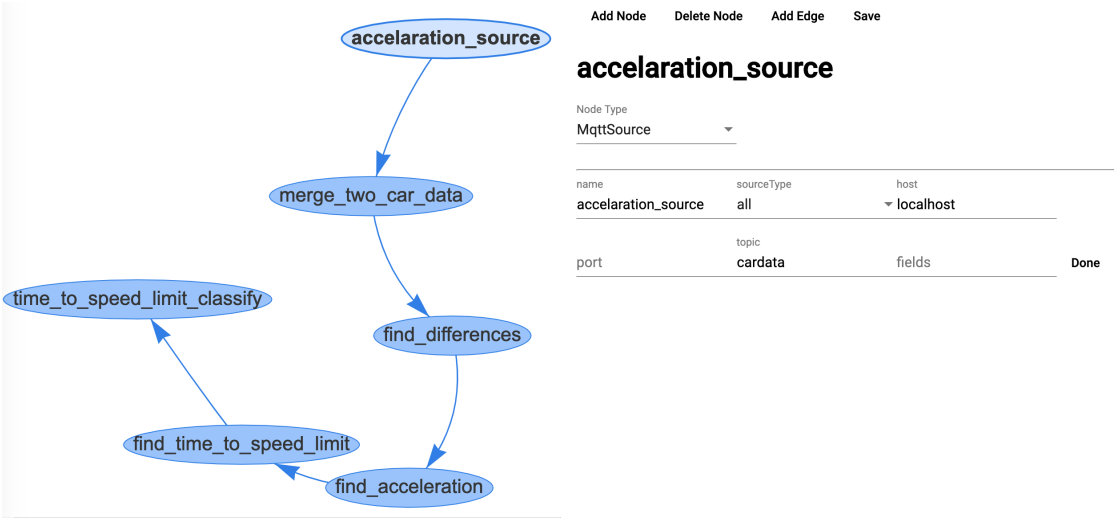


Figure 6.7: Car data classifier flow

Name	Type	Description
acceleration_source	Mqtt Source	Gathers data from cars
merge_two_car_data	Grouping	Merge two subsequent data of the same car
find_differences	Reduce	Find differences between speed and timestamp of two data
find_acceleration	Map	Find acceleration using differences
find_time_to_speed_limit	Map	Find expected time to speed limit using speed limit, current speed and acceleration
time_to_speed_limit_classify	Ontology Sink	Sends generated data to the ontology engine

Table 6.11: Nodes of car data classifier flow

The car warning flow can be seen in figure 6.8. It gathers all data that are classified as a warning from the ontology engine. It first checks whether the last three warning are subsequent and time differences are low enough. It sends to warning topic of MQTT broker if the three warnings satisfy the condition to be a warning. In table 6.12, descriptions of each node can be found.

