

Electronic Thesis and Dissertation Repository

---

5-27-2019 2:00 PM

## Virtual Sensor Middleware: A Middleware for Managing IoT Data for the Fog-Cloud Platform

Fadi AlMahamid  
*The University of Western Ontario*

Supervisor  
Lutfiyya, Hanan  
*The University of Western Ontario*

Graduate Program in Computer Science  
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science  
© Fadi AlMahamid 2019

Follow this and additional works at: <https://ir.lib.uwo.ca/etd>



Part of the [Other Computer Sciences Commons](#), and the [Systems Architecture Commons](#)

---

### Recommended Citation

AlMahamid, Fadi, "Virtual Sensor Middleware: A Middleware for Managing IoT Data for the Fog-Cloud Platform" (2019). *Electronic Thesis and Dissertation Repository*. 6221.  
<https://ir.lib.uwo.ca/etd/6221>

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact [wlsadmin@uwo.ca](mailto:wlsadmin@uwo.ca).

## Abstract

Internet of Things is a massively growing field where billions of devices are connected to the Internet using different protocols and produce an enormous amount of data. The produced data is consumed and processed by different applications to make operations more efficient. Application development is challenging, especially when applications access sensor data since IoT devices use different communication protocols.

The existing IoT architectures address some of these challenges. This thesis proposes an IoT Middleware that provides applications with the abstraction required of IoT devices while distributing the processing of sensor data to provide a real-time or near real-time response and enable the applications to choose from where to consume sensor data. The suggested middleware architecture minimizes the development efforts required by the applications by automating the processing of sensor data on multiple nodes (fog nodes) deployed near IoT devices and making it configurable. Furthermore, the dissemination of sensor data using the publish-subscribe paradigm makes it easier for applications to decide from where to consume sensor data while maintaining decoupling from IoT devices.

**Keywords:** Internet of Things, IoT Middleware Architecture, Cloud Computing, Fog Computing, Fog-Cloud Platform, Publish-Subscribe Paradigm, Virtual Sensor.

## Acknowledgments

First and foremost, I would like to express my heartiest gratitude to the Almighty God (Allah), the most gracious, and the most merciful, for providing me with the ability, knowledge and patience to accomplish this thesis successfully.

I would like to express my sincere gratitude to my supervisor Prof. *Hanan Lutfiyya* for all the support she provided me to accomplish this work. It was a great and enjoyable journey with her. I have learned from her a lot and words are powerless to express my gratitude for her guidance, patience, and motivation. Without her support, this work could not have been accomplished.

I would also thank my late father *Lorans AlMahamid*, who encouraged me always to learn and pursue further my studies. I would thank my mother *Afaf AlMahamid*, for the care, support, and encouragement she provided me in my entire life. I would like to thank my lovely wife *Muntaha Muhaidat* for all the support she gave me to stand up and accomplish my goals in life.

Last but not least, I would like to thank my brothers, sisters, teachers, and friends which no words would be sufficient enough to describe my gratitude to them.

## **Dedication**

To the memory of my late father **Lorans AlMahamid**.

To my mother **Afaf AlMahamid**.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgments</b>	<b>ii</b>
<b>Dedication</b>	<b>iii</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Appendices</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.1.1 Internet Of Things . . . . .	1
1.1.2 Wireless Sensors Network and Protocols . . . . .	1
1.1.3 Cloud Computing . . . . .	2
1.1.4 Fog Computing . . . . .	2
1.2 Problem Statement . . . . .	3
1.3 Thesis Objective . . . . .	5
1.4 Thesis Outline . . . . .	5
<b>2 Related Work</b>	<b>6</b>
2.1 Virtual Sensors . . . . .	6
2.1.1 Related Work . . . . .	7
2.2 CoAP Proxy . . . . .	7
2.3 DPWS . . . . .	12
2.3.1 DPWS for IoT devices . . . . .	12
2.3.2 DPWS Gateways for Accessing WSN . . . . .	15
2.4 Middleware . . . . .	18
2.4.1 Middleware-IoT Devices communication . . . . .	19
Adapters/Connectors/Wrappers . . . . .	19
Publish/Subscribe . . . . .	20
2.4.2 Related Work . . . . .	21
2.5 Gap Analysis . . . . .	22
<b>3 Virtual Sensors Middleware Architecture</b>	<b>24</b>

3.1	Virtual Sensor . . . . .	24
3.2	Middleware Communication . . . . .	25
3.2.1	Disseminating Data . . . . .	25
3.2.2	Publish-Subscribe communication patterns . . . . .	26
3.2.3	Publish-Subscriber Topologies . . . . .	26
3.2.4	Why Federated Architecture? . . . . .	29
3.2.5	Virtual Sensor Communication . . . . .	29
3.3	Virtual Sensor Deployment Structure . . . . .	30
3.4	Middleware Components . . . . .	31
3.4.1	Middleware Fog Components . . . . .	32
3.4.2	Middleware Cloud Components . . . . .	32
3.4.3	Virtual Sensor Components . . . . .	33
	Virtual Sensor Configurations . . . . .	33
	Components . . . . .	34
3.5	Middleware Components Interaction Scenarios . . . . .	35
3.5.1	Creating Virtual Sensor Configurations . . . . .	35
3.5.2	Instantiating Virtual Sensor Configurations . . . . .	36
3.5.3	Exchanging Messages Between Virtual Sensors . . . . .	37
3.6	UML Diagram . . . . .	38
<b>4</b>	<b>Virtual Sensor Middleware Implementation</b>	<b>41</b>
4.1	Platform . . . . .	41
4.2	Development Tools . . . . .	41
4.3	Middleware Cloud Components . . . . .	41
4.3.1	Virtual Sensor Configurator Interface . . . . .	41
4.3.2	Virtual Sensor Configurator . . . . .	43
4.3.3	Virtual Sensor Deployer . . . . .	43
4.3.4	Virtual Sensor Configurations . . . . .	43
4.4	Middleware Fog Components . . . . .	47
4.4.1	Publish-Subscribe Message Broker . . . . .	47
4.4.2	Knowledge-Base . . . . .	47
4.4.3	Database . . . . .	48
4.4.4	Virtual Sensor Orchestrator . . . . .	48
4.4.5	Virtual Sensor Container . . . . .	48
4.4.6	Virtual Sensor Libraries . . . . .	48
	RabbitMQ Java Client Library . . . . .	48
	Java UUID Generator (JUG) . . . . .	49
	Apache Log4J . . . . .	49
	JSR 374 (JSON Processing) . . . . .	49
4.5	Implementation Classes . . . . .	50
4.5.1	VS Aggregator . . . . .	50
4.5.2	Publisher . . . . .	50
4.5.3	Consumer . . . . .	51
4.5.4	VS Orchestrator . . . . .	52

<b>5</b>	<b>Evaluation</b>	<b>53</b>
5.1	Evaluation Environment . . . . .	53
5.2	Evaluation Factors and Metrics . . . . .	54
5.2.1	Evaluation Factors . . . . .	54
5.2.2	Evaluation Metrics . . . . .	55
5.3	Baseline . . . . .	56
5.4	Evaluation Scenarios . . . . .	57
5.4.1	Scenario 1 . . . . .	58
5.4.2	Scenario 2 . . . . .	60
5.4.3	Scenario 3 . . . . .	62
5.4.4	Scenario 4 . . . . .	65
5.4.5	Scenario 5 . . . . .	68
5.4.6	Scenario 6 . . . . .	70
5.5	Discussion of Results . . . . .	72
<b>6</b>	<b>Conclusion and Future Work</b>	<b>75</b>
6.1	Conclusion . . . . .	75
6.2	Future Work . . . . .	76
	<b>Bibliography</b>	<b>78</b>
<b>A</b>	<b>Virtual Sensor Configuration File</b>	<b>82</b>
<b>B</b>	<b>Reading Evaluations Tables and Figures</b>	<b>84</b>
B.1	Reading Evaluation Table . . . . .	84
B.2	Box and Whisker Plot . . . . .	85
	<b>Curriculum Vitae</b>	<b>87</b>

# List of Figures

1.1	Applications accessing IoT devices using different protocols[1]	2
1.2	Cloud Computing	3
1.3	Fog Computing	3
1.4	IoT Architectures focus on disseminating data vs. processing data	4
2.1	Global Sensors Network Architecture [1]	8
2.2	Applications Communications with CoAP WSN [9]	8
2.3	Protocol Stack adaption between CoAP and HTTP carried through Proxy	9
2.4	CoRE Resource Discovery [9]	10
2.5	DPWS Protocol Stack [23]	13
2.6	WS-Discovery used by DPWS	14
2.7	WS-Eventing used between IoT Devices	15
2.8	implemented layers of the OSI model by TCP/IP, 6LoWPAN, and ZigBee protocols	16
2.9	IP Based Wireless Sensor Node	16
2.10	DPWS Gateway connecting IP-based applications to 6LoWPAN WSN [41]	17
2.11	DPWS Gateway Sequence Diagram [23]	18
2.12	IoT middleware communication channels	19
2.13	Middleware-IoT devices communication using adapters	20
2.14	Middleware Publish/Subscribe	20
2.15	FIWARE IoT Device Management GE architecture [15]	21
2.16	Senaas functional architecture [3]	21
2.17	Wireless Sensor Networks with Publish-Subscribe Communication [3]	22
3.1	Virtual Sensor Abstraction	25
3.2	Virtual Sensor Input Sources	25
3.3	Centralized	27
3.4	Clustered	27
3.5	Federated	28
3.6	Peer-to-Peer	28
3.7	Federated Clusters	29
3.8	VSM Middleware Communications	30
3.9	Virtual Sensors Physical Deployment	30
3.10	Virtual Sensors conceptual deployment	31
3.11	Virtual Sensor Conceptual Deployment: Acyclic Graph	31
3.12	VS Middleware Architecture	32
3.13	Virtual Sensor configurations vs. running virtual sensor	34



3.14	Virtual Sensor Components . . . . .	35
3.15	Creating Virtual Sensor Configurations . . . . .	36
3.16	Instantiating Virtual Sensor . . . . .	37
3.17	Exchange Messages Between Virtual Sensors . . . . .	39
3.18	UML Diagram . . . . .	40
4.1	VS Configurator GUI - Virtual Sensor Information . . . . .	42
4.2	VS Configurator GUI - Deployment Node . . . . .	43
4.3	VS Configurator GUI - Publish Information . . . . .	44
4.4	VS Configurator GUI - Subscribe Information . . . . .	45
4.5	VS Configurator GUI - Generated Configuration Ready For Deployment . . . . .	46
4.6	RabbitMQ Simple Publisher/Consumer Architecture . . . . .	48
5.1	Evaluation Environment . . . . .	54
5.2	Evaluation Environment - Creating Configurations . . . . .	54
5.3	Evaluation Environment - Load Testing . . . . .	54
5.4	Baseline - Experiment 1 - Performance Evaluation for Running Raspberry Pi without the middleware . . . . .	57
5.5	Baseline - Experiment 2 - Performance Evaluation for Running Raspberry Pi with the Middleware . . . . .	57
5.6	Baseline - Performance Comparison . . . . .	57
5.7	Scenario 1 - Experiment 1 - Frequency Performance Evaluation . . . . .	60
5.8	Scenario 1 - Experiment 2 - Frequency Performance Evaluation . . . . .	60
5.9	Scenario 1 - Experiment 3 - Frequency Performance Evaluation . . . . .	60
5.10	Scenario 1 - Experiment 4 - Frequency Performance Evaluation . . . . .	61
5.11	Scenario 1 - Experiment 5 - Frequency Performance Evaluation . . . . .	61
5.12	Scenario 1 - Frequency Performance Comparison . . . . .	61
5.13	Scenario 2 - Experiment 1 - Number of Inputs per Virtual Sensor Performance Evaluation . . . . .	63
5.14	Scenario 2 - Experiment 2 - Number of Inputs per Virtual Sensor Performance Evaluation . . . . .	63
5.15	Scenario 2 - Experiment 3 - Number of Inputs per Virtual Sensor Performance Evaluation . . . . .	63
5.16	Scenario 2 - Experiment 4 - Number of Inputs per Virtual Sensor Performance Evaluation . . . . .	64
5.17	Scenario 2 - Experiment 5 - Number of Inputs per Virtual Sensor Performance Evaluation . . . . .	64
5.18	Scenario 2 - Number of Inputs per Virtual Sensor Performance Comparison . . . . .	64
5.19	Scenario 3 - Experiment 1 - Number of Virtual Sensors Performance Evaluation . . . . .	64
5.20	Scenario 3 - Experiment 2 - Number of Virtual Sensors Performance Evaluation . . . . .	66
5.21	Scenario 3 - Experiment 3 - Number of Virtual Sensors Performance Evaluation . . . . .	66
5.22	Scenario 3 - Experiment 4 - Number of Virtual Sensors Performance Evaluation . . . . .	66
5.23	Scenario 3 - Experiment 5 - Number of Virtual Sensors Performance Evaluation . . . . .	67
5.24	Scenario 3 - Number of Virtual Sensors Performance Comparison . . . . .	67
5.25	Scenario 4 - Experiment 1 - Number of Levels Performance Evaluation . . . . .	68

5.26	Scenario 4 - Experiment 2 - Number of Levels Performance Evaluation . . . . .	68
5.27	Scenario 4 - Experiment 3 - Number of Levels Performance Evaluation . . . . .	68
5.28	Scenario 4 - Performance Comparison . . . . .	69
5.29	Scenario 5 - Experiment 1 - Performance Evaluation for 150 VS Running on Single Node . . . . .	69
5.30	Scenario 5 - Experiment 2 - Performance Evaluation for 150 VS distributed into tow nodes . . . . .	71
5.31	Scenario 5 - Experiment 3 - Performance Evaluation for 150 VS distributed into three nodes . . . . .	72
5.32	Scenario 6 - Experiment 1 - Frequency Performance Evaluation of 111 Virtual Sensors . . . . .	73
5.33	Scenario 6 - Experiment 2 - Frequency Performance Evaluation of 111 Virtual Sensors . . . . .	73
5.34	Scenario 6 - Experiment 3 - Frequency Performance Evaluation of 111 Virtual Sensors . . . . .	74
5.35	Scenario 6 - Experiment 4 - Frequency Performance Evaluation of 111 Virtual Sensors . . . . .	74
5.36	Scenario 6 - Frequency Performance Comparison for 111 Virtual Sensors . . . .	74
B.1	Evaluation Table Parts . . . . .	84
B.2	Evaluation Table Part 1 . . . . .	85
B.3	Evaluation Table Part 2 . . . . .	85
B.4	Box and Whisker Plot Explanation . . . . .	85

# List of Tables

4.1	UUID Structure . . . . .	49
4.2	UUID Breakdown . . . . .	49
5.1	Performance Baseline . . . . .	56
5.2	Evaluation Scenarios . . . . .	58
5.3	Impact of the frequency over the performance . . . . .	59
5.4	Impact of number of inputs per virtual sensor over the performance . . . . .	62
5.5	Impact of the number of virtual sensors over performance . . . . .	65
5.6	Impact of the number of levels over performance . . . . .	67
5.7	Impact of distribute processing over the performance . . . . .	69
5.8	Impact of the frequency using 111 virtual sensors over the performance . . . . .	71

# List of Appendices

Appendix A: Virtual Sensor Configuration File . . . . .	82
Appendix B: Reading Evaluations Tables and Figures . . . . .	84

# Chapter 1

## Introduction

In this chapter, section 1.1 provides background and section 1.2 describes the problem statement, where section 1.3 describes the objective of thesis and section 1.4 shows the thesis outline.

### 1.1 Background

#### 1.1.1 Internet Of Things

The Internet of Things (IoT) is a massively growing field where the anticipated number of devices by 2020 could range from 38 billion to 212 billion [2][37]. These devices can exist anywhere in our home, workplace, or city.

An IoT device is an electronic device that can connect to the Internet and interact with applications or other IoT devices in a specific context. For example, a simple form of an IoT device can be a temperature sensor that sends data to a weather application, or it might be a sensor that measures road traffic and sends the data to an application or actuator to control the light signal. It can also be a smartphone or wearable device used to track the steps and body activities. The IoT devices vary in terms of processing power, hardware used, power consumption, and underlying protocol used. For example some of the IoT devices work on a battery, while others work on a continuous power source. However, what they have in common is IoT devices communicate with other IoT devices and/or applications using an underlying protocol.

IoT devices are used in many domains such as smart cities, manufacturing, healthcare, logistics and transportation, automotive, buildings, and home. So that tasks may be more efficient and accurate, which helps to save time and money, and to provide more convenience to the users of the IoT devices.

#### 1.1.2 Wireless Sensors Network and Protocols

Wireless Sensor Networks (WSN)s consist of heterogeneous devices called things. Most of these devices need to reduce energy consumption since they might operate on batteries. HTTP can be used by IoT devices but IoT devices have limited resources and power. Therefore, IoT

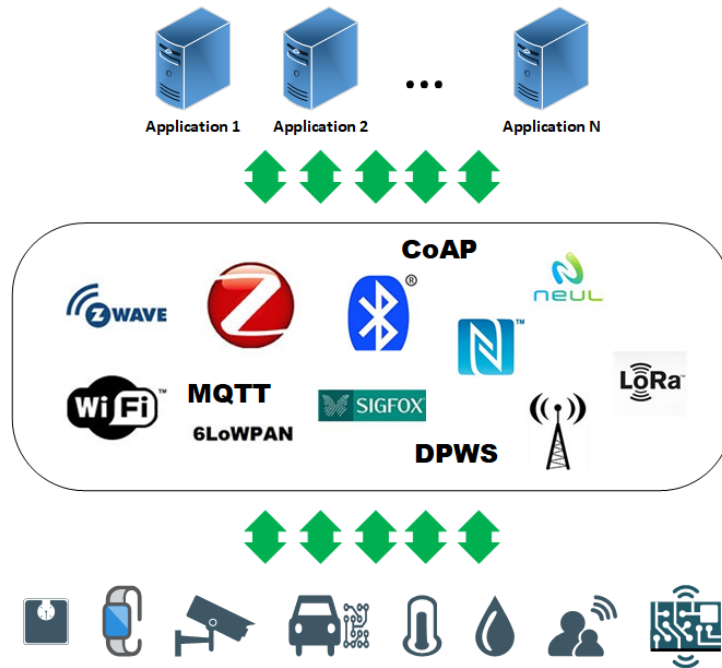


Figure 1.1: Applications accessing IoT devices using different protocols[1]

devices use different protocols for communicating data e.g., CoAP [9], MQTT [22], 6LowPan [30], and ZigBee [4].

### 1.1.3 Cloud Computing

Cloud computing as defined by The National Institute of Standards and Technology (NIST) is “A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [29]”. IoT devices’ data (sensor data) is sent to the cloud for processing and enables the data to be available to different applications. For example temperature readings from sensors distributed in different cities, which then could be made available for multiple applications available for other websites or applications such as *AccuWeather.com* or *Weather.com*.

### 1.1.4 Fog Computing

Fog computing is an extension of cloud computing, where computing resources exist closer to the network edge [10]. Fog computing provides computing resources that are less powerful than the cloud and are deployed at the edge of the network to provide real-time communication or near real-time communication. Fog computing also has advantages such as distributed processing of data.