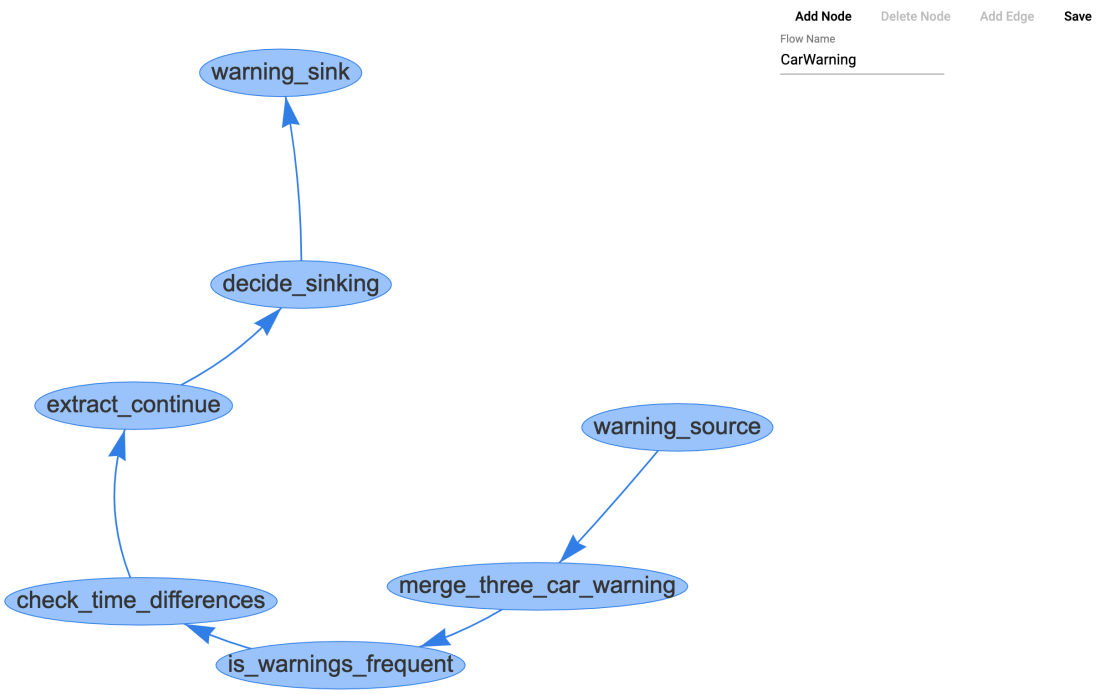


Figure 6.8: Car Warning Flow

Name	Type	Description
warning_source	Ontology Source	Gathers data with warning class
merge_three_car_warning	Grouping	Merges three warning data
is_warning_frequent	Map	Checks and flags whether three data is subsequent
check_time_differences	Reduce	Checks and flags time difference of each data is below 20 seconds
extract_continue	Branch	Decides to continue by checking flags
warning_sink	Mqtt Sink	Sends data to warning topic

Table 6.12: Nodes of car warning flow

The critical warning flow is just used to redirect all critical warning data to critical warning topic of MQTT broker. The flow and nodes are shown in figure 6.9 and table 6.13.

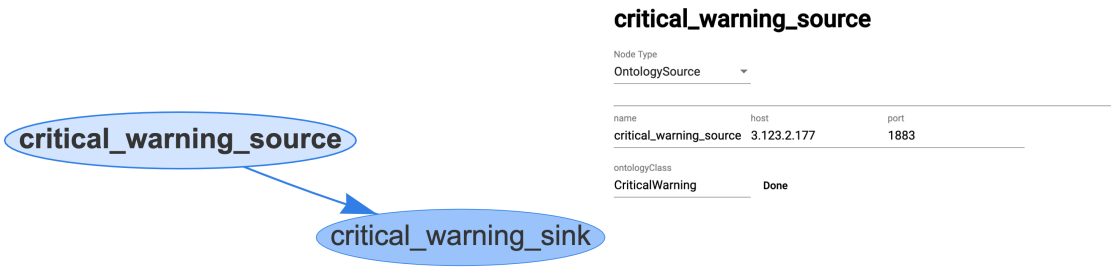


Figure 6.9: Car Critical Warning Flow

Name	Type	Description
critical_warning_source	Ontology Source	Gathers data with critical warning class
critical_warning_sink	Mqtt Sink	Sends data to critical warning topic

Table 6.13: Nodes of car critical warning flow

A part of the OWL representation for the ontology model is shown in listing 6.1. The ontology model can classify data as a warning or critical warning. It checks time to speed limit field of data to do this classification. As it can be seen, any data with less than 5 time to speed limit value is classified as a critical warning. They are also classified as a warning since the critical warning is a sub-class of warning. If the time to speed limit value is less than 20 but not 5, then data will be classified only as a warning.

Listing 6.1: OWL representation of Critical Warning

```
<owl:Class rdf:about="http://example.com##CriticalWarning">
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty
        rdf:resource="http://example.com##hasTime_to_speed_limitValue"/>
            <owl:someValuesFrom>
                <rdfs:Datatype>
                    <owl:onDatatype
                        rdf:resource="int"/>
                    <owl:withRestrictions rdf:parseType="Collection">
                        <rdf:Description>
```

```
                        <xsd:maxExclusive>
                            5
                        </xsd:maxExclusive>
                    </rdf:Description>
                </owl:withRestrictions>
            </rdfs:Datatype>
        </owl:someValuesFrom>
    </owl:Restriction>
</owl:equivalentClass>
<rdfs:subClassOf rdf:resource="http://example.com##Warning"/>
</owl:Class>
<owl:Class rdf:about="http://example.com##Time_to_speed_limitField">
<rdfs:subClassOf rdf:resource="http://example.com##Field"/>
</owl:Class>
```

Simulations are used during the test of this use-case since real cars could not be used. A simulation of a car generates meaningful car data to be used in the use-case. They randomly accelerate positively or negatively within certain limits. They also change their speed limit values to simulate a car that is changing roads.

The tests are done using 5, 10, 30, 50, 100, and 500 different simulations in parallel. Therefore, an environment that can simulate the system until 500 cars is created. The data frequency for each simulation is changed between 2, 1, 0.5, and 0.2 data per second. EC2 instances with 1 GB RAM, 1 virtual CPU, and 8 GB SSD storage are used to deploy an ontology engine, ontology manager and, query manager as the testing environment.

In figure 6.10, the execution time vs. time graph for 5 virtual devices that are each sending 2 data per second is shown. The rule engine shows promising and stable results for this scenario. It has an average of 5.64 milliseconds flow execution time for the warning system.

Another graph can be seen in figure 6.11 with 50 different virtual cars. These virtual cars are programmed to send data to the flow on every 2 seconds. The flow execution time changes between 5 milliseconds and 66 milliseconds. There is a remarkable difference between the minimum and maximum values overall execution. The different data arrived at the same time causes this difference. The new data needs to wait until the execution of existing data in the flow is ended.

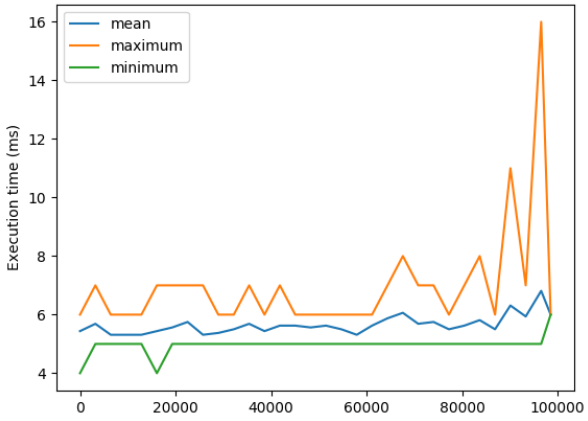


Figure 6.10: Flow execution time for overspeed warning system with 5 devices and frequency 2 s^{-1}

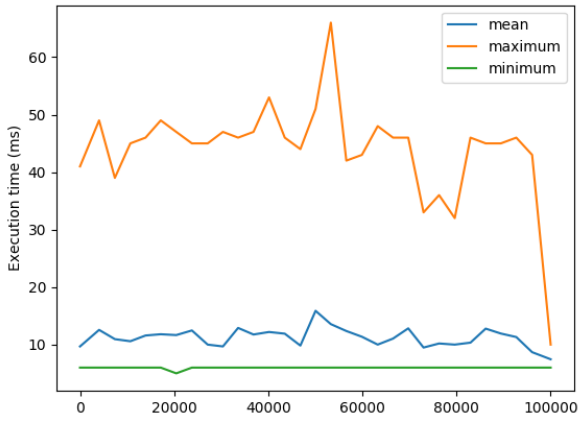


Figure 6.11: Flow execution time for overspeed warning system with 50 devices and frequency 0.5 s^{-1}

When the number of devices is changed to 500 with the data frequency of 1 per second, the execution time grows over time. The race between data that entered flow occasions this likewise the test with 50 virtual devices. Accumulated data over time increase the execution time. The flow execution time is increased by 50 seconds. After

data sending is stopped by virtual devices at the end of the test, it starts to decrease. Because of accumulated data pool got smaller.