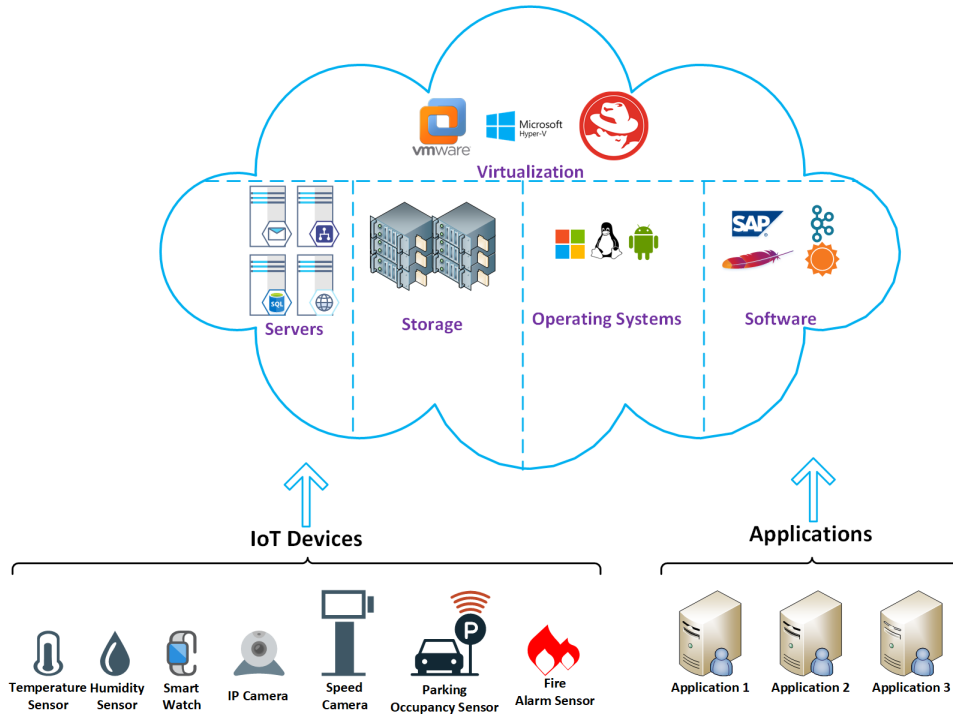


Figure 1.2: Cloud Computing

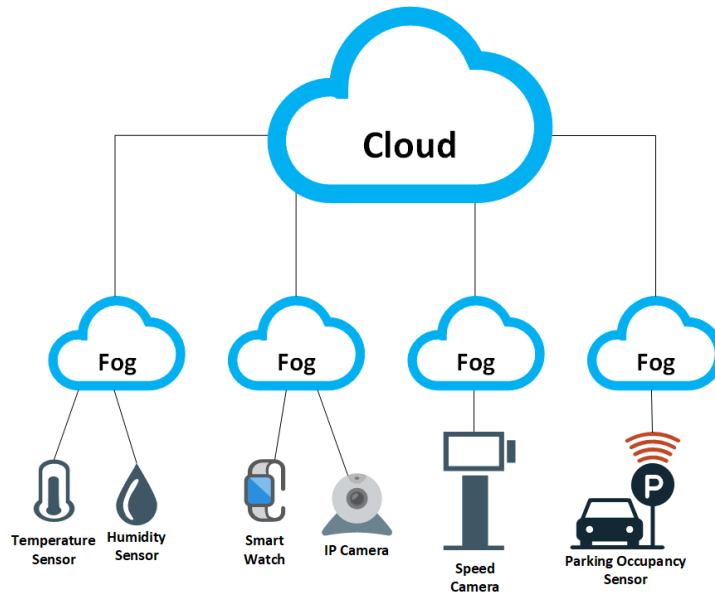


Figure 1.3: Fog Computing

## 1.2 Problem Statement

To deal with the collection of sensor data, existing IoT architectures (e.g., [15][1][9][23]) focus on mechanisms that allow applications to access sensor data without requiring knowledge of the specific communication protocols required by the IoT devices. This typically is done by

using intermediaries that support some form of adaptation. For example, applications use a single protocol to communicate with an intermediary which communicates with a specific IoT device. The protocol between the applications and the intermediary are the same regardless of the protocol used between the intermediary and the IoT device. These intermediaries (or adapters) are the foundations of IoT middleware environments.

Applications often require access to data from multiple IoT devices simultaneously, where each application have different requirements and might need access to a subset of sensor data or require different processing of the same sensor data. In the example of a traffic control system, there may be an application that controls the traffic signals and another application that monitors car speeds on roads. In this case, each application must set its own criteria for collecting and processing sensor data.

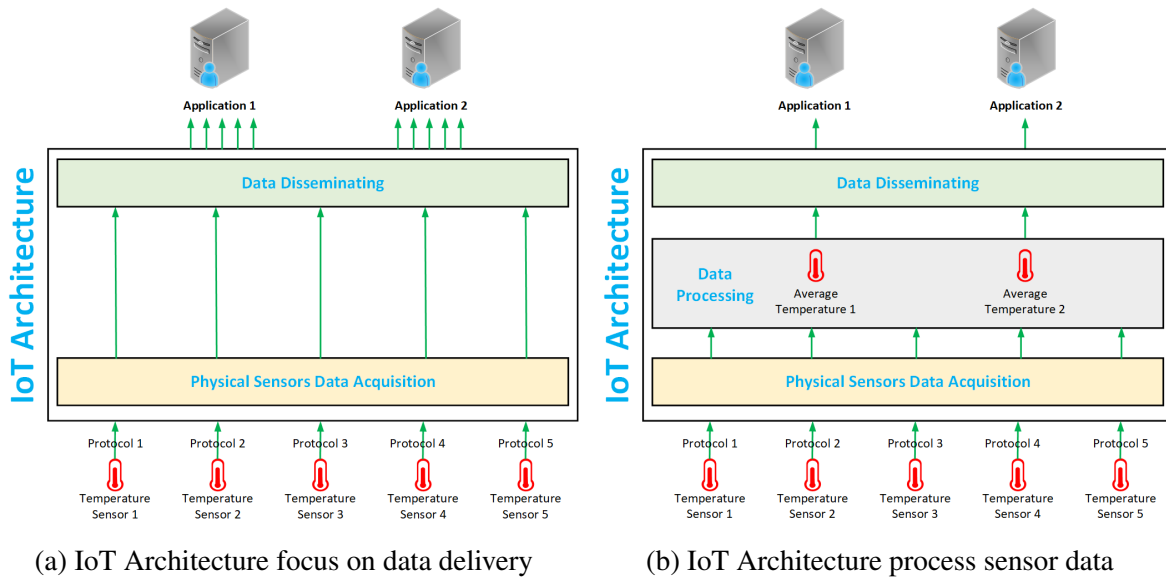


Figure 1.4: IoT Architectures focus on disseminating data vs. processing data

Furthermore, the reviewed work focuses on the delivery of sensor data to applications through the cloud or a single node where all processing is performed. The implication is that either the computing node is overloaded and/or the network core is overloaded. Processing sensor data can not be performed on IoT devices since it has limited processing power and battery life. One approach is to use fog computing where computing nodes are deployed near data sources i.e., IoT devices. The use of fog computing complicates application development. For example, assume an application requires the average temperature. This requires the aggregation of temperature values in order to calculate the average. Some fog nodes may collect temperature data from temperature sensors while another fog node may collect temperature data from the other fog nodes in order to calculate the average temperature. This distribution of tasks among distributed nodes is more complicated than sending all the data to the cloud.

A mechanism is needed to automate the deployment of sensors. Applications require seamless access to processed sensor data without tight integration between applications and IoT devices.



## 1.3 Thesis Objective

This thesis proposes a solution to address the problem described in Section 1.2. The solution combines the use of fog computing along with the use of virtual sensors, and the publish-subscribe design pattern for communication so that data producers (IoT devices) and data consumers (applications) do not need to maintain information about each other. The distributed processing of the sensor data through the use of computing nodes called *fog nodes* that are placed near sensors could minimize latencies in data processing and speed up data processing; therefore, it would provide a real-time or near real-time analysis of sensor data for applications. The use of virtual sensors provides an abstraction from communication protocols. While the use of publish-subscribe design pattern for communication would add another level of abstraction and makes it easier for applications to consume sensor data, since data producers and consumers do not need to maintain information about each other. Only registration to a message broker is needed.

## 1.4 Thesis Outline

The thesis is organized as the following: Chapter 2 described the related work of applications can access sensor data. Chapter 3 describes the architecture of the proposed middleware (Virtual Sensor Middleware) and the interaction between different components. Chapter 4 describes how the middleware was implemented and the used technology. Chapter 5 shows results of the experiments used to measure the middleware performance. Chapter 6 discuss the conclusion and the future work.

# Chapter 2

## Related Work

This chapter describes the different approaches that can be used so that application developers can develop their applications without getting into low-level application programming to communicate with IoT devices that exist within WSN and receive/fetch data from these devices.

### 2.1 Virtual Sensors

A virtual sensor can be considered as a software abstraction of one or more physical sensors that can be deployed on the cloud or near the physical sensor on a fog node. A virtual sensor communicates with other virtual sensors or with one or more physical sensors and are usually deployed as part of a middleware.

Virtual sensors can be classified by communication-direction type with physical sensor(s) as listed below [28]:

1. **One-to-Many:** This is when one physical sensor is connected with many virtual sensors. This is used when multiple clients need to connect to the same physical sensor and have different requirements for the frequency that data is needed. It would be difficult to program/configure a single sensor to send data at different intervals. However, it still can be achieved by extending the sensor through multiple virtual sensors, where each virtual sensor sends the data to the client at its pre-configured interval. Thus, it would make it easier to have a personalized configuration for each client by configuring each virtual sensor independently. Furthermore, it would make it easier to manage the billing individually for each client if it is required.
2. **Many-to-One:** This is useful when there are multiple sensors generating different readings at different intervals or the same interval; The received readings need to be aggregated to produce one single reading. For example, if multiple physical sensors are deployed across the city to provide temperature readings, but the client/application requires the average temperature reading across the city. Therefore, a virtual sensor can be configured to receive the readings from all various physical sensors across the city, and then computes the average and submits the results to the client on a pre-configured interval.

3. Many-to-Many: This is a combination of many-to-one and one-to-many, where the physical sensor might participate in both relations. The physical sensor sends the data to multiple virtual sensors, but it submits the data to another virtual sensor which aggregates the data received from other physical sensors.
4. Derived: In the derived relation, a virtual sensor receives readings from heterogeneous physical sensors but produces a reading derived from other readings, but not of the same reading type. The derived sensors are used in two ways:
  - (a) Simulate a physical sensor e.g., configure a virtual proximity sensor using deployed light sensors
  - (b) Derives or computes a value that cannot be calculated using just one type of sensor. The virtual sensor communicates with different types of physical sensors. It then calculates the value based on various readings. For example, assume we have a food container that has different physical sensors, which provide readings for temperature, humidity, and level of oxygen. These readings cannot be used separately to determine if the environment is safe for food storage. However, a configured virtual sensor, which communicates with physical sensors and then processes the received data using an equation can determine whether the environment is safe for the food storage or not.

### 2.1.1 Related Work

Global Sensor Networks (GSN) emphasizes on the use of virtual sensors for abstraction from applications where it defines two types of virtual sensors, one that uses wrappers corresponding to data received from physical sensors, while the other type corresponds to data received from virtual sensors [1]. Both types are hosted in a container that can be deployed on multiple nodes. Each container hosts one or more virtual sensors, where virtual sensors exchange information across the different nodes [1]. Clients can access GSN through the interface layer which uses the data access layer and data integrity layer to provide authentication and confidentiality [1].

## 2.2 CoAP Proxy

In constrained environments power consumption is an essential factor for IoT devices, especially for devices using batteries, which need to last for years [6]. Thus, devices tend to sleep most of the time and wake when communication of data is needed. However, there are other factors that can reduce the power consumption, such as the protocol used for communication between the devices (Machine-to-Machine) and between the devices and the applications. Constrained Application Protocol or CoAP is a software transfer protocol that was developed and standardized by the Internet Engineering Task Force (IETF), which extends REST architecture style with HTTP to access IoT devices in constrained environments inside Wireless Sensors Network.

Since CoAP extends REST it uses URIs (Universal Resource Indicators) to identify IoT devices. For example, if we are considering a RESTful URI for a resource named R1 that

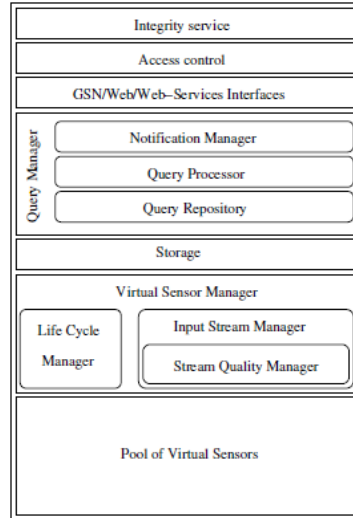


Figure 2.1: Global Sensors Network Architecture [1]

is controlled by a server named example.com, then the URI would look like the following: `http://example.com/R1`. Similarly, CoAP identifies devices using URIs, so if we consider an IoT device named R1, then the URI would look like the following: `coap://node.example.com/R1`.

So, by looking at the links, we see that the two protocols have almost the same URIs. This raises the following question: do applications need to understand CoAP to be able to communicate with IoT device? The answer is yes and no, because there are two different ways applications can use to communicate with IoT devices as illustrated in Figure 2.2.

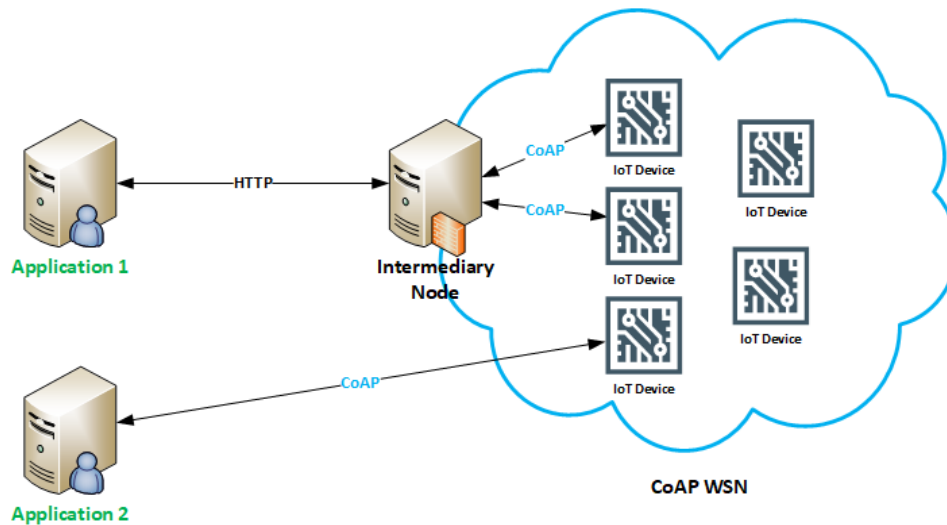


Figure 2.2: Applications Communications with CoAP WSN [9]

1. Invoke IoT URI directly: If the application understands the CoAP protocol, then it can recognize the protocol standards and message formats, which enables the application to communicate directly with IoT devices by making direct CoAP URI requests

- Invoke IoT URI via a proxy: If the application only understands HTTP, then it needs a translation mechanism (adaption) between HTTP and CoAP since the two protocols have different protocol stacks. The translation is carried by an intermediary node called a proxy that understands HTTP from one side and CoAP from the other side. The proxy translates between the two protocols.

Therefore, applications can be developed independently from the CoAP protocol standards, since it does not need to understand CoAP. Figure 2.3 illustrates how a proxy carries out protocol adaption between IoT devices and HTTP applications.

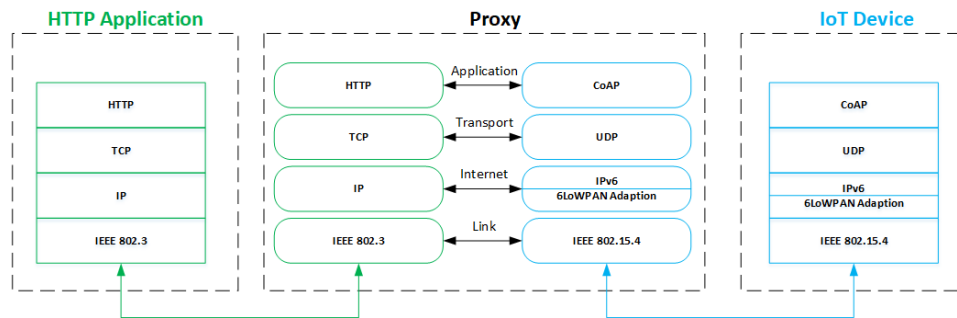


Figure 2.3: Protocol Stack adaption between CoAP and HTTP carried through Proxy

Proxies provide an abstraction similar to virtual sensors that allow application developers to use CoAP devices without needing to know the implementation details related to message formats and CoAP protocol standards. Thus, it promotes a loosely-coupled approach, where the application can be updated independently of CoAP WSN, while IoT devices still have the advantage of using the CoAP WSN.

CoAP WSN supports these features:

- Built-in Resource Discovery:** Machine-to-Machine (M2M) discovery without human interaction is very important in WSN due to the ad-hoc nature of IoT devices, where new IoT devices might join the WSN, while some others might leave. CoAP provides a built-in M2M discovery without the need for human interactions, by providing a well-known link hosted on CoAP Server called "Constrained RESTful Environments (CoRE) Link" as defined in RFC6690 [38], which is an extension of "Web Linking" defined in RFC5988 [31].

CoRE links are mainly used in the discovery process for the resources hosted by CoAP server and its attributes and relations. Each CoAP device (IoT device) registers itself with the CoAP server, where a CoAP server defines a well-known URI `"/.well-known/core"` that is used as a default entry point to discover hosted resources, their attributes, and their relations with other resources, which is known as CoRE Discovery process [38].

- Observer Design Pattern:** In a constrained environments where device power consumption matters, the design pattern and the protocol used makes a difference. In a typical HTTP scenario clients need to perform a GET request to fetch the resource's data. This would drain the resource (IoT device) power, if it is going to respond to each received GET request.

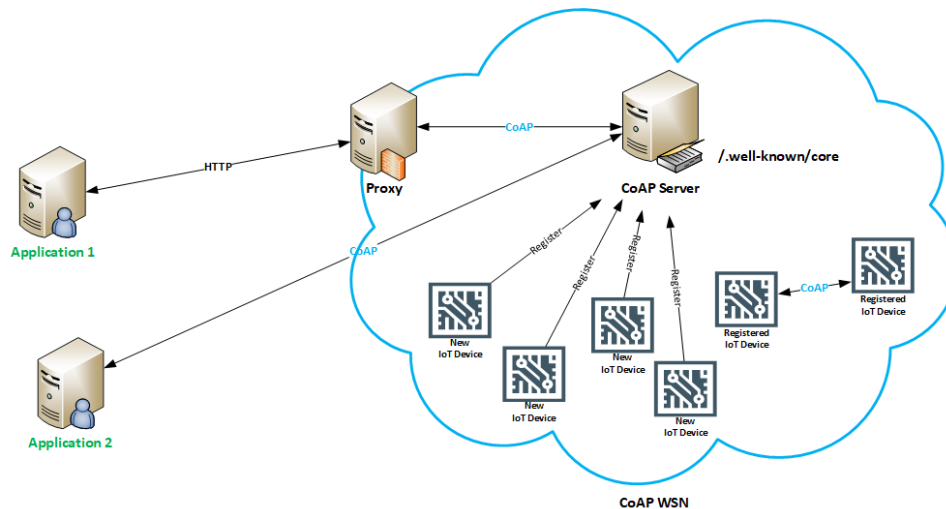


Figure 2.4: CoRE Resource Discovery [9]

CoAP uses an asynchronous approach to overcome this issue [9], where the client application can specify in the first issued GET request, its interest in receiving further updates from the resource, by specifying the "Observe" option. The resource can then sleep when there are no updates, and when it has new updates it will send it to all interested clients. This approach called "Observer design pattern", which is less expensive and more feasible to implement compared to the publish subscribe pattern [9].

3. **Caching:** Similar to the observer design pattern caching helps to reduce the power consumption since the client would reuse the cached responses as long as they are valid rather than making new requests to the IoT device. Two models are used for valid caching [39]:
  - **Freshness Model:** The response is "fresh" and valid to use as long as it has not expired, which is determined by setting the Max-Age option in seconds. The response is considered not valid and can not be used if its age is greater than the value specified in the Max-Age option. The Max-Age option has a default value of 60 seconds.
  - **Validation Model:** If cached responses are not fresh they are no longer valid. A client can give the originating device a chance to use a cached response by sending a request to update the response. The IoT device sets the ETag option if it wishes to update the freshness of the response, which indicates that the response is "valid". In some cases, there might be more than one unfresh response. Therefore, the originating device decides which response to set as "valid".
4. **Built-in Reliability:** The originating device can mark a message as "Confirmable" if it is expecting an "Acknowledgment" message from the receiver. Once the receiver receives "Confirmable" message, then the receiver should reply back by cloning the message identifier and include a response or reply with an empty message. If the originating device does not receive an "Acknowledgment", it can retransmit at exponentially increasing

intervals until it receives an "Acknowledgment" or runs out of attempts [39].

5. Multicasting: CoAP support multicasting - sending information from one or more endpoints to set of endpoints, through a series of unicasts, which involves sending information from one sender to one receiver; endpoints that offer multicast service can be discovered by other endpoints or CoAP applications using multicast service discovery. A multicast request must be Non-confirmable. Therefore, the server decides when to respond back to multicast so it picks a duration to respond back called "Leisure", which depends on the application or can be derived using specific variables. The server then selects a random point of time within the leisure period to send back the unicast response to the multicast request [39].

## 2.3 DPWS

Device Profile for Web Services (DPWS) applies Service-Oriented architecture for device abstraction [23], where each device defines an XML profile, that is used to discover devices and interact with them in a similar way that applications communicate with SOAP web service. IoT devices that need to implement DPWS need to follow SOA interaction patterns [23]:

- Addressing: Each device should have an IP address
- Discovery: Each device can advertise itself so that other devices or applications can discover it. The discovery process involves two parts: The first part has the device advertise itself by sending a multicast message. The second part covers searching for devices, which is done by sending a multicast message to devices, and then waiting to receive a reply (unicast message) from each device that matches the broadcasted search criteria.
- Description: Each device should have a description that can be used by other devices to learn about the device properties, available services, and the associated message format.
- Control: This is the process of invoking a service of another device, by sending a control message.
- Eventing: This is implemented using a publish-subscribe design pattern, where a device's service publishes events, and where other interested devices can register to (subscribe), in order to receive update messages when an event occurs. Subscriptions expire over time, and the registration to an event needs to be renewed if the receiving device is still interested to receive further updates.

The above SOA patterns are used by other technologies that implements SOA architecture, such as SOAP protocol, which uses SOAP messages (XML messages) to exchange information between web services (SOAP Web Services). In fact DPWS extends the SOAP architectural style, which implements "web services protocol stack", which stacks four different protocols: **1.** Service Transport Protocol **2.** XML Messaging Protocol **3.** Service Description Protocol, and **4.** Service Discovery Protocol.

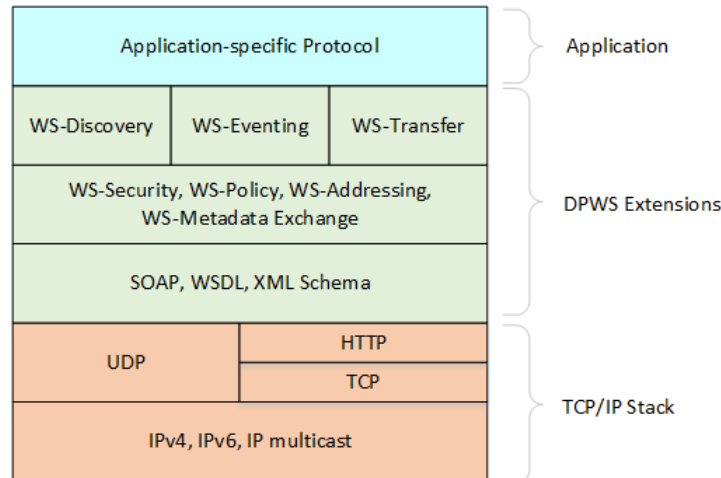
### 2.3.1 DPWS for IoT devices

DPWS and sometimes referred to as is a device-level protocol which is used for constrained devices and used in UPnP [41] and UPnP V2 [23]. DPWS uses a protocols stack as illustrated in Figure 2.5.

Some of the listed DPWS extensions are used by SOAP web services to implement additional features that are not supported by the underlying protocol. The following describes the functionality of each extension used by DPWS:

- WS-Security: This is concerned with message integrity, confidentiality, and reliability rather than channel security. DPWS uses the WSS-SOAP Message Security standard, which can be used with wide security models such as KPI, Kerberos, and TLS/SSL, where it provides support for different token formats, signature formats, multiple trust





**DPWS Protocol Stack**

Figure 2.5: DPWS Protocol Stack [23]

domains, and encryption technologies [23]. WSS-SOAP Message security combines security tokens with digital signatures to insure the integrity and confidentiality of SOAP messages [23].

- **WS-Policy:** This is a flexible and extensible grammar that is used to express different requirements of web services such as security, quality etc. These policies can be defined by the web service provider, or web service consumer. WS-Policy defines policies in collections, where each collection consists of a set of policy assertions. Each policy assertion is used to define a different requirement. For example one policy assertion might define the transport protocol, where another policy assertion might define more critical requirements such as privacy policy or QoS characteristics. These policies are attached with web services using WS-PolicyAttachment, which defines the mechanism on how the WS-Policy is attached to a web service, WSDL artifact, and UDDI element [7].
- **WS-Addressing:** This includes message routing and address information in the SOAP header, which would enable more complex message exchange patterns rather than depending on the underlying protocol for message exchange and routing [23]. WS-Addressing uses asynchronous message exchange, where every resource is referenced using an End-Point Reference (EPR), which consists of (i) Address, and (ii) Reference Properties [23].
  - **Address:** This is the URI that uniquely identifies the device and is resolved to the device physical address
  - **Reference Properties:** These are introduced by the sender and includes:
    - \* **To:** The URI of the message destination
    - \* **Action:** Mandatory attribute which contains URI that identifies the message semantics, and is associated with the WSDL definition.

- \* ReplyTo: This is when a response is expected, or used to route the response to a desired endpoint
  - \* FaultTo: This represents a URI that points to endpoint that receives fault messages
  - \* MessageId: This is a unique identifier of the message that must be specified if a reply is expected
  - \* From: This is an optional attributes that contains the address of the message originator
  - \* RelatesTo: This is mandatory if it is a response message, which would contain the message URI of the related message [11]
- WS-MetadataExchange: This defines how web services metadata included in a web service description can be retrieved and embedded in WS-Addressing endpoint reference. WS-Metadata Exchange is not intended for the purpose of general query or other data associated with a web service [12].
  - WS-Discovery: This is used for device discovery, where the discovered devices are able to declare themselves and their services. WS-Discovery uses an ad-hoc approach to search and locate devices. The discovery process starts when a device wants to join the network, where it needs to declare itself by sending a Hello message to the multicast group using SOAP-over-UDP binding, and thus it helps other devices to detect newly joined devices without the need for continuous probing.

When a client/application wants to access a target device, it multicasts a Probe message using SOAP-over-UDP binding to all devices in the network. Only the target device that matches the Probe message responds with unicast WS-Discovery Probe Match message using SOAP-over-UDP binding to the client [24].

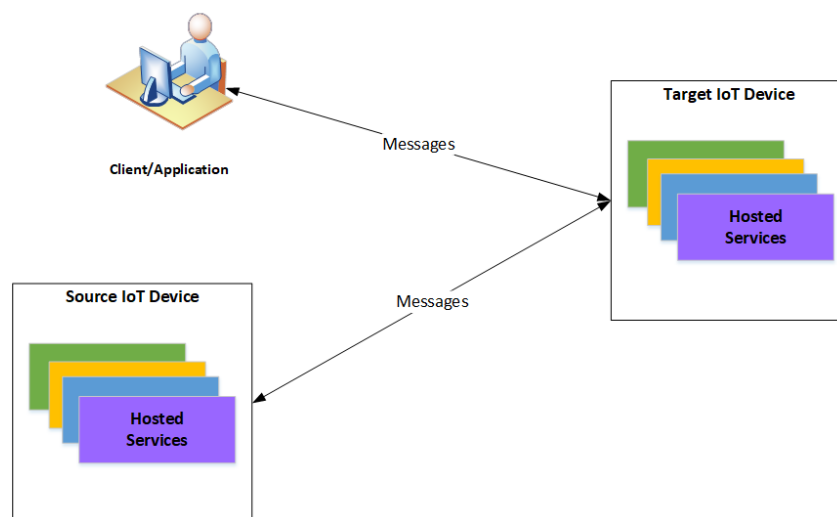


Figure 2.6: WS-Discovery used by DPWS

- **WS-Eventing:** This implements the publish-subscribe design pattern, where if a web service called "A" is interested in another web service called "B", Then "A" would subscribe to "B", which is considered as the "Event Source" and starts sending notifications to "A", which would be considered as an "Event Sink". Notifications received by the Event Sink can be filtered when subscribing to the Event Source, so then the Event source only sends the notification based on the requested filter by the Event Sink [23].

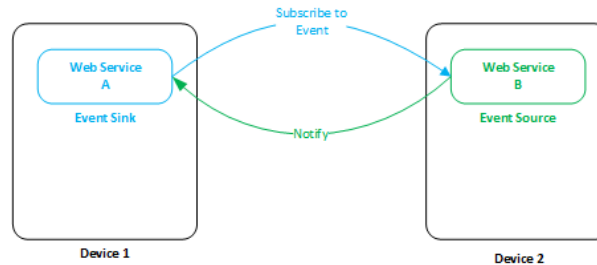


Figure 2.7: WS-Eventing used between IoT Devices

In order to support publish-subscribe design pattern then the WS-Eventing supports three built-in operations [23]:

- **Subscribe:** This is used to subscribe to a service, so the sink web service can start to receive notifications
  - **Renew:** The subscription expires over the time and needs to be renewed
  - **Unsubscribe:** This is used when the web service wishes to receive no more notifications from the event source
- **WS-Transfer:** This is like WS-MetadataExchange except it is used to retrieve all the metadata of an endpoint rather than retrieve partial data

### 2.3.2 DPWS Gateways for Accessing WSN

In order to understand the need of the DPWS gateway [41], we first need to understand the difference between the IP standards used in IP-networks, and the IP standard used by 6LoWPAN WSN. 6LoWPAN supports two types of addresses: either the extended address of IEEE 64-bit, or the short address of 16-bit, and can transfer a maximum frame size of 128-bytes.

IPv6, which is considered as a successor of IPv4, implements 128-bit for the address size, which consists of two parts: i) 64-bit network prefix, and ii) 64-bit hosting address, and can transfer up to 1500-bytes. Further, 6LoWPAN implements PHY, and MAC of the OSI model, compared to IPv6 which follows the TCP/IP protocol implementation as illustrated in Figure 2.8.

Thus, it is necessary to implement an adaption mechanism to enable the communication between IP-based networks and a WSN that implements 6LoWPAN. This adaption is implemented via an adaption layer that is defined by 6LoWPAN, and used by wireless sensors nodes, and is responsible for the following:

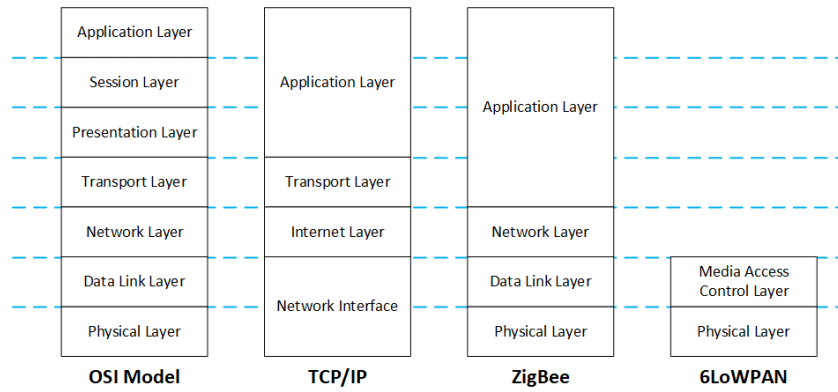


Figure 2.8: implemented layers of the OSI model by TCP/IP, 6LoWPAN, and ZigBee protocols

1. Compress IPv6 header information by disregarding fields that either can be fetched from the source or destination nodes
2. Fragmentation and reassembly of IPv6 packets to meet the minimum MTU of 1280-bytes

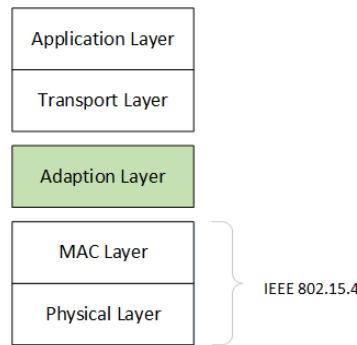


Figure 2.9: IP Based Wireless Sensor Node

Each LoWPAN network has one coordinator, which is identified using PAN\_ID and is used by other nodes to communicate with each other. Since we have the full-picture of the differences between the protocols and why the adaption is required, we can now better understand the solution proposed by [41] which implements DPWS Gateway to enable communication between WSN and IP-based networks as illustrated in Figure 2.10. DPWS Gateway solution consists of the following components:

1. Client: This is the application that wants to communicate with wireless sensors nodes and use their services
2. DPWS Gateway: This is an intermediary between the applications and wireless sensors nodes, which contains a routing table of all sensor nodes registered in the WSN. The routing tables stores WSDL for each node, which is used to describe the services offered by the node, and how it can be triggered. Furthermore, it uses WS-Eventing that allows clients to respond to specific events published by the nodes that enables the application to start receiving notifications from the nodes for registered events

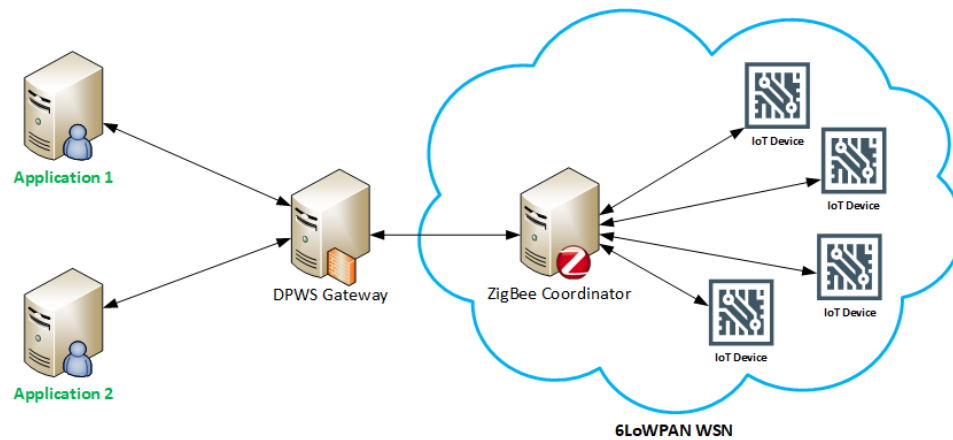


Figure 2.10: DPWS Gateway connecting IP-based applications to 6LoWPAN WSN [41]

3. Wireless Sensor Nodes: Each sensor node that joins WSN has a unique 64-bit ID (EUID-64) that will be registered in DPWS Gateway. This then allows the node to advertise its services using WSDL
4. Coordinator: This is a ZigBee node responsible for packet translation in both directions as illustrated in figure 2.10.

Figure 2.11 graphically depicts how the interaction is carried between sensor nodes and applications using DPWS Gateway

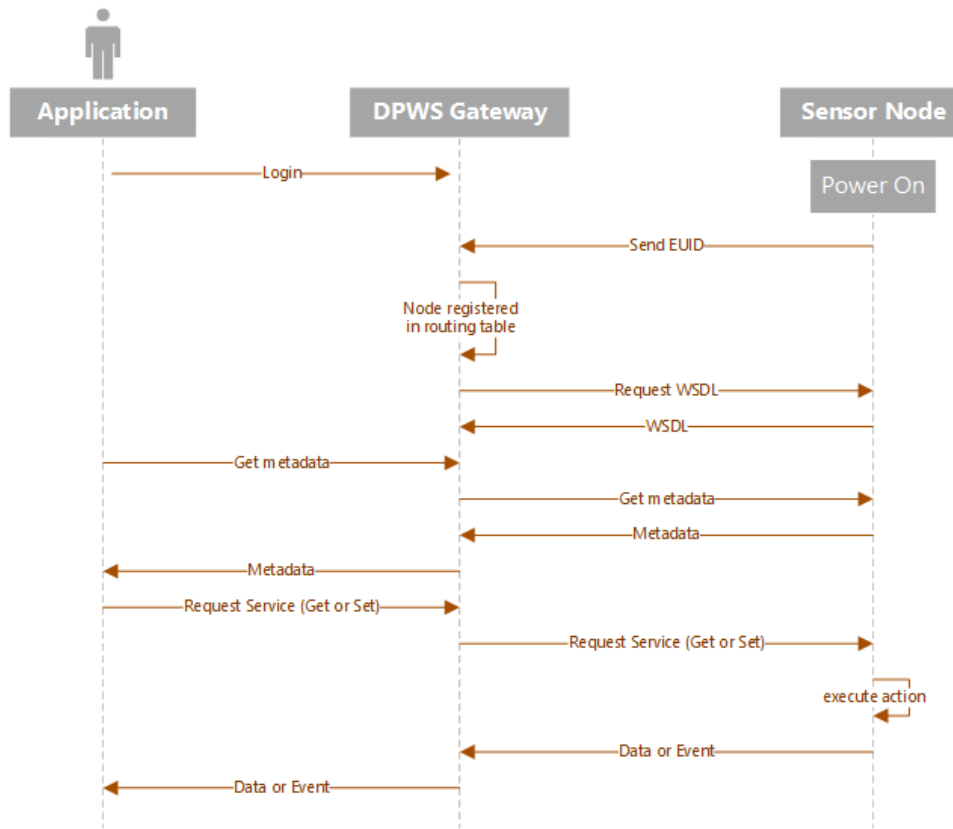


Figure 2.11: DPWS Gateway Sequence Diagram [23]

## 2.4 Middleware

Another approach where applications do not have to communicate with the sensors directly, is through a middleware, which is a software system that facilitates interaction/communication between two or more technological aspects and the applications, which helps minimize development efforts [6].

An IoT middleware serves as an intermediary between the applications and IoT devices, which allows applications to access IoT devices data without the need to manage low level communications with IoT devices or worries about protocol standards. However, the middleware must carry the communication with IoT devices and make the data available to the applications.

Each middleware has its own architecture which implements different techniques to facilitate communications between IoT devices and applications. There are two communication channels that needs to be addressed as illustrated in figure 2.12.

1. middleware-iot devices communication
2. middleware-application communication

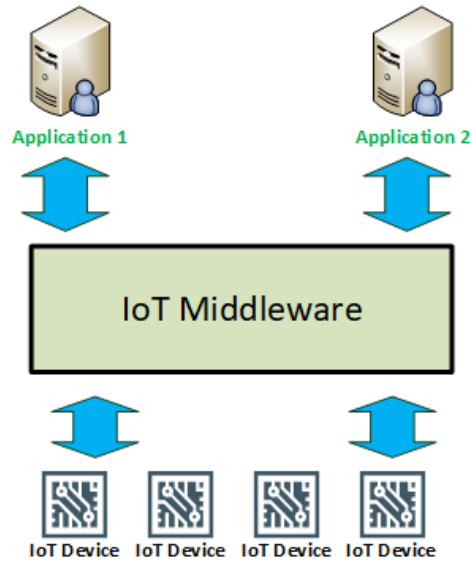


Figure 2.12: IoT middleware communication channels

### 2.4.1 Middleware-IoT Devices communication

The data generated by the physical sensors (IoT devices) can either be sent by the physical sensor or fetched by the middleware, where in the first case the middleware would act as a listener and wait the data to be sent by the sensor, where in the second approach the middleware will be the actor and it fetches the data from the IoT device.

#### Adapters/Connectors/Wrappers

In some cases, the middleware needs to have more control over the frequency that it retrieves data from the IoT device. The middleware should be able to establish a connection with IoT device. The middleware communicates with the operating system (OS) of the IoT device. However, each IoT device might use a different operating systems, and the each operating systems (OS) might be customized to support specific hardware functions of the IoT device. Thus, the middleware should be able to interact with different operating systems to support various IoT devices.