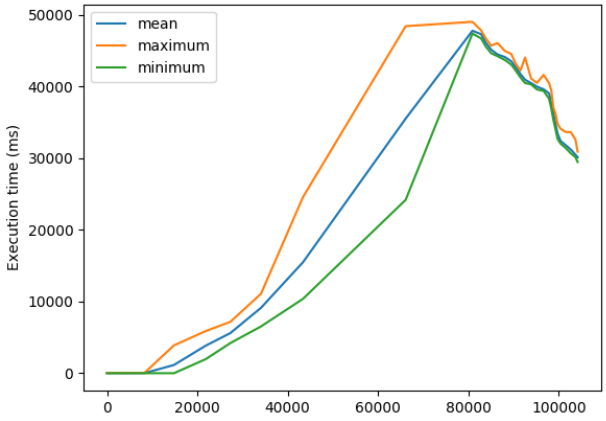


Figure 6.12: Flow execution time for overspeed warning system with 500 devices and frequency 1 s^{-1}

In table 6.14, the summary of all tests that are with the overspeed warning system for connected cars use case is indicated. The flows have no problem with processing 2 data per second until 100 different cars. It can even process data of 500 different cars when they send data every 5 seconds.

Number of Simulations	Frequency	Minimum	Maximum	Mean
5	2	4 ms	16 ms	5.64 ms
10	2	5 ms	47 ms	8.15 ms
30	2	5 ms	57 ms	18.20 ms
50	2	6 ms	338 ms	23.3668 ms
100	2	6 ms	89 ms	22.16 ms
500	2	6 ms	35208 ms	25756.42 ms
5	1	7 ms	133 ms	10.66 ms
10	1	6 ms	52 ms	12.966 ms
30	1	8 ms	56 ms	15.91 ms
50	1	8 ms	189 ms	21.06 ms
100	1	8 ms	60 ms	20.70 ms
500	1	8 ms	49044 ms	31399.83 ms
5	0.5	10 ms	21 ms	11.18 ms
10	0.5	10 ms	22 ms	11.84 ms
30	0.5	11 ms	154 ms	15.76 ms
50	0.5	5 ms	66 ms	11.32 ms
100	0.5	5 ms	333 ms	16.35 ms
500	0.5	3 ms	75049 ms	34204.03 ms
5	0.2	4 ms	6 ms	5.04 ms
10	0.2	7 ms	109 ms	7.03 ms
30	0.2	5 ms	125 ms	7.11 ms
50	0.2	6 ms	136 ms	7.98 ms
100	0.2	5 ms	46 ms	12.08 ms
500	0.2	4 ms	4361 ms	271.59 ms

Table 6.14: Performance results of overspeed warning system

Every flow can only run in a single machine and cannot be distributed. Therefore, horizontal scaling does not help to improve these results. Nonetheless, there are few options to increase the performance to support more devices. The first one is to scale instance with the flow vertically. The second option is to create identical ontologies with a load balancing ontology ahead. The logical ontologies should not be reachable by the real world devices but only load balancing ontology. The load balancing ontology should redirect all data the logical ontologies with any custom load balancing algorithm.

7 Future Works

The current implementation of the rule engine establishes a practical environment for data processing. However, there is still more space for it to grow. With future work to be held, it can support more technologies and will not lock users to work with few technology options. Furthermore, it can also provide more tools to effectively serve other needs of data processing.

In this chapter, the critical prospective works for the improvement of the rule engine will be listed and discussed.

7.1 Dockerization

The rule engine consists of different sub-systems. Management and setup of every sub-system can be tricky for some users. Therefore, the dockerization of every sub-system is crucially important. The whole system shall also be dockerized into only one container. Thus, the management will be simpler for users who want to keep everything in the same machine instance.

7.2 Auto-scaling

Ontology manager is not stateless, and flows cannot be shared between two computers. Therefore, generic auto-scaling technologies of cloud vendors cannot be used directly.

On the other hand, the query manager is aware of the load of each system since it handles load balancing. Therefore, the query manager can also manage the scaling of sub-systems automatically. It can launch new instances when the load of existing instances is too high.

7.3 Support for More Communication Protocols

The rule engine only supports MQTT for data receiving and sending. However, there are much more different communication protocols that are used in IoT. Thus, more protocols should be supported by the flows. The future protocols that must be supported by the flows can be listed as;

- CoAP
- HTTP
- Kafka
- Websocket
- LwM2M

7.4 Time-Scheduled Flow Triggers

Flows in the rule engine can only run when a data received by its source nodes. The current source nodes restrain to create flows that run on a scheduled time. In other words, the flows cannot be programmed to create daily summaries from the historical data at midnight. Flows shall be programmable to run on a scheduled time to handle such cases.