Instead middlewares rely on software modules that communicate directly with different IoT devices. These modules go by different names but implies the same function, which is interacting with IoT devices. These modules are called connectors, adapters, or wrappers. A middleware uses adapters for the most common operating systems used by IoT devices, such as TinyOS, Snappy, and Android Things. However, the middleware should be flexible enough so that the new adapters can be added anytime to support more IoT devices.

An example of a middleware that uses wrappers to collect data from physical sensors is Linked Stream Middleware (LSM) that defines wrappers in the data acquisition layer which is responsible for collecting data from various sensor sources to provide unified output format used by the LSM [26].

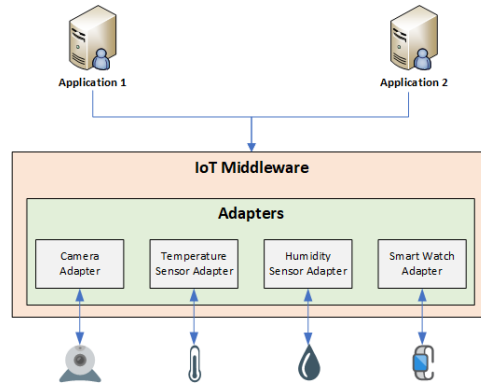


Figure 2.13: Middleware-IoT devices communication using adapters

### Publish/Subscribe

Publish/Subscribe is a design pattern where the IoT device is a publisher that sends the data to subscribers who are interested in the data. On the other hand, the middleware has to subscribe to the IoT device to receive the published data. This approach allows the IoT device to determine when to send (publish) the data and how frequent, which helps to conserve battery power and hence the IoT device goes into sleep mode if it does not have any updates. The delivery of data to the middleware can happen directly or indirectly.

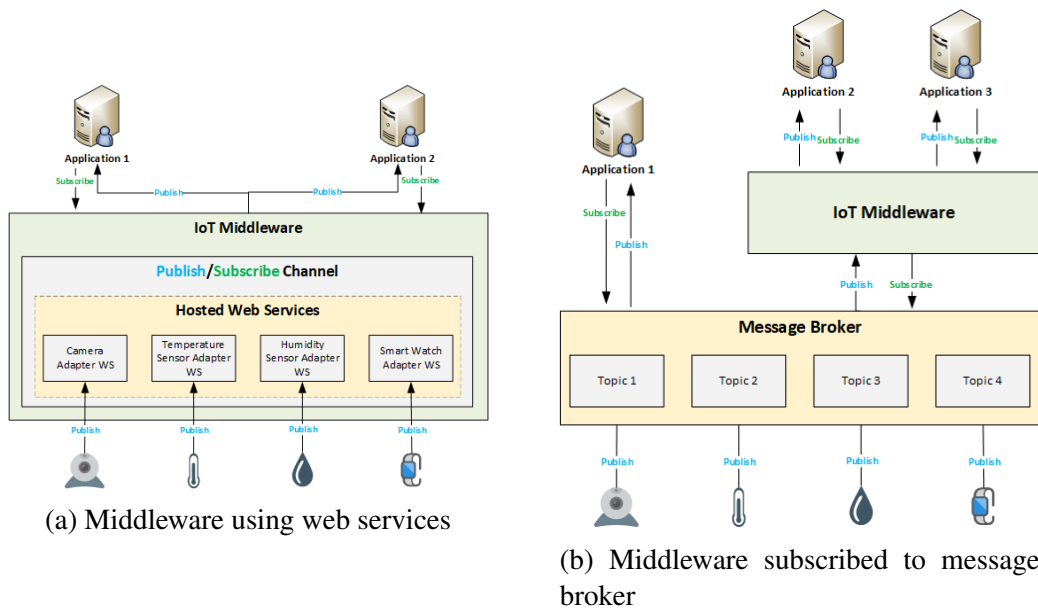


Figure 2.14: Middleware Publish/Subscribe

The IoT device can send the data directly to the middleware by calling a web service hosted in the middleware or by executing a remote procedure call. However, this approach requires the middleware to host a web service or remote method which receive the data and decides how to process it next. Indirect message delivery happens when the IoT device send the data indirectly to the middleware via a message broker. The message broker then delivers the data



to subscribes, which can be the middleware or other interested applications. MQTT protocol or its extensions might reflect the indirect message delivery to the middleware.

### 2.4.2 Related Work

FIWARE is a context-aware middleware which defines IoT agents to connect to IoT devices. Each agent supports a specified IoT protocol such as M2M, MQTT, and CoAP. Therefore, integrators should choose the IoT agent based on the protocol used by the IoT device [15]. The IoT agent would register the IoT device with the context broker and defines a service to receive observations/readings from the IoT device [16].

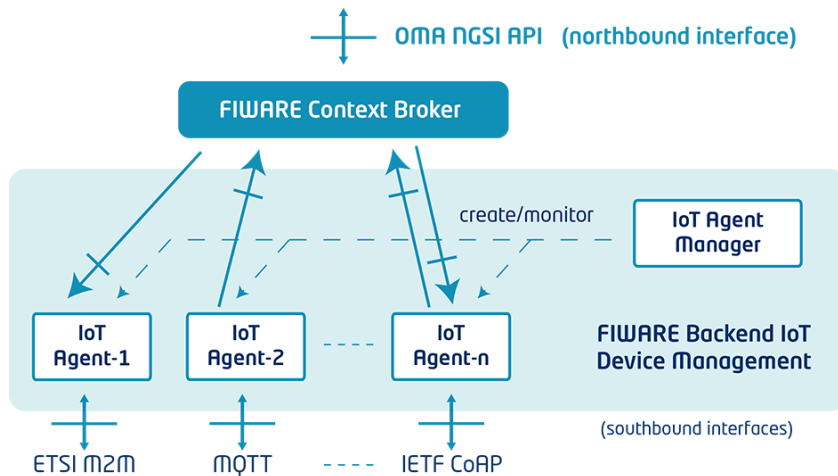


Figure 2.15: FIWARE IoT Device Management GE architecture [15]

Senaas is a middleware that uses sensor-as-a-service technique that declares different services offered by an IoT device on the cloud, and uses adapters to connect to IoT devices. This would allow IoT devices to produce events to the framework, and then the framework would make these events available for different clients since it uses an event-driven service oriented architecture (e-SOA) [3].

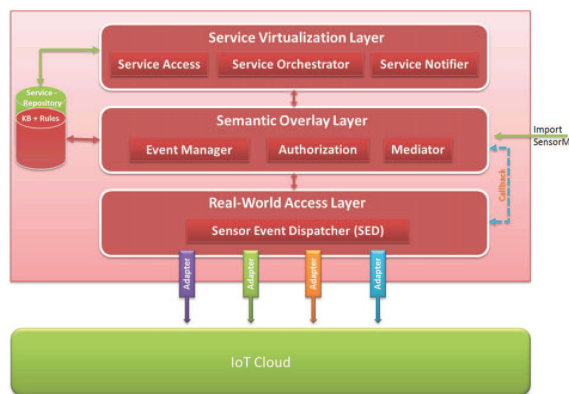


Figure 2.16: Senaas functional architecture [3]

Hunkeler & Truong et al [22] discuss the uses of publish-subscribe architecture where IoT devices publish its data to a broker through WSN gateway which provides access to the broker since the broker exists in a traditional network, and applications on the other hand subscribe to a broker to receive published sensor data they are interested in. In this context sensors publish data directly to the broker and applications receive data directly from the broker as illustrated in figure 2.17.

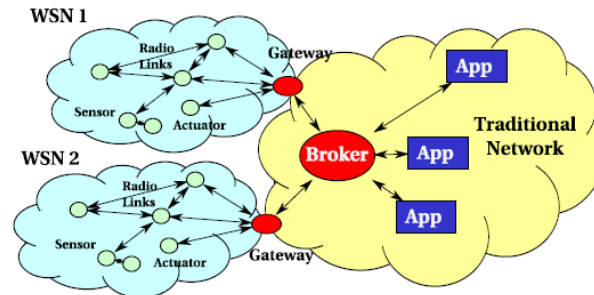


Figure 2.17: Wireless Sensor Networks with Publish-Subscribe Communication [3]

## 2.5 Gap Analysis

All the presented IoT architectures in this chapter served as intermediaries between IoT devices and applications providing proving abstraction between IoT devices and applications. These architectures can be evaluated using the following factors:

- **Adaption:** All the architectures provided a form of adaption between the underlying protocol and the applications. Therefore, it eliminated the need for applications to understand different protocol than what they need.
- **Adaptability:** Some of the architectures served well within a specific protocol and where not be able to communicate with different protocols such as CoAP Proxy and DPWS Gateway. Other architectures was able to communicate with different underlying protocols using adapters as a foundation for communication with IoT devices.
- **Processing location of sensor data:** Most of the architectures suggested processing of sensor data in one location without specifying if the data is processed on the cloud or near IoT devices. Some of the architectures such as Senaas [3] was clear the data is be processed on the cloud.
- **Extend physical sensor functionality:** All the architecture discussed a mechanism of acquiring sensor data and make it accessible of the applications without further processing of sensor data. except of GSN [1] which suggested the use of virtual sensors to process sensor data.
- **Distributing processing of sensor data:** All of the architectures except for GSN [1] assume processing of sensor data at single node.

What the discussed architectures are missing is the distributed processing of sensor data by adding carrying some of the processing required by application into distributed nodes, while providing the flexibility for applications to decide from where to consume sensor data. The use of adapters is essential to communicate with IoT devices regardless of the underlying protocol. However, applications needs a dynamic approach to receive sensor data rather than receiving data from a particular sensor. The use of the publish-subscribe design pattern would provide the applications the choice to subscribe to topics rather than sensors. Furthermore, the publish-subscribe would make it easier to introduce new sensor data since there is no dependency between data providers (IoT devices) and data consumers (applications).

# Chapter 3

## Virtual Sensors Middleware Architecture

This chapter describes the architecture which consists of components placed on fog nodes and the cloud. This chapter is organized as follows. Section 3.1 describes the virtual sensor. Section 3.2 describe the middleware communication and the use of the publish-subscribe design pattern. Section 3.4 describes the middleware different components at the fog node and the cloud. Where section 3.5 shows how components interact with each other using different scenarios.

### 3.1 Virtual Sensor

A Virtual Sensor (VS) is a core component of the middleware, which is responsible for processing sensor data by applying a combination of one or more of the following: transformation, filtering and aggregation. Virtual sensors can be either deployed at a fog node or on the cloud. The deployment of the virtual sensors at different locations does not impact how virtual sensors are conceptually structured. Each virtual sensor receives data from one or more input sources and produces data to only one output destination. The following briefly describes a virtual sensor's components used for handling data received by the virtual sensor:

1. **Consumer:** The consumer receives input and aggregates data received from the multiple inputs into a single tuple, which is forwarded to the Processor for further processing. The consumer applies a fault handling policy, which determines how to proceed if some of the data is missing or not received.
2. **Processor:** The Processor is responsible for processing tuples received from the consumer. The result is forwarded to the Publisher.
3. **Publisher:** The Publisher is responsible for disseminating data received from the Processor so that it can be used by other virtual sensors and applications.

The Virtual Sensor is responsible for processing data received from one or more input sources. The virtual sensor input sources, can be of the same type or mixed types as shown in figure 3.2. A virtual sensor can connect to a physical sensor using an adapter and/or receive its input from other deployed virtual sensors.

Each virtual sensor publishes its data to the publish-subscribe manager that is on the same node where the virtual sensor is deployed.

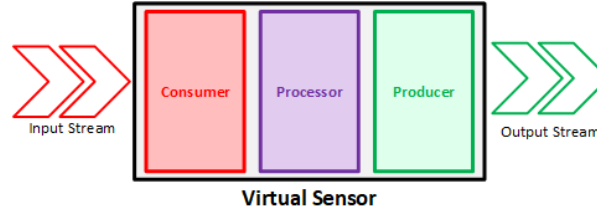


Figure 3.1: Virtual Sensor Abstraction

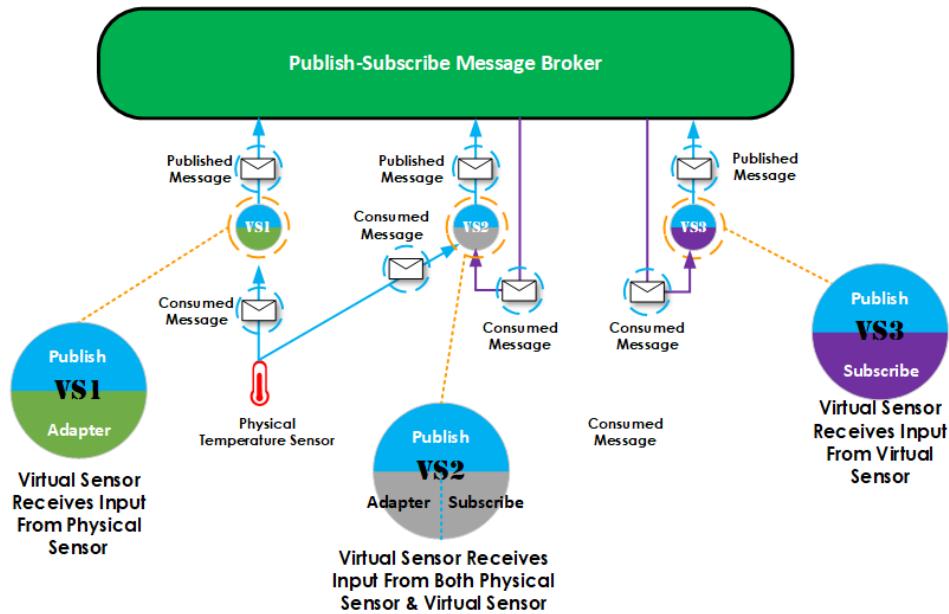


Figure 3.2: Virtual Sensor Input Sources

## 3.2 Middleware Communication

Section 3.1 describes the main components of the virtual sensor and the possible input sources of a virtual sensor. This section also describes the middleware topology and the adopted approach for disseminating data.

### 3.2.1 Disseminating Data

IoT devices use different protocols and the number of IoT devices vary over time. Therefore, there is a need for the use of a communication pattern that allows different devices to communicate with each other regardless of the underlying protocol. There is also a need to be able to make use of sensor data without having a prior knowledge of producers of the sensor data. The Publish-Subscribe design pattern is used since it provides a message proxy that allows data producers (publishers) and data consumers (subscribers) to exchange information without the need to maintain information about each other and the IoT devices can use different underlying protocols.

The publish-subscribe middleware is a message-oriented middleware [21], where it uses a

message broker to distribute messages to subscribers. There are three communication advantages:

- **Distributed communication:** Publishers and subscribers do not need to exist in the same domain. They can be anywhere in the network as long as they can communicate with the message broker
- **Loosely-coupled communication:** Publishers do not need to maintain a list of subscribers or any information about them. It is the responsibility of the message broker to deliver messages to all subscribers. Publishers need to deliver their messages to the message broker.
- **Asynchronous communication:** Publishers and subscribers do not need to be active at the same time. Publishers can produce messages to the message brokers, and subscribers can pick messages when they are active.

### 3.2.2 Publish-Subscribe communication patterns

The following is a list of models used by message brokers:

1. **Topic-Based:** Publishers publish data to topics declared on the message broker, where different publishers can publish to the same topic. Subscribers with interest in a topic register to receive updates on the selected topics [40].
2. **Content-Based:** This allows subscribers to enable restrictions on the received content by imposing filters that consist of pairs of attributes and their values. This model is more expressive since subscribers express their interest in contents (messages) by specifying conditions over the content they are interested in [40].
3. **Type-Based:** Messages are treated as objects that have different types. Consumers subscribe to receive messages of a specific type or sub-instance of a type regardless of the publisher or the content of the message [14].

Subscribers do not subscribe directly to publishers. Instead they subscribe to the topics of interest. Therefore, subscribers do not hold any information about the publishers since they are interested in topics rather than who publishes these topics.

### 3.2.3 Publish-Subscriber Topologies

The message broker can have different topologies depending on the criticality of the system, and number of the publishers and subscribers.

1. **Centralized:** With a centralized topology, there is a single message broker. This architecture has a single point of failure (SPOF), which means that if the message broker failed/stopped neither publishers nor subscribers would be able to produce or consume messages.

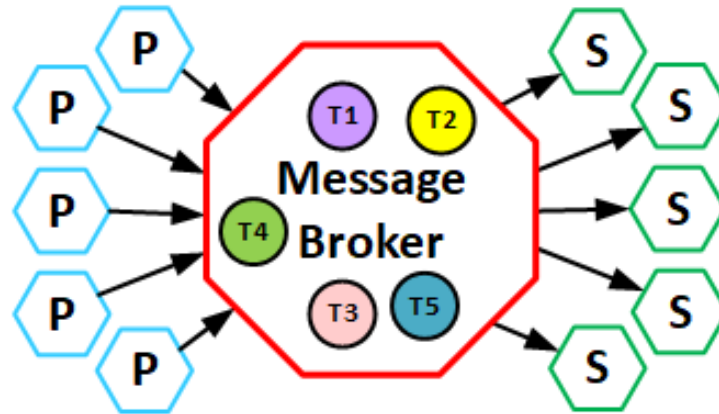


Figure 3.3: Centralized

2. **Clustered:** With the clustered approach, there is a cluster of message brokers that act as a single message broker. Each message broker serves a subset of publishers and subscribers that define their own topics. If one of the message brokers fail it would not hinder the operation, as the publishers, subscribers, and the topics in the failing node will be moved to another message broker; this process is called fail-over. Another advantage of clustering, is that load balancing can be used to distribute the load between the different message brokers. For example, if there are two message brokers *A* and *B*, the publishers and subscribers join the cluster using the cluster virtual name, rather than joining a specific message broker. Once they join the cluster, then the cluster decides on which message broker they will be added to. If the message broker *A* is down then all publishers, subscribers, and topics of message broker *A* will be moved to message broker *B*. In other words message broker *A* will perform a fail-over to message broker *B*.

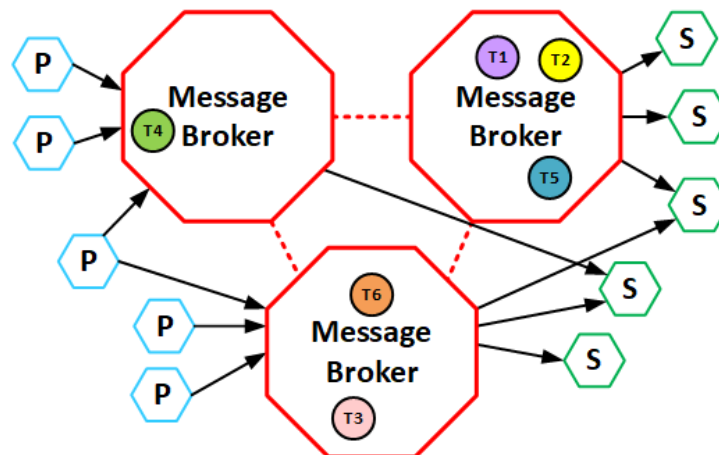


Figure 3.4: Clustered

3. **Federated:** Work can be distributed among message brokers, where each message broker can serve different geographical regions, a specific set of publishers/subscribers, or

set of topics. If one of the message brokers fails then only the publishers/subscribers using the failed message broker would be impacted. For example, if there are two message brokers *A* and *B* each message broker would be set to serve a specific city. The publishers and subscribers should specify which message broker they want to use. Publishers and subscribers on both message brokers *A* and *B* exchange messages. However, if message broker *A* is down then all publishers, subscribers, and topics of message broker *A* will be affected until message broker *A* resumes service. Publishers and subscribers of message broker *B* remains unaffected.

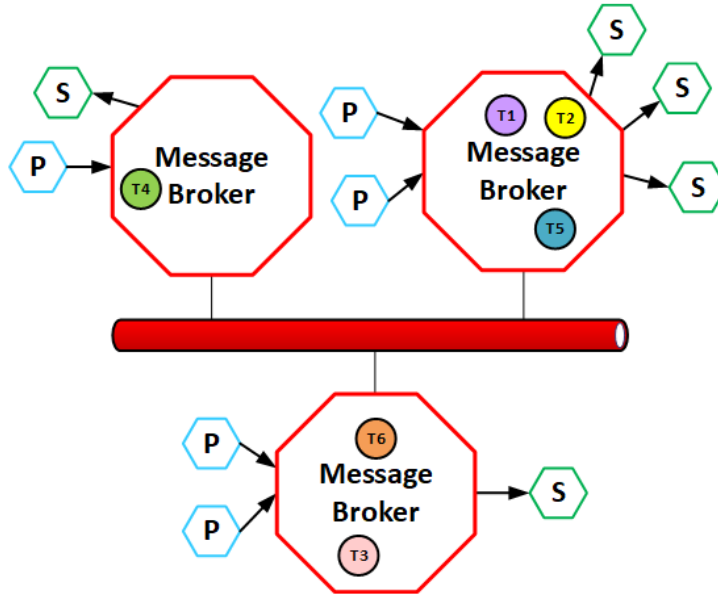


Figure 3.5: Federated

- 4. **Peer-to-Peer:** This type of architecture is used in Data Distribution Services (DDS) where publishers/subscribers write/read from/to Global Data Space (GDS) which is a logical (virtual) channel for exchanging information. DDS can define different GDS and it does not have single point of failure.

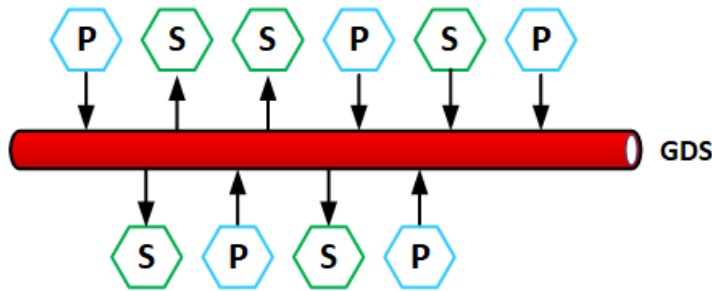


Figure 3.6: Peer-to-Peer



### 3.2.4 Why Federated Architecture?

For this work we considered several factors in selecting the topology to be used:

- The distributed nature of physical sensors
- The use of fog computing, where multiple fog-nodes can be deployed near wireless sensors networks, each of which may have multiple virtual sensors running on it.
- Failure of any part of the middleware should not hinder the overall operation of the middleware

The federated topology would serve best in the mentioned scenarios since it would allow virtual sensors to communicate across multiple fog nodes and if one fog node went down it would not affect the operation of other fog nodes but it might effect some subscribers who is waiting for data from publishers hosted on the affected node. However in a more complex scenario where high-availability is the desired option then a federation of clusters can be applied as shown in figure 3.7

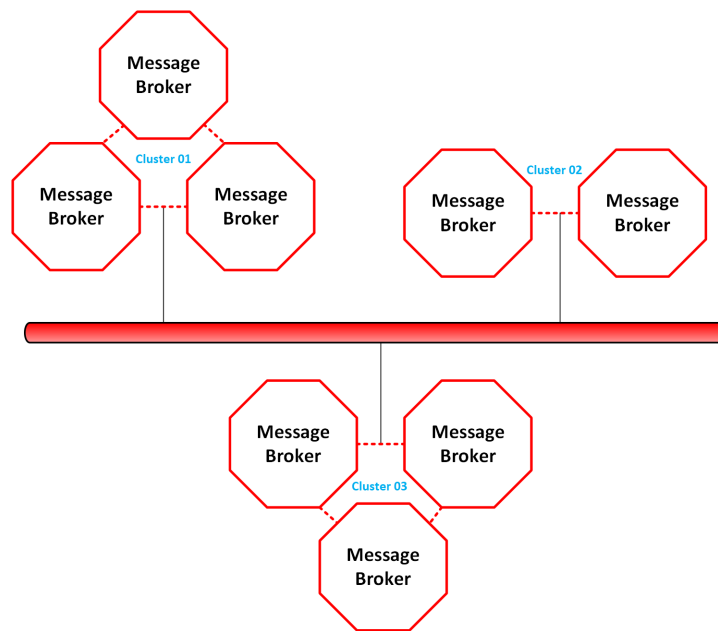


Figure 3.7: Federated Clusters

### 3.2.5 Virtual Sensor Communication

All virtual sensor communications use the Publish-Subscribe message broker as illustrated in figure 3.2. If a virtual sensor is to receive data from a physical sensor, then the virtual sensor uses an adapter that is responsible for connection management with a physical sensor in order to receive the data. Once the data is received, the virtual sensor publishes the data to the broker. A virtual sensor can subscribe to other virtual sensors as illustrated in figure 3.8.

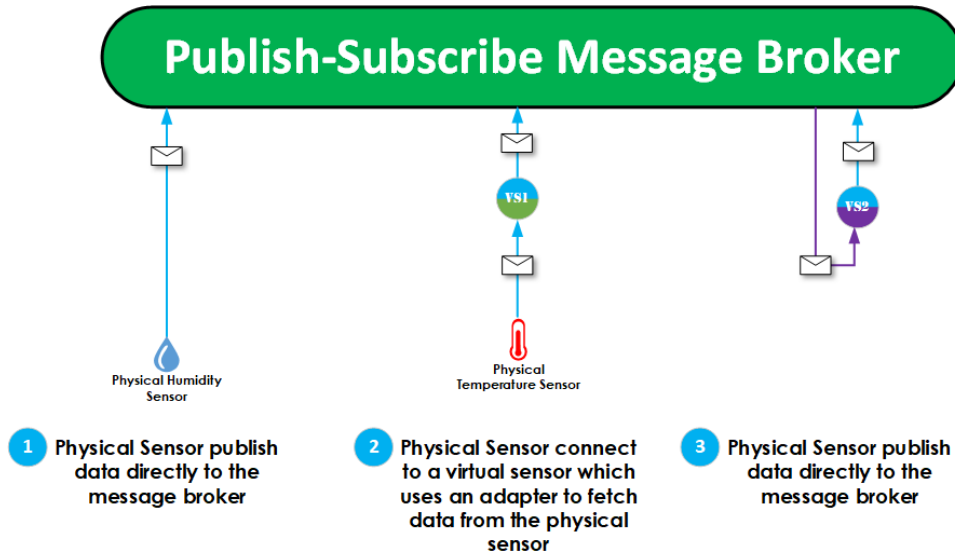


Figure 3.8: VSM Middleware Communications

### 3.3 Virtual Sensor Deployment Structure

Each virtual sensor is deployed within a container hosted on a fog node or cloud. All virtual sensor communications go through the Publish-Subscribe message broker as illustrated in figure 3.9.

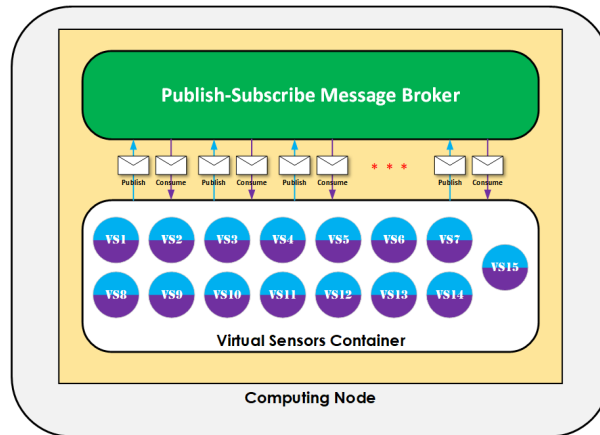


Figure 3.9: Virtual Sensors Physical Deployment

Conceptually virtual sensors can be structured into levels by specifying from where each virtual sensor should receive its input sources. Figure 3.10 shows an example deployment where virtual sensors are deployed at four different conceptual levels. If we look at VS9 of level 2 the virtual sensor receives its input from VS1 and VS2 of level 1. VS9 processes the data received from the two inputs, and then publishes the result to the next level (level 3), which is consumed by VS13.

Figure 3.11 shows a more complex conceptual deployment where virtual sensors might access data from different levels. A conceptual deployment might not contain levels and can

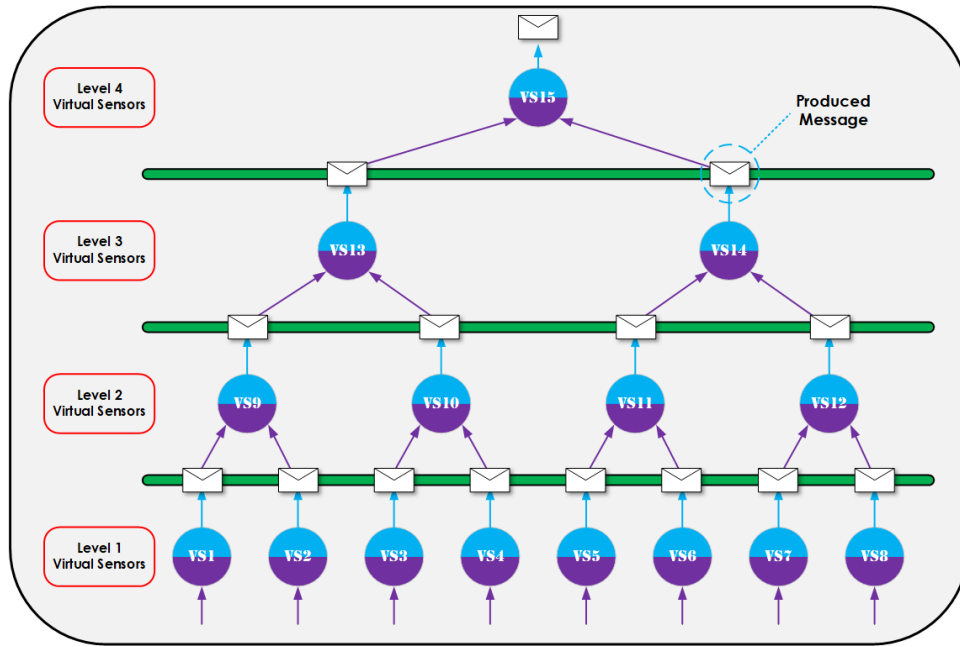


Figure 3.10: Virtual Sensors conceptual deployment

be totally random, which should be determined by the applications needs and the problem to solve.

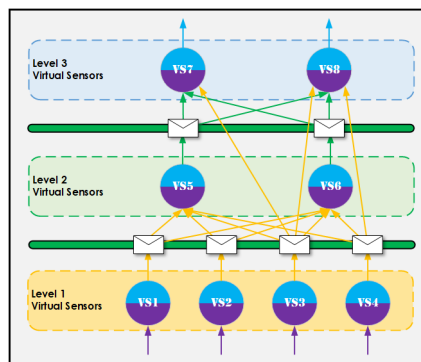


Figure 3.11: Virtual Sensor Conceptual Deployment: Acyclic Graph

### 3.4 Middleware Components

This section describes the middleware components deployed at the cloud and fog nodes and how they interact with each other. Figure 3.12 shows the different components of the middle-ware.

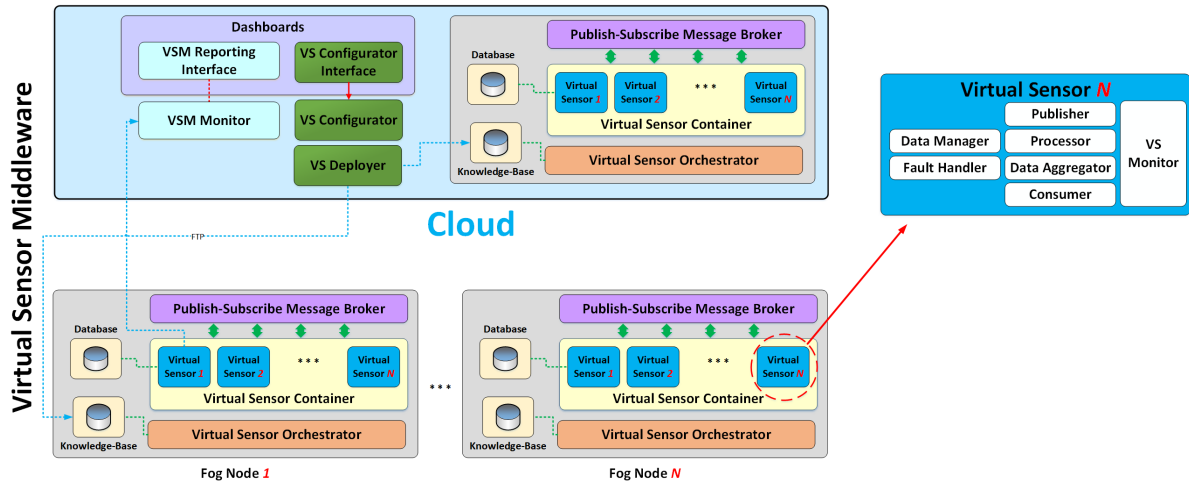


Figure 3.12: VS Middleware Architecture

### 3.4.1 Middleware Fog Components

1. **Virtual Sensor:** A virtual sensor is responsible for processing data received from physical sensors and make this data available to the applications. The components of the virtual sensor are described in more detail in section 3.4.3
2. **Publish-Subscribe Message Broker:** The publish-subscribe message broker at the fog node receives published data from the virtual sensors, where applications can access the published data by subscribing to the Publish-Subscribe Message Broker, regardless of where applications are hosted i.e. cloud or fog node.
3. **Database:** A virtual sensor can store data in a database, which enables auditing or analytics.
4. **Knowledge-Base:** This contains the different configurations (see section 3.4.3) of all deployed virtual sensors hosted on the fog node. These configurations are used to deploy and run the virtual sensor.
5. **VS Orchestrator:** The VS Orchestrator is responsible to act upon notifications received from the knowledge-base for newly added virtual sensor configurations or updates for existing configurations. The Orchestrator instantiates or updates the virtual sensor based on the provided configuration in the knowledge-base and maintains a reference of all instantiated virtual sensors inside the *Virtual Sensor Container* in case it requires to update or dispose the virtual sensor.
6. **VS Container:** Contains all running instances of the virtual sensors

### 3.4.2 Middleware Cloud Components

An administrator is responsible for specifying virtual sensor configuration (described in Section 3.4.3) and selecting the fog nodes for hosting the virtual sensors. This can be done through

either a script or a graphical user interface (GUI). In addition to script/GUI for specifying configurations and the publish-subscribe message broker, the following components are needed:

1. **Virtual Sensor Configurator:** This component receives configuration information from the administrator using a GUI referred to as the *VS Configurator Interface*. It creates and validates the syntax of the virtual sensors configurations and stores the configurations in a file. Section 3.5.1 describes the creation process of virtual sensor configurations.
2. **VSM Monitor:** This component is responsible for monitoring the status of all deployed virtual sensors. Each virtual sensor sends a message to the VSM Monitor at a configured frequency to report its status. If the VSM Monitor does not receive a signal from the virtual sensor it assumes that the virtual sensor is down.
3. **Virtual Sensor Deployer:** This component is responsible for deploying the virtual sensor configurations to the selected fog nodes. Once the deployer receives the configurations from the VS Configurator, it initiates the deployment process of virtual sensor configurations to the fog node. Section 3.5.2 describes the deployment process of virtual sensor configurations

### 3.4.3 Virtual Sensor Components

#### Virtual Sensor Configurations

Attributes of a configuration includes the following:

- **Virtual Sensor Output Destination:** Each virtual sensor should publish its data to unique topic defined at the message broker. The output source defines the unique identifier (ID) of the topic and the location of the message broker where it declares the topic and publishes the data.
- **Virtual Sensor Input Sources:** This defines a list of all topics that the virtual sensor is interested in. For each topic the virtual sensor should define the unique name of the topic and the location of the message broker where the topic is declared in order to receive published data related to the topic.
- **Virtual Sensor Frequency:** The frequency of the virtual sensor defines the rate at which the virtual sensor would check input queues for new data published by the input sources (topics).
- **Fault Handling Policy:** This refers to the policy used when there is a failure in receiving the data from input sources. The fault handling policy determines how the virtual sensor should act or proceed in case of such failures.
- **Data Management:** This contains information for connecting to the database if the virtual sensor needs to store data.

Virtual sensor configurations are sent to the *Knowledge-Base* component by the *Virtual Sensor Orchestrator* component responsible for creating the running instance of the virtual

sensor based on the provided configurations. The *Virtual Sensor Orchestrator* waits for an event from the knowledge-base to notify it with newly added configurations or updates on current configurations. Figure 3.13 shows how the Orchestrator instantiates VS1 based on the deployed configurations of VS1.

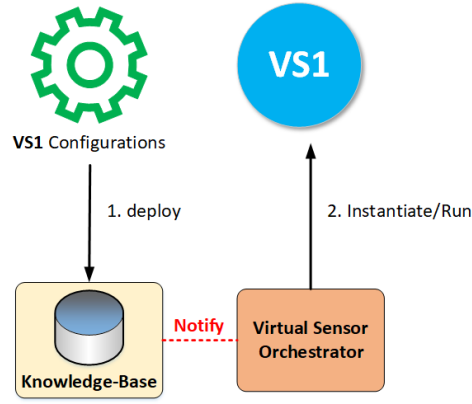


Figure 3.13: Virtual Sensor configurations vs. running virtual sensor

## Components

The virtual sensor consist of a set of components that are responsible for consuming, processing and publishing sensor data.. The components as illustrated in figure 3.14 are the following:

- **Consumer:** The *Consumer* connects to the input source and consumes data received from the input source. If the input source is a virtual sensor then the consumer subscribes to the message broker. However, if the input source is a physical sensor then adapters are used to establish connections to the physical sensor. Once the connection is established then the consumer will receive data from input sources.
- **Data Aggregator:** The data aggregator component is responsible for aggregating data received from multiple inputs. Data might arrive at different rates. The data aggregator applies a technique that places data received from different input sources into priority queues to sort messages/data based on their timestamps. There is a priority queue per input source. The *Data Aggregator* dequeues available data from all the queues and store the data into a tuple. The aggregator runs (dequeue data) at a fixed rate specified in the virtual sensor configuration using the *vsFrequency* attribute. The tuple produced by the aggregator might contain a subset of the data from the input sources, and it is up to the fault handler to determine how to proceed in this case. Once the data aggregator is ready to proceed with the data processing it forwards the data to the *Processor* component.
- **Fault Handler:** The fault handler is triggered by the data aggregator component when there is missing data needed for a tuple. The fault handler uses the action specified in the fault handling policy to be applied when data is missing. For example, it can proceed with processing partial data or it can wait for the missing data to arrive.

- **Processor:** The *Processor* is responsible for processing the produced tuple. Once the *Processor* produces the result it forwards it to the Publisher.
- **Publisher:** The Publisher is responsible for establishing a connection to the Publish-Subscribe message broker in order to publish the data produced by the processor.
- **Data Manager:** The virtual sensor can be configured to store produced data by the processor. The data manager component is responsible for establishing a connection to the database based on the virtual sensor configuration attribute *saveToDB* which indicates if the data needs to be saved to a database, and *databaseConnectionInfo* which contains the database connection information. This can be useful if applications would like to access historical data produced by the virtual sensor.
- **VS Monitor:** The virtual sensor monitor is responsible for reporting virtual sensor status by sending a signal to *VSM Monitor* which is hosted on the cloud. The signal is sent at a configured frequency and the absence of the signal means that the virtual sensor is down.

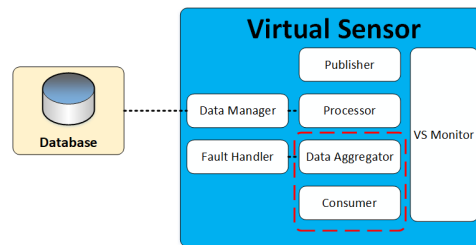


Figure 3.14: Virtual Sensor Components

## 3.5 Middleware Components Interaction Scenarios

### 3.5.1 Creating Virtual Sensor Configurations

An administrator is responsible for creating various virtual sensor configuration and for the deployment of the configuration to the fog nodes or the cloud. The virtual sensor configurations are used to initialize a virtual sensor. Figure 3.15 describes the steps of how the virtual sensor configuration is created.

1. The administrator specifies various virtual sensor configurations using the web-based interface component referred to as the *VS Configurator Interface*. The *VS Configurator Interface* sends the configuration to the *VS Configurator* component
2. The *VS Configurator* component generates the configuration based on the configuration received values from *VS Configurator Interface*
3. The VS configuration file will be loaded by the virtual sensor at the designated node. The virtual sensor object process data, subscribe to topics, and publish data based on the configurations stored inside the generated configuration file.

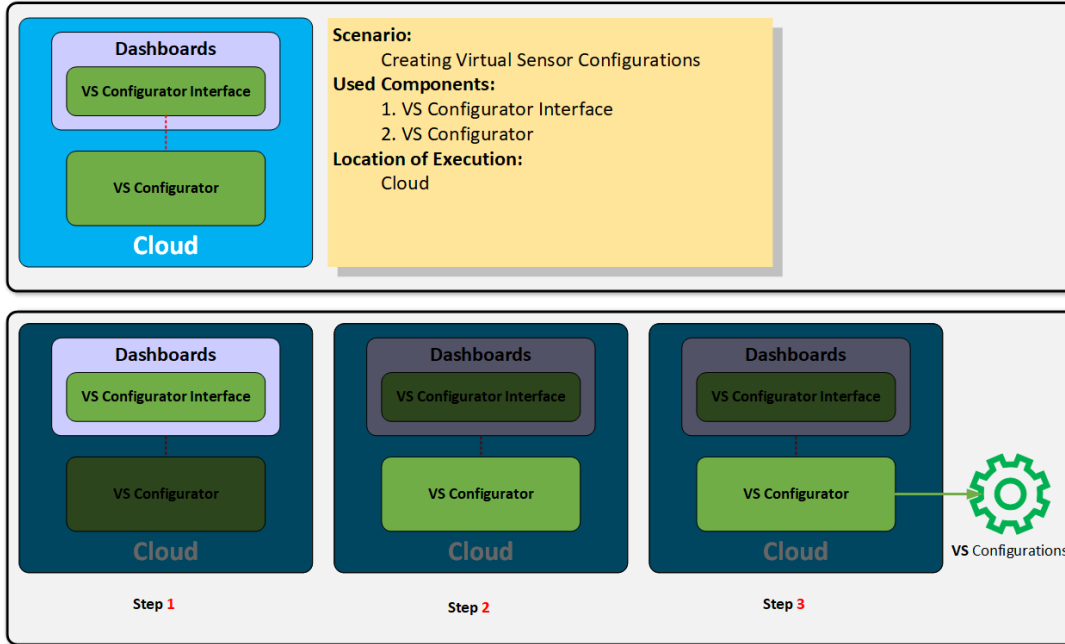


Figure 3.15: Creating Virtual Sensor Configurations

### 3.5.2 Instantiating Virtual Sensor Configurations

Figure 3.16 shows the steps needed to instantiate a virtual sensor.

1. The *VS Configurator* forwards the virtual sensor configuration to the *VS Deployer*, which is located on the cloud.
2. Upon receiving the configuration file, the *VS Deployer* uses the *vsNode* attribute of the configuration file to determine where the virtual sensor is to be deployed.
3. If the node is a fog node, then the *VS Deployer* transfers the configuration file to the *Knowledge-Base*.
4. *VS Orchestrator* will wait for events from the *Knowledge-Base* to inform it of updates (e.g., new configuration file added). Therefore, once a configuration file is added to the *Knowledge-Base*, The *VS Orchestrator* receives a reference to configuration file and instantiate the virtual sensor (explained in the next step).
5. The *VS Orchestrator* instantiates a virtual sensor by creating a new object of the virtual sensor and passing the configuration file. The virtual sensor processes the configuration file and sets the different attributes required by the virtual sensor using the attributes specified inside the configuration file. These attributes defines the input sources and output destinations, and other attributes as defined in section 3.4.3.



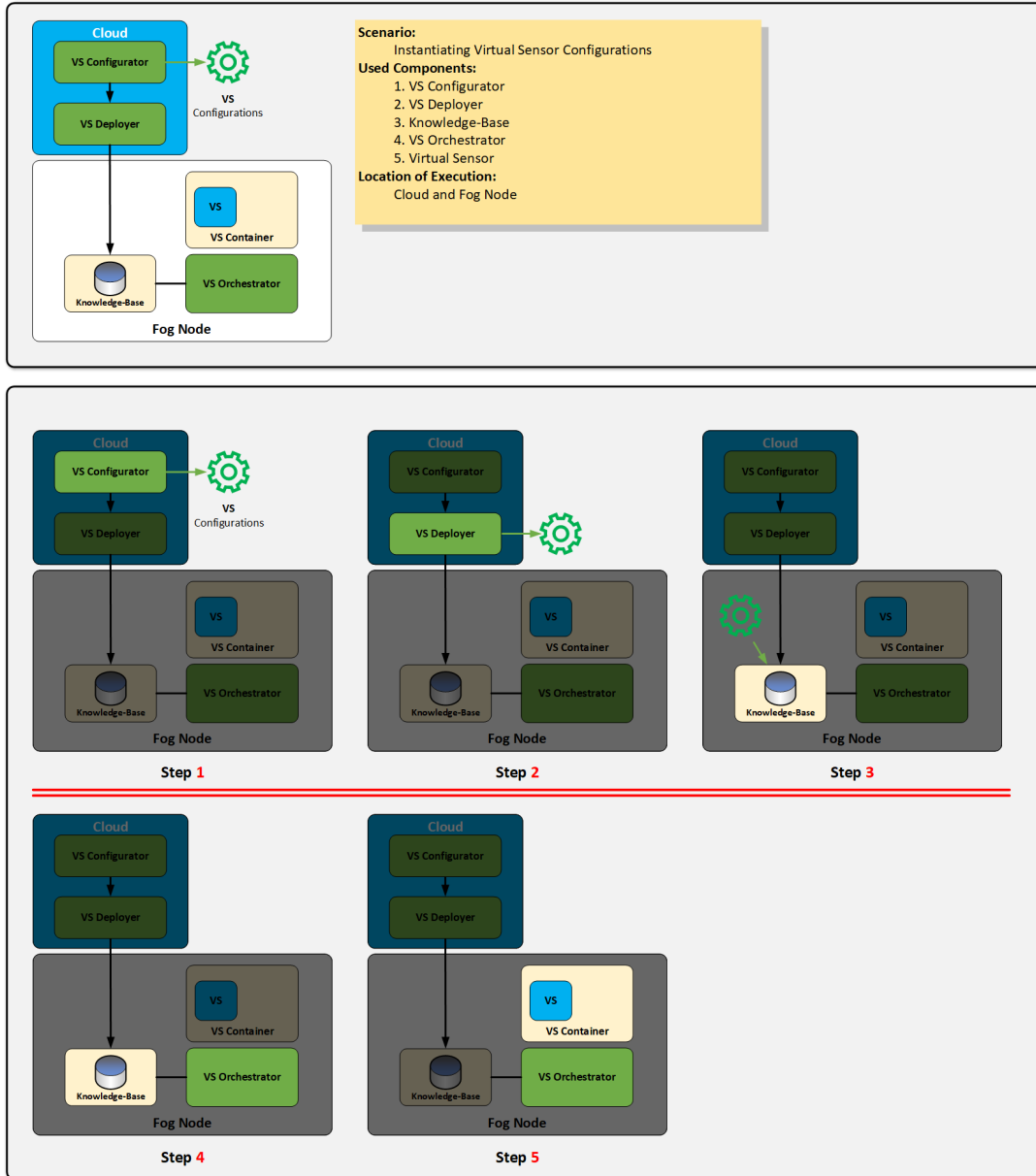


Figure 3.16: Instantiating Virtual Sensor

### 3.5.3 Exchanging Messages Between Virtual Sensors

This scenario is graphically depicted in Figure 3.17 shows two virtual sensors: The first virtual sensor publishes data while the second virtual sensor receives the published data after subscribing to the message broker. The second virtual sensor would know the topic where to receive its data based on the configuration file, which determines the input sources.

The scenario assumes that the data is ready for publishing at the first virtual sensor, and it assumes no fault handling or data management is required. The scenario ends when the second virtual sensor publishes the consumed data from the first virtual sensor. We will refer to the first virtual sensor as VS1, and the second virtual sensor as VS2. Both VS1 and VS2 exists on

the same fog node, and if VS1 and VS2 exists at two different nodes, the scenario would not change except for the fact that VS2 needs to subscribe to the topic of the message broker where VS1 publishes its data.

1. The *VS1 Publisher* timestamps the data and publishes it as a message to the topic declared by *VS1 Publisher* at *Publish-Subscribe Message Broker*. *VS1 Publisher* has the message broker details set from the instantiation process explained in Section 3.5.2. The *Virtual Sensor Configurations* contains the details and the location of the message broker and the name of the topic.
2. Upon receiving the message from *VS1 Publisher*, the *Publish-Subscribe Message Broker* searches for all interested subscribers.
3. The *Publish-Subscribe Message Broker* forwards the published message from *VS1 Publisher* to *VS2 Consumer* since it is registered as subscriber. The *VS2 consumer* timestamps the message and add it to a designated queue that stores data received from *VS1*. *VS2* has a designated queue per input source.
4. The *Data Aggregator* runs at a configured frequency that specifies the rate (per second) that it needs to check for data in the queues. Once it has the data, it creates a tuple with the data found from all queues and forwards it to the *Processor*.
5. When the *Processor* receives the tuple, it performs the configured function based on the attribute *vsAggregateFunction* stored inside the configuration file i.e. Average, Summation...etc. using the data stored in the tuple. It then forwards the result to the *VS2 Publisher*.
6. The *VS2 Publisher* timestamps the data and publishes it as a message to the *Publish-Subscribe Message Broker*.

## 3.6 UML Diagram

The middleware components described earlier uses different classes and tools to achieve the overall function of the middleware. Figure 3.18 describes some of the created components to serve specific functions of the middleware.

- **VirtualSensorOrchestator** represents the *VS Orchestrator*, which is responsible for monitoring the knowledge-base for new virtual sensor configurations.
- **VirtualSensor** is the main class that is used to instantiate all virtual sensors and initiating all dependencies required by the running virtual sensor.
- **VirtualSensorConfigObserver** is responsible for loading received configurations and pass it to different *VirtualSensor*, *VirtualSensorPublisher*, and *VirtualSensorConsumer*.
- **VirtualSensorConsumer** is responsible for maintaining references for the different input sources and initiating *VirtualSensorAggregator*.

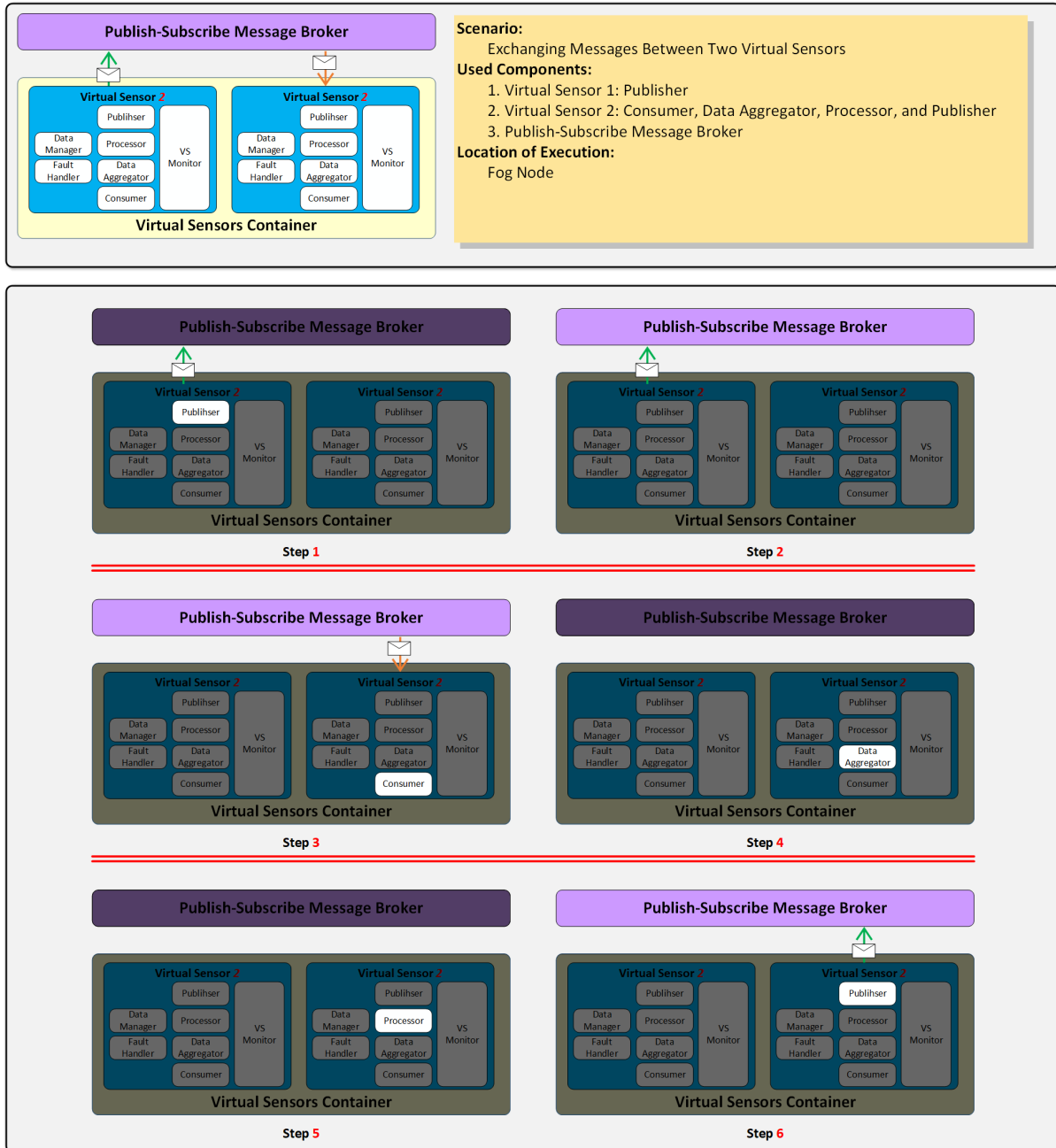


Figure 3.17: Exchange Messages Between Virtual Sensors

- **SubscribeExchange** is responsible for establishing connection with the message broker to receive published messages.
- **VirtualSensorAggregator** is responsible for aggregating data received from multiple inputs (multiple SubscribeExchange).
- **FaultHandler** is responsible to decide how to proceed if aggregated data is incomplete. It works closely with VirtualSensorAggregator.

- **VirtualSensorProcessor** is responsible for processing received tuples from the consumer.
- **VirtualSensorDataManager** is responsible for storing results produced by the processor if the virtual sensor is configured to store data in the database.
- **VirtualSensorPublisher** is responsible for initiating the PublishExchange to publish received data from the processor.
- **PublishExchange** is responsible for establishing connections to the message broker and publishing produced data.
- **VirtualSensorExchange** is a super class for both PublishExchange and SubscribeExchange that contains common features and functions used by both classes.
- **VirtualSensorMonitor** is responsible for reporting the virtual sensor status to the cloud by sending a signal at a configured frequency.

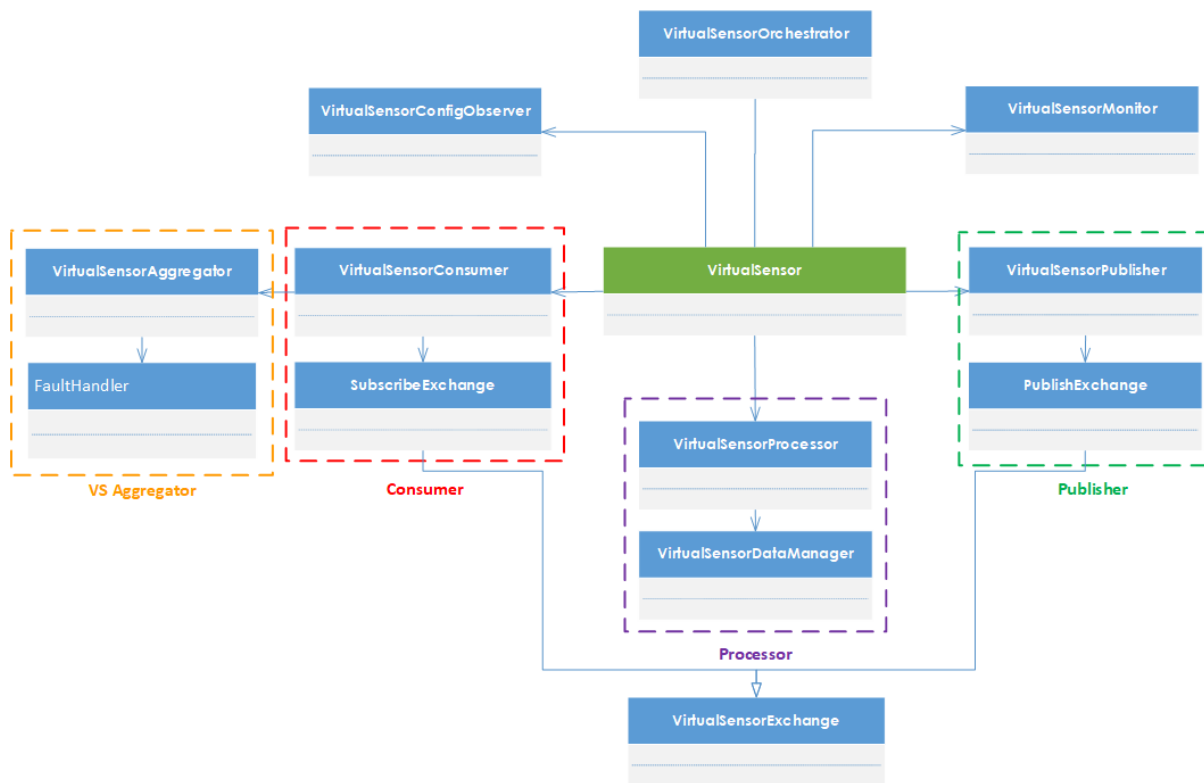


Figure 3.18: UML Diagram

# Chapter 4

## Virtual Sensor Middleware Implementation

This chapter describes the underlying technology used in the virtual sensor middleware (vsm) and the software and tools used to implement the interactions between the middleware different components.

### 4.1 Platform

Our platform uses *Raspberry Pi 3 Model B+*. The Raspberry Pi has the following specifications [18]:

- 1.4 GHz 64-bit quad-core ARM Cortex-A53 CPU
- 1 GB RAM (LPDDR2 SDRAM)
- Wireless LAN – dual-band 802.11 b/g/n/ac
- Gigabit Ethernet (300 Mbps)

The operating system installed on the Raspberry Pi is Raspbian Stretch version 4.14 [19], which is a Linux Debian based operating system optimized for Raspberry Pi.

### 4.2 Development Tools

The software was written using Java SE v1.8 and Maven was used to import the required libraries from the Maven repository.

### 4.3 Middleware Cloud Components

#### 4.3.1 Virtual Sensor Configurator Interface

The *Virtual Sensor Configurator Interface* is a web-based component hosted on the cloud that is responsible to generate configuration for a single virtual sensor. The following technology

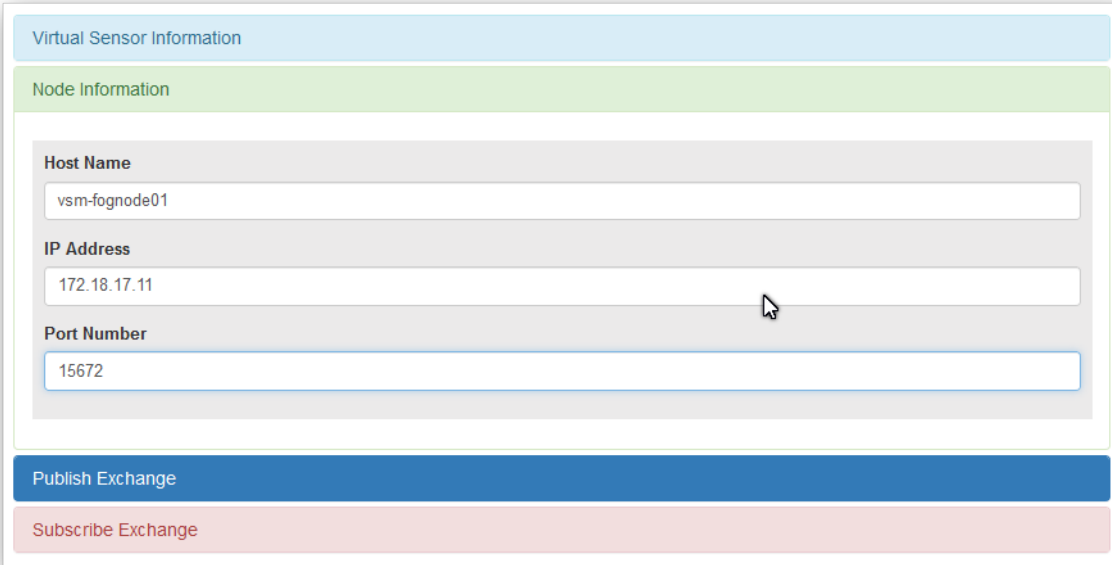
is used to create the interface:

- **Java Server Pages (JSP):** The JSP is a Java server-side programming language that is used to display the interface required to enter virtual sensor configuration values.
- **JQuery** is a rich JavaScript library [25] that is used to collect the entered data in the web-interface fields, generate JSON syntax on the fly, and then display it inside the JSON Editor web-component.
- **JSON Editor:** JSON Editor is a web-based tool used to view, edit, format, and validate JSON code. It provides different modes to view JSON code such as a tree editor, a code editor, and a plain text editor [13].
- **Bootstrap:** The Bootstrap is a framework that is used to provide responsive HTML components and enhance the look-and-feel of the interface [8]

Figure 4.1 through 4.4 shows the interface used to enter virtual sensor configurations.

The image shows a web-based configuration interface for a Virtual Sensor. The main section is titled "Virtual Sensor Information" and contains several input fields and dropdown menus. The fields are: "VS Name" (text input with value "VS-01-000001"), "VS ID" (text input with value "01-000001"), "VS Type" (dropdown menu with value "Processor"), "Frequency" (text input with value "1"), "Initial Delay" (text input with value "1"), and "Fault Handling Policy" (dropdown menu with value "Policy 1"). Below this section are three buttons: "Node Information" (green), "Publish Exchange" (blue), and "Subscribe Exchange" (red).

Figure 4.1: VS Configurator GUI - Virtual Sensor Information



The screenshot displays the 'Virtual Sensor Information' GUI for configuring a deployment node. It features a light blue header, a green 'Node Information' section, and three input fields for 'Host Name' (vsm-fognode01), 'IP Address' (172.18.17.11), and 'Port Number' (15672). At the bottom, there are two buttons: 'Publish Exchange' in blue and 'Subscribe Exchange' in red.

Figure 4.2: VS Configurator GUI - Deployment Node

### 4.3.2 Virtual Sensor Configurator

The *Virtual Sensor Configurator* component is a Java Servlet that receives the entered JSON configurations from the *Virtual Sensor Configurator Interface* and creates a temporary JSON file on the server and pass it *Virtual Sensor Deployer* to initiate the deployment process.

### 4.3.3 Virtual Sensor Deployer

The *Virtual Sensor Deployer* component is a Java Servlet that uses Apache Commons Net v3.6 [5] to transfer virtual sensor configurations (JSON file) to the desired node using File Transfer Protocol (FTP). The *Virtual Sensor Deployer* deploys the JSON file based on the *vsNode* attributes that contains the node information.

### 4.3.4 Virtual Sensor Configurations

The virtual sensor configuration is a JSON file that contains different attributes of the virtual sensor which is used to specify the different settings of the virtual sensor, such as the input sources and output destination. The configuration file can be created individually using *Virtual Sensor Configurator GUI* or it can be generated using a *VS Configuration Generator Script*, which is used to create configurations for a bulk of virtual sensors. The following are the attributes that are used inside the JSON file:

- **vsName:** vsName represent the virtual sensor name
- **vsID:** vsID is a unique identifier that is used to uniquely identify the virtual sensor across the middleware. It consist of two two parts. The first part refers to the node number

The screenshot shows the 'Publish Exchange' configuration page in the VS Configurator GUI. The page is divided into three main sections: 'Virtual Sensor Information' (light blue header), 'Node Information' (light green header), and 'Publish Exchange' (dark blue header). The 'Publish Exchange' section contains several form fields:

- Exchange Name:** A text input field containing 'PE-01-000001'.
- Exchange Type:** A dropdown menu with 'fanout' selected.
- Username:** A text input field containing 'admin'.
- Password:** A text input field containing 'admin'.
- Durable:** A dropdown menu with 'True' selected.
- Auto-Acknowledge:** A dropdown menu with 'True' selected.
- Node:** A section containing two text input fields: 'Host Name' with 'vsm-fognode01' and 'IP Address' with '172.18.17.11'.

At the bottom of the form, there is a red button labeled 'Subscribe Exchange'.

Figure 4.3: VS Configurator GUI - Publish Information

where the virtual sensor is deployed and second part refers to the virtual sensor sequence number on the selected node.

- **vsFrequency:** vsFrequency is used to specify the rate in seconds at which *VS Aggregator* would check the queues for a newly arrived data
- **vsInitialDelay:** vsInitialDelay is used to set the delay time in seconds before the *VS Aggregator* starts running at the specified frequency.
- **vsAggregateFunction:** vsAggregateFunction is used to specify the function to be performed by the *Processor* on the received data e.g., average, summation.
- **vsNode:** vsNode is a JSON object that contains the hostname and IP address of the node at which the JSON file is to be deployed.
- **publishExchange:** publishExchange is a JSON object that is used to describe the output destination of the virtual sensor. In other words, it is used to determine where the virtual sensor will publish its data. publishExchange objects contains the following values:



The screenshot shows the 'Subscribe Exchange' configuration page in the VS Configurator GUI. The page has a light blue header 'Virtual Sensor Information', a light green sub-header 'Node Information', a dark blue 'Publish Exchange' button, and a light red 'Subscribe Exchange' header. Below this, there are several form fields:

- Exchange Name:** PE-00-000001
- Exchange Type:** fanout
- Username:** admin
- Password:** admin
- Durable:** True
- Auto Acknowledge:** True
- Queue Name:** SE-Queue-01-000001-00-000001
- Node:**
  - Host Name:** vsm-fognode01
  - IP Address:** 172.18.17.11

At the bottom left of the form is a red 'Add Subscriber' button.

Figure 4.4: VS Configurator GUI - Subscribe Information

- **exchName:** exchName specifies the name of the exchange (topic) at which the virtual sensor publishes its data.
- **exchType:** exchType specifies how the exchange distributes data to subscribers. The current supported value is *fanout* which means the published data is delivered to all subscribers
- **exchNode:** exchNode is a JSON object that specifies the information of the node at which exchange is going to be declared.
- **exchUsername:** exchUsername specifies the username required by the virtual sensor to connect to the message broker (RabbitMQ) and publish data to the exchange (topic).
- **exchPassword:** exchPassword specifies the password required by the virtual sensor

```

1 {
2   "vsName": "VS-01-000001",
3   "vsID": "01-000001",
4   "vsType": "Processor",
5   "vsFrequency": "1",
6   "vsInitialDelay": "1",
7   "faultHandlerPolicyID": "policy1",
8   "vsNode": {
9     "hostname": "vsm-fognode01",
10    "hostip": "172.18.17.11",
11    "portNumber": "15672"
12  },
13  "publishExchange": {
14    "exchName": "PE-01-000001",
15    "exchType": "fanout",
16    "exchUsername": "admin",
17    "exchPassword": "admin",
18    "exchDurable": "true",
19    "exchAutoAck": "true",
20    "exchNode": {
21      "exchHostName": "vsm-fognode01",
22      "exchHostIP": "172.18.17.11"
23    }
24  },
25  "subscribeExchanges": [
26    {
27      "exchName": "PE-00-000001",
28      "exchType": "fanout",
29      "exchUsername": "admin",
30      "exchPassword": "admin",
31      "subExchDurable": "True",
32      "subExchAutoAck": "True",
33      "exchQueueName": "SE-Queue-01-000001--00-000001",
34      "exchNode": {
35        "exchHostName": "vsm-fognode01",
36        "exchHostIP": "172.18.17.11"
37      }
38    }
39  ]
40 }

```

Figure 4.5: VS Configurator GUI - Generated Configuration Ready For Deployment

to connect to the message broker (RabbitMQ)

- **subscribeExchanges:** This is a JSON array representing all input sources (topics). Each item in the array has the same attributes of the *publishExchange* in addition to *exchQueueName*. Each array item describes an input source.
- **exchQueueName:** *exchQueueName* is used to define the queue name at which the virtual sensor stores the data received from the input source (exchanges/topics that virtual

sensor is subscribed to)

- **faultHandlerPolicyID**: `faultHandlerPolicyID` is used to set the fault handling policy of the virtual sensor, which is used by *FaultHandler* component.
- **saveToDB**: `saveToDB` is a boolean attribute used to specify if the sensor data needs to be stored in a database.
- **database**: `database` is a JSON object that is used to specify the database information required to connect to the database if *saveToDB* is set to true.
  - **dbName**: `dbName` is the database name that is used to store sensor data
  - **driverURL**: `driverURL` is a database URL to server that hosts the database. The URL contains the host name, port number, and the database name.
  - **dbUsername**: `dbUsername` is used to set the username required to connect to the database.
  - **dbPassword**: `dbPassword` is used to set the password required to connect to the database.

A sample JSON file that contains the virtual sensor configurations can be found in **appendix A**

## 4.4 Middleware Fog Components

### 4.4.1 Publish-Subscribe Message Broker

The Publish-Subscribe Message Broker is implemented using RabbitMQ, which is an open source message broker that uses AMQP 0-9-1 as underlying protocol to exchange messages between publishers and subscribers asynchronously [34]. Additionally, RabbitMQ supports transmitting messages using HTTP and other messaging protocols such as Simple Text Oriented Messaging protocol (STOMP), Message Queuing Telemetry Transport protocol (MQTT), and AMQP 1.0 [36]. RabbitMQ is used as a message broker to exchange messages between different virtual sensors, and IoT devices.

RabbitMQ can be deployed on single mode as a centralized message broker or deployed across multiple nodes to support cluster or federation architectures. In the current implementation RabbitMQ is deployed on different nodes using federation topology, where the virtual sensors deployed at different nodes would be able to exchange messages using the RabbitMQ-federation.

Figure 4.6 shows how the publisher declares an exchange which represents the topic of the publisher, where the consumer declares queue to receive published messages.

### 4.4.2 Knowledge-Base

The *Knowledge-Base* is a designated folder in the filesystem that stores all virtual sensor configurations (JSON files). Once a new file is added to the Knowledge-Base it produces an event

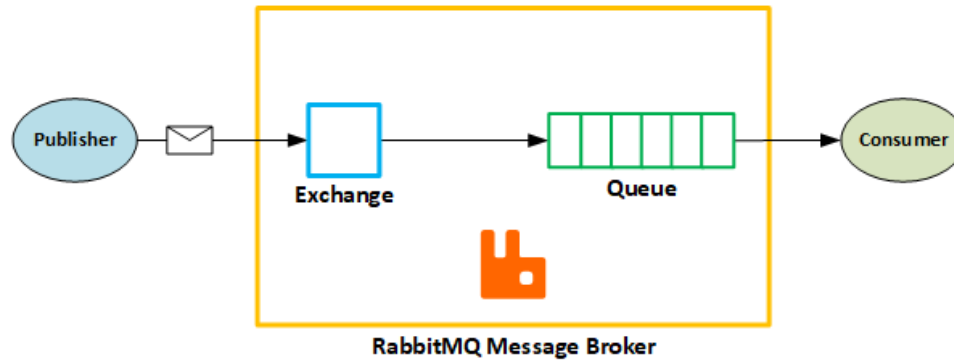


Figure 4.6: RabbitMQ Simple Publisher/Consumer Architecture

that is picked up by the *Virtual Sensor Orchestrator* to create a new virtual sensor based on the configuration file.

### 4.4.3 Database

The *Database* is used to store virtual sensor data if the virtual sensor is configured to store data. The middleware supports the use of any database as long as it is supported by the operating system on the fog node and it has a Java driver to establish connection to the database such as MySQL.

### 4.4.4 Virtual Sensor Orchestrator

The *Virtual Sensor Orchestrator* implemented using Java, where it awaits a notification from the Knowledge-Base which contains the virtual sensor configuration file location. Then the Orchestrator would instantiate the virtual sensor using *VirtualSensor* class and add it to the *Virtual Sensor Container*.

### 4.4.5 Virtual Sensor Container

The *Virtual Sensor Container* is a Java *ArrayList* object that contains all references of the instantiated Virtual Sensor objects. These references used to update virtual sensor based on updates on virtual sensor configurations.

### 4.4.6 Virtual Sensor Libraries

The following software libraries are used by a virtual sensor:

#### RabbitMQ Java Client Library

*RabbitMQ Java Client Library* provides the APIs required for the virtual sensor to connect to the RabbitMQ server (Message Broker) and publish/consume messages to/from the server [35]. As described earlier the virtual sensor consist of Publisher and Consumer components. These

components are required to establish a connection with the RabbitMQ server using *RabbitMQ Java Client Library* and declare the necessary *Exchanges* and *Queues* to exchange messages with RabbitMQ message broker.

### Java UUID Generator (JUG)

Java UUID Generator is a Java APIs developed by FasterXML that is used to generate a *Universal Unique Identifier (UUID)*, This is used to identify published messages. UUIDs does not require a centralized authorities to administer them and they are fixed to 128 bits that are unique across space and time [33]. UUID consist of 32 case-sensitive hexadecimal-digits as shown below [33]:

Table 4.1: UUID Structure

<b>A</b>	-	<b>B</b>	-	<b>C</b>	-	<b>DE</b>	-	<b>F</b>
----------	---	----------	---	----------	---	-----------	---	----------

Table 4.2: UUID Breakdown

Code	Attribute	Value
A	time-low	8 hex-digit
B	time-mid	4 hex-digit
C	time-high-and-version	4 hex-digit
D	clock-seq-and-reserved	2 hex-digit
E	clock-seq-low	2 hex-digit
F	node	12 hex-digit

Since RabbitMQ does not assign message identifiers with the messages it is the responsibility of the application to assign unique message identifiers. Therefore, the virtual sensor *Publisher* component generates UUID per published message to guarantee the uniqueness of the message across the fog nodes.

### Apache Log4J

Logging is very important in order to debug the application and monitor the communication across multiple components. Therefore, the middleware uses Log4J [17], which is a Java logging framework developed by Apache to support log generation to the filesystem instead of logging to standard output. The generated logs can be configured to contain the required information and the level of debugging.

### JSR 374 (JSON Processing)

JSR 374 is a Java library used to generate, parse, transform, and query JSON files [32]. The library is used to process virtual sensor configuration files that contains the different attributes used to create the virtual sensor.

## 4.5 Implementation Classes

The following are the main classes implemented in Java running the virtual sensor. Each virtual sensor would have its own instances of these classes.

### 4.5.1 VS Aggregator

Algorithm 1 is responsible for aggregating data from multiple input sources if the virtual sensor subscribes to more than one input source. Algorithm 1 checks for received data in the queues. Algorithm 1 runs in the background in a thread at configured frequency that is set at the time of starting the thread.

---

#### Algorithm 1: Virtual Sensor Aggregator

---

```

Input : Virtual Sensor Queues queues
1 t ← new Tuple();
2 if queues ≠ empty then
3   for each qi ∈ queues do
4     if Peek(qi) ≠ empty then
5       message ← dequeue(qi) ;
6       t ← add(message) ;
7   if isComplete(t) == true then
8     forwardToProcessor(t) ;
9   else
10    forwardToFaultHandler(t) ;

```

---

The algorithm starts by creating a new tuple at line 1, it then checks if the queues received data from the input sources (topics/exchanges) at line 2. Lines 4-6 is applied to each queue. The Peek function on line 4 checks the head of the queue to determine there is data. If the queue has data then data is dequeued(line 5) and added to the tuple (line 6). if the queue has data then it will dequeue the head of the queue at line 5 and then add it to the tuple at line 6.

After the algorithm checks all the queues it then check if the tuple size is equal to the number of the queues at line 7 by calling a function *isComplete()*. If the tuple is complete the algorithm forwards the tuple to *Processor* component at line 8. if the tuple is not complete then it forwards the tuple to the *fault-handler* component at line 10 to decide on the action to be done based on the policy number.

### 4.5.2 Publisher

The publisher component is responsible to establishing connections to the message broker and publish the produced results of the *Processor* component. The publisher is called by the processor upon the completion of the computation.

Algorithm 2 starts by establishing a connection to the message broker (RabbitMQ) at 1 and creating a channel at line 2 as required by RabbitMQ to create a topic, and then the algorithm at line 3 declares an exchange (topic) based on the information associated with the virtual sensor

---

**Algorithm 2:** Publish Virtual Sensor Data

---

**Input** : Data  $d_i$ , Virtual Sensor  $VS_i$

- 1  $connection \leftarrow connectToMessageBroker()$ ;
- 2  $channel \leftarrow connection.getChannel()$ ;
- 3  $exchnage \leftarrow getExchangeProperties(VS_i)$  ;
- 4  $channel.declare(exchange)$  ;
- 5 create new Message  $M_i$  ;
- 6  $M_i.timestamp \leftarrow getCurrentTimestamp()$  ;
- 7  $M_i.messageID \leftarrow generateUUID()$  ;
- 8  $channel.publish(M_i)$  ;

---

which was loaded from the JSON file during the virtual sensor instantiation process. Once the topic is declared, it adds exchange to the message broker at line 4. Then at line 5 it creates a new message with the data that is going to be publish, and timestamps the message at line 6 and add UUID to the message at line 7. After setting up the message it get published at line 8.

### 4.5.3 Consumer

The main function of the consumer is to establish connection for each input source (topic) that it should fetch data from. The consumer define an array of objects that maintain references for all input source. These objects are declared as *SubscribeExchange* that is responsible to establish connection with the message broker, declare a queue to the corresponding exchange, then receive published messaged delivered to the queues.

---

**Algorithm 3:** Consume Virtual Sensor Data

---

**Input** : Data  $d_i$ , Virtual Sensor  $VS_i$

- 1  $connection \leftarrow connectToMessageBroker()$ ;
- 2  $channel \leftarrow connection.getChannel()$ ;
- 3  $exchnage \leftarrow getExchangeProperties(VS_i)$  ;
- 4  $queue \leftarrow getQueueProperties(VS_i)$  ;
- 5  $channel.declare(exchange)$  ;
- 6  $channel.declare(queue)$  ;

---

Algorithm 3 starts by establishing connection to the message broker (RabbitMQ) at 1 and creating a channel at line 2 as required by RabbitMQ to declare the topic that the consumer wants to receive updates from. At line 3 the consumer declares an exchange (topic) based on the information associated with the virtual sensor which was loaded from the configuration file (JSON file) during the virtual sensor instantiation process. Once the exchange is declared, a consumer needs to define the queue that is going to be associated with the exchange to receive published messages at line 4. The consumer then adds the exchange and queue to the channel at lines 5 and 6 respectively. After the consumer declares the topic that it is going to listen to and associate a queue with the topic, it will start listening to the channel, waiting for updates from the publisher.

#### 4.5.4 VS Orchestrator

VS Orchestrator is the main class that is responsible for loading and running virtual sensors based on the configuration file stored in the knowledge-base.

---

**Algorithm 4:** Watch Knowledge-Base For New Configurations

---

**Input** : Knowledge-Base  $K$

```
1  $vsArray \leftarrow new Array()$  ;  
2 while  $watch(K) == true$  do  
3   | if  $K$  received new configuration  $C_i$  then  
4   |   |  $VS_i \leftarrow new VirtualSensor(C_i)$  ;  
5   |   |  $vsArray.add(VS_i)$  ;
```

---

Algorithm 4 starts at line 1 by creating a new array to store all references for virtual sensors in case a virtual sensor needs to be updated or disposed. At line 2 the algorithm waits for an event to be created by the knowledge-base (new file is added). If a new configuration file (JSON) is added to the knowledge-base then an event is created which satisfies the condition and contains a reference to the new configuration file at line 3. A virtual sensor is then instantiated at line 4 and the reference for the virtual sensor is stored in  $vsArray$  at line 5.



# Chapter 5

## Evaluation

This chapter describes the performance evaluation of Virtual Sensor Middleware discussed in the previous chapters. The main objective of the evaluation is to observe the different factors that are related to the middleware and define how they have impact over the performance. Section 5.1 describes the evaluation environment and the different tools and software used to perform the evaluation. Section 5.2 describes the different factors that are evaluated. Section 5.3 describe the baseline performance of the devices used for evaluation. Section 5.4 describes the different evaluation scenarios used to show the impact of the different factors over the performance. Section 5.5 summarize the evaluation results.

### 5.1 Evaluation Environment

As described in chapter 3 the middleware consist of different components that run on the cloud or on the fog node. The cloud components provides the interface required to generate and deploy virtual sensors configuration. Where the fog node would contain the different components responsible to process sensor data. Therefore, the following are used to simulate the performance as shown in figure 5.1.

- **Raspberry Pi:** The Raspberry Pi is a computing device used in represents the fog node. The reason behind selecting the Raspberry Pi is the cheap cost of the device on one hand, and since it has a moderate processing powers, therefore if the proof-of-concept carried on such device it can be scaled-up on higher processing devices.
- **Configurator Script:** The *Configurator Script* is used to generate a bulk of virtual sensor configurations at once and deploy them to the corresponding fog node as shown in figure 5.2. The *Configurator Script* can be configured to generated the desired number of virtual sensors and define the relations among them, where it generates the configurations on a designated folder on a computing device (i.e. laptop) other than the fog node, where this folder can be copied to the *Knowledge-Base* at the desired fog node.
- **Load Testing Script:** The *Load Testing Script* is responsible for simulating the generation of sensor data produced by physical sensors and send it to the message broker at each participating fog node as shown in figure 5.3. The *Load Testing script* is executed on a laptop.

- **Data Collection Tools:** *dstat* and *top* commands both are Linux-based commands executed to collect information about the CPU and memory usage.

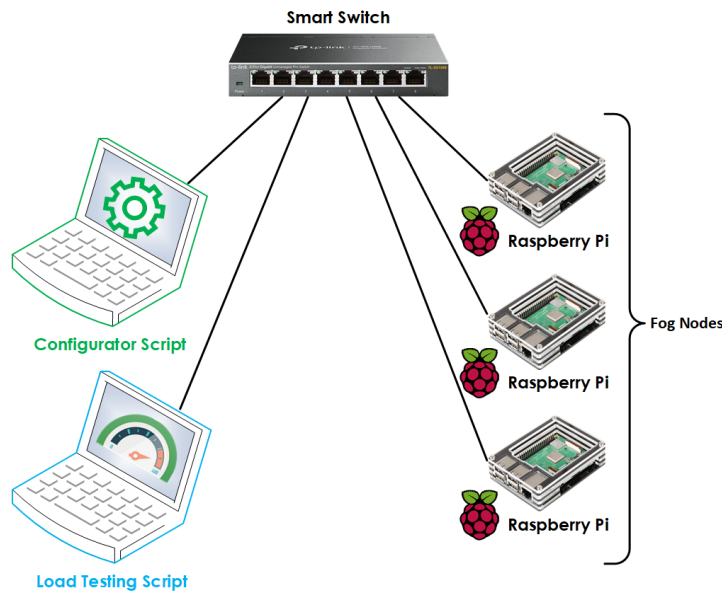


Figure 5.1: Evaluation Environment

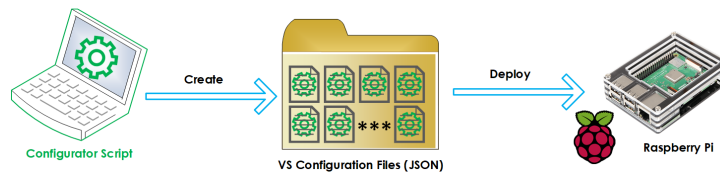


Figure 5.2: Evaluation Environment - Creating Configurations

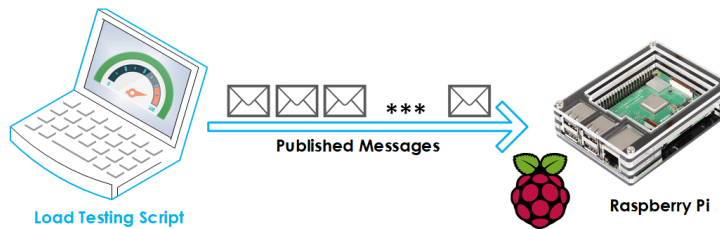


Figure 5.3: Evaluation Environment - Load Testing

## 5.2 Evaluation Factors and Metrics

### 5.2.1 Evaluation Factors

There are different factors related to the middleware that can impact positively or negatively the performance of the middleware. These factors relate to how the virtual sensors are configured,

structured, and deployed. The following describes each factor:

- **Virtual Sensor Frequency:** *Virtual Sensor Frequency* determines the rate at which the virtual sensor checks the input queues for newly arrived data. The *Virtual Sensor Frequency* is configured in seconds. This factor will be evaluated by running the virtual sensor at different frequencies and verify the impact over the performance.
- **Number of Virtual Sensors:** This factor refers to the number of virtual sensors deployed on a single fog node and on a single level and at the same frequency. The purpose of evaluating this factor is to see how the performance scales by increasing the number of virtual sensors and to determine the maximum number of virtual sensors the Raspberry Pi can handle.
- **Total Number of Input Sources per Virtual Sensor:** The virtual sensor can subscribe to one or more input sources to receive data. The purpose for evaluating the number of input sources per virtual sensor to see if the increase of the input sources have impact over performance.
- **Number of Virtual Sensors per Level:** The virtual sensors can be structured in multiple levels to perform aggregate functions. The purpose of evaluating this factor is see the impact of structuring virtual sensors into levels on the performance.
- **Distributed Processing of Virtual Sensors:** The main purpose of the middleware is to distribute processing of sensor data which can be achieved by deploying virtual sensors into multiple nodes. The purpose of evaluating this factor is to see the impact on performance when having the virtual sensors running on multiple nodes versus running them on a single node.

### 5.2.2 Evaluation Metrics

We used the following metrics to evaluate the middleware performance:

- CPU Utilization
- Memory Utilization

The time required to process sensor data is an important metric that can be used to understand the impact over the performance. However, it was impossible to measure the time due to different factors:

- Each message is timestamped once it is received at each virtual sensor, since each Raspberry Pi (fog node) in the current setup has a different clock/time that is setup manually, then it makes the timestamps inconsistent when messages send across different fog nodes.
- The messages are aggregated at different levels, which makes it impossible to link the message generated by the load tester with the messages aggregated at the fog node since the virtual sensor would aggregate different messages into a single message
- Since it is a custom-built middleware it was hard to find a software that can perform the load testing and measure the response time

### 5.3 Baseline

The main purpose of this scenario is to define the performance baseline for the CPU and memory of the fog node since the Raspbain OS would contain a preinstalled packages. Therefore, we executed two experiments monitoring the performance of the CPU and memory for six minutes independently. The first experiment we monitored the performance without running the middleware, only the message broker was running in the background with no activities. Where, the second experiment, we ran the middleware without instantiating any virtual sensor.

Table 5.1 shows the results of the two experiments. Experiment 1 defines the performance baseline for the Raspberry Pi at 3.86% for the CPU and 39.29% for the memory. In Experiment 2 it shows there is a little overhead in the CPU and memory utilization when we loaded the middleware. This increase is expected due to the fact that the middleware has to load the libraries that it requires to do its job. Figure 5.5 shows the performance gauge for the for the CPU and memory side by side which shows the slight increment in the utilization in Experiment 2, compared to figure 5.4.

Table 5.1: Performance Baseline

Experiment 1					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	-	-	-	-	-
<b>VS Inputs</b>	-	-	-	-	-
<b>Frequency</b>	-	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	0.25%	32.91%	3.86%	3.51%	3.26%
<b>Memory</b>	39.13%	40.09%	39.29%	39.29%	0.09%

Experiment 2					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	-	-	-	-	-
<b>VS Inputs</b>	-	-	-	-	-
<b>Frequency</b>	-	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	2.52%	31.31%	6.64%	5.77%	3.86%
<b>Memory</b>	43.99%	45.61%	44.13%	44.11%	0.17%

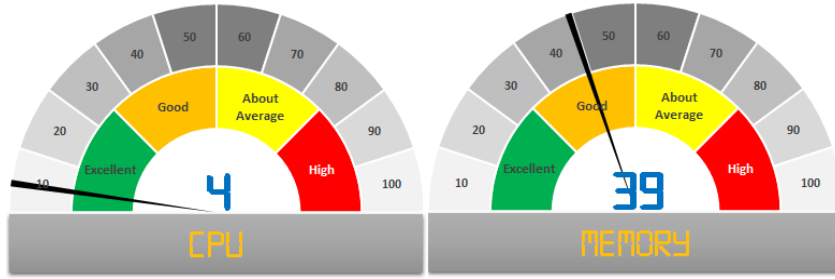


Figure 5.4: Baseline - Experiment 1 - Performance Evaluation for Running Raspberry Pi without the middleware

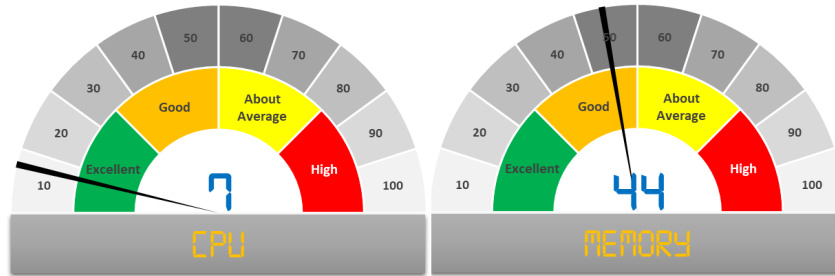
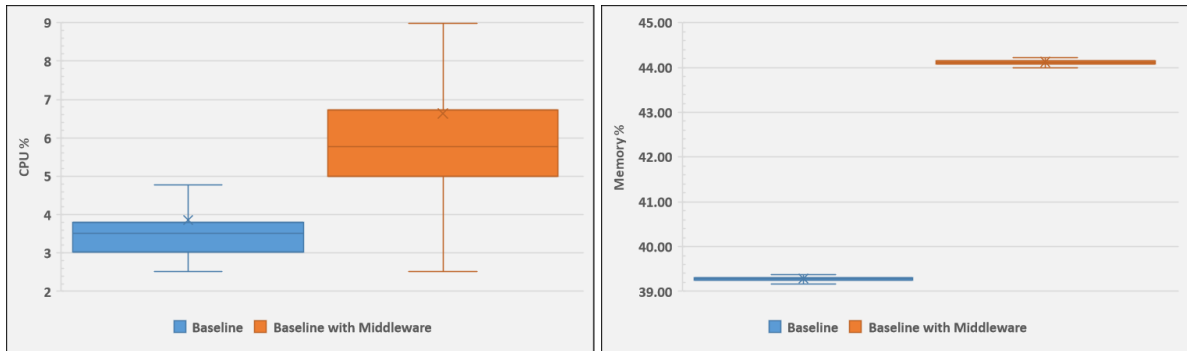


Figure 5.5: Baseline - Experiment 2 - Performance Evaluation for Running Raspberry Pi with the Middleware



(a) CPU Comparison

(b) Memory Comparison

Figure 5.6: Baseline - Performance Comparison

## 5.4 Evaluation Scenarios

In the evaluation we are going to perform 5 scenarios, each scenario evaluates one of the factors described in section 5.2.1 and carries multiple experiments to show the impact of the changing values on the performance. Table 5.2 explains the different scenarios to be executed.

Table 5.2: Evaluation Scenarios

Scenario	Factor	Description
Scenario 1	Virtual Sensor Frequency	This scenario consist of 5 experiments using 1 virtual sensor, each experiment runs at different frequency rate 1, 5, 10, 15, 60 seconds
Scenario 2	Total Number of Input Sources per Virtual Sensor	This scenario consist of 5 experiments using 1 virtual sensor having different numbers of input sources 1, 10, 50, 100, 150
Scenario 3	Number of Virtual Sensors	This scenario consist of 5 experiments using 1, 10, 50, 100, 150 virtual sensors. All virtual sensors have the same frequency rate and input sources
Scenario 4	Number of Virtual Sensors per Level	This scenario consist of 3 experiments. Each experiment uses 150 virtual sensors, with the difference that the virtual sensors are deployed into 1 level, 2 levels, or 3 levels
Scenario 5	Distributed Processing of Virtual Sensors	This scenario consist of 3 experiments using 150 virtual sensors. In the first experiment all of the virtual sensors are processed on one fog node, the second experiment processes the virtual sensors on two fog nodes, and the third experiment processes the virtual sensors on three fog nodes
Scenario 6	Virtual Sensor Frequency for multiple virtual sensors	This scenario consist of 4 experiments using a total of 111 virtual sensors, each experiment runs at different frequency rate 5, 10, 15, 60 seconds

### 5.4.1 Scenario 1

The purpose of this scenario is to evaluate the affect of the virtual sensor frequency over the performance. We defined in section 4.3.4 the use of *vsFrequency* which determines the rate at which virtual sensor checks the input queues. Therefore, in this scenario we use only one sensor that is deployed on level 1. The same sensor will be used in all experiments of this scenario with one difference is the frequency rate at which the virtual sensor will be running. We ran the virtual sensor at 1, 5, 10, 15, an 60 seconds. The results show that the CPU and memory utilization decreased by increasing the frequency rate of the virtual sensor. When we increased the frequency from 1 to 5 there was an enhancement in the CPU utilization from 13.51% to 12% or 1.5% enhancement in the CPU performance. However, when we increased the rate more there was a slight enhancement in the CPU performance and it almost became unnoticeable when we increased the frequency rate from 15 seconds to 60 seconds. Figure 5.7 through 5.11 show the performance gauge for CPU and memory for the experiments 1-5,

which shows a decrease in CPU and memory utilization when we increased the virtual sensor frequency.

Table 5.3: Impact of the frequency over the performance

Scenario 1 - Experiment 1					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	6.78%	57.03%	13.51%	12.41%	6.02%
<b>Memory</b>	46.27%	51.73%	50.28%	50.31%	0.39%

Scenario 1 - Experiment 2					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	5	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	5.54%	57.36%	12%	10.22%	6.30%
<b>Memory</b>	46.28%	51.68%	50.19%	50.21%	0.36%

Scenario 1 - Experiment 3					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	10	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	5.58%	78.21%	11.62%	9.62%	7.49%
<b>Memory</b>	46.50%	51.30%	49.89%	49.87%	0.28%

Scenario 1 - Experiment 4					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	15	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	5.06%	57.44%	10.74%	9.32%	0.16%
<b>Memory</b>	46.32%	50.85%	49.15%	49.17%	0.29%

Scenario 1 - Experiment 5					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	60	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	5.30%	48.72%	9.96%	8.33%	5.25%
<b>Memory</b>	46.69%	51.42%	49.83%	49.82%	0.28%

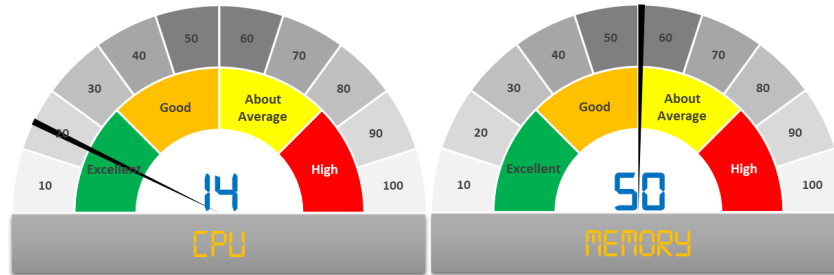


Figure 5.7: Scenario 1 - Experiment 1 - Frequency Performance Evaluation

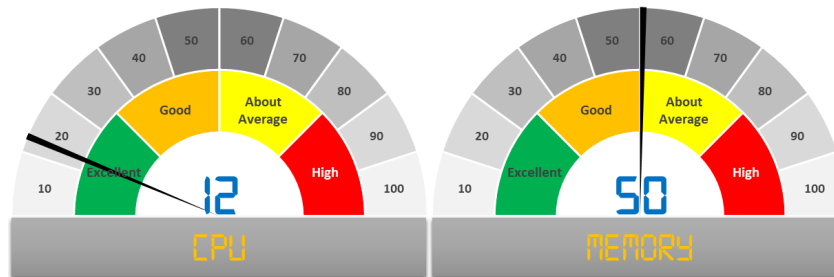


Figure 5.8: Scenario 1 - Experiment 2 - Frequency Performance Evaluation

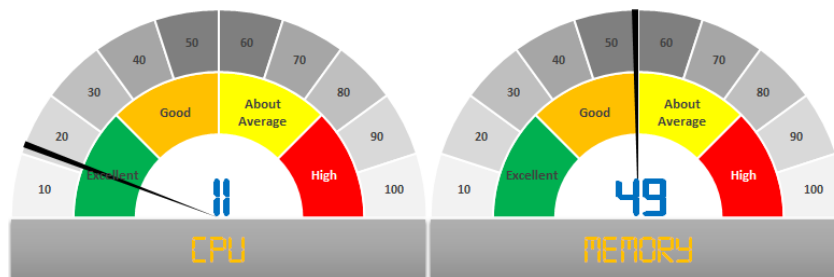


Figure 5.9: Scenario 1 - Experiment 3 - Frequency Performance Evaluation

## 5.4.2 Scenario 2

The virtual sensor may subscribe to one or more topics. The number of topics that the virtual sensor wishes to subscribe to, specifies the number of input sources that the virtual sensor should have. Therefore, the purpose of this scenario is to evaluate the impact of the number of input sources per virtual sensor over the performance. Each experiment would use the same



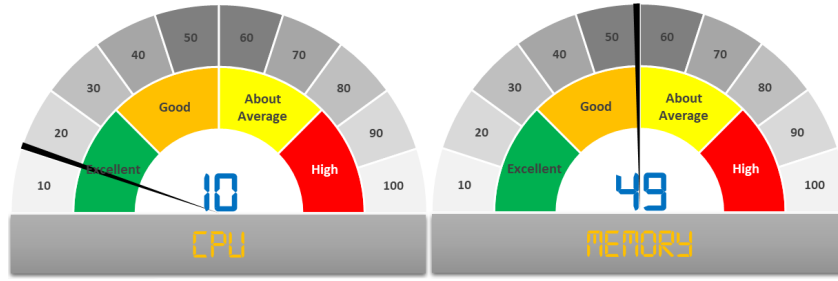


Figure 5.10: Scenario 1 - Experiment 4 - Frequency Performance Evaluation

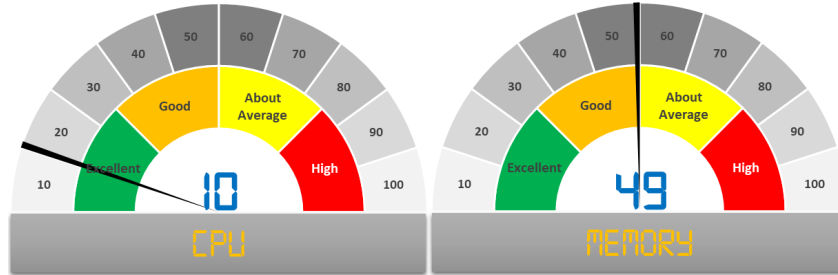