7.5 Practical Environment for Ontology Model Creation and Modification

The users are expected to create and modify their custom ontology models by using third-party tools. The support for viewing and modifying ontology models shall be added to both the web application and IoTeQL. It should also provide practical ways for modification of the models. Therefore, users can modify their ontology models without expert knowledge.

7.6 Semantic Queries to Historical Data

The semantic ontologies are only used for on-demand data classification. The usage of the semantic ontologies can be extended to gather more meaningful information using both historical data and ontology models. The flows can be used to query historical data by using IoTeQL and SPARQL.

7.7 Better Environment for Node Development

The open-source platforms similar to the rule engine are powered by huge communities as can be seen in chapter 5. The node development should be simplified to enable a

development environment for the practical rule engine. The developers shall be able to integrate their custom nodes to all sub-systems easily. Therefore, a library consists of custom nodes, developed by the community, can be created.

7.8 Development Tools

The rule engine provides a practical environment for development with its web application and IoTeQL. Nevertheless, the development environment can be improved further. IoTeQL syntax highlighters for the popular text editors and command-line tools to manage both ontologies and flows shall be developed.

7.9 Integration with Related Works

In chapter 5, popular related platforms has been presented. The users might want to use the rule engine together with the features of these platforms. Thence, the integration modules for these platforms shall be published in the libraries of the platforms.

List of Figures

List of Tables

Bibliography

[1] G. Suciu, S. Halunga, A. Vulpe, and V. Suciu, "Generic platform for iot and cloud computing interoperability study," in *International Symposium on Signals, Circuits and Systems ISSCS2013*, July 2013, pp. 1–4.

[2] M. G. Kibria, M. A. Jarwar, S. Ali, S. Kumar, and I. Chong, "Web objects based energy efficiency for smart home iot service provisioning," in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, July 2017, pp. 55–60.

[3] N. Lee and H. Lee, "Device objectification for internet of things services," in *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, June 2014, pp. 1–2.

[4] S. Shukla and K. Sornalakshmi, "Internet of things: Rule based event management," *International Journal of Science and Research (IJSR)*, vol. 4, no. 4, pp. 1214–1216, Apr 2015.

[5] S. Seo, S. Chun, B. Oh, and K. Lee, "Sdpa: Sensor data processing architecture for modeling semantic data from sensor steams," in *2015 IEEE International Conference on Information Reuse and Integration*, Aug 2015, pp. 9–16.

[6] D. Park, H. Bang, C. S. Pyo, and S. Kang, "Semantic open iot service platform technology," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, March 2014, pp. 85–88.

[7] X. Zhang, Y. Zhao, X. Wang, and D. Pan, "An approach to provide visual data service for heterogeneous sensor data based on ssn ontology," in *2015 International Conference on Identification, Information, and Knowledge in the Internet of Things (IIKI)*, Oct 2015, pp. 254–257.

[8] C. E. Kaed, I. Khan, A. Van Den Berg, H. Hossayni, and C. Saint-Marcel, "Sre: Semantic rules engine for the industrial internet-of-things gateways," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 2, pp. 715–724, Feb 2018.

[9] JS Foundation. (2019) About Node-RED. [Online]. Available: https://nodered.org/about/

[10] ——. (2019) Node-RED : Documentation. [Online]. Available: https://nodered.org/docs/platforms

[11] ——. (2019) Node-RED library. [Online]. Available: https://flows.nodered.org

[12] openHAB Foundation. (2019) Who we are | openHAB. [Online]. Available: https://www.openhab.org/about/who-we-are.html

[13] ——. (2019) openHAB. [Online]. Available: https://www.openhab.org/

[14] ——. (2019) Add-ons | openHAB. [Online]. Available: https://www.openhab.org/addons/

[15] ——. (2019) Rules | openHAB. openHAB Foundation. [Online]. Available: https://www.openhab.org/docs/configuration/rules-dsl.html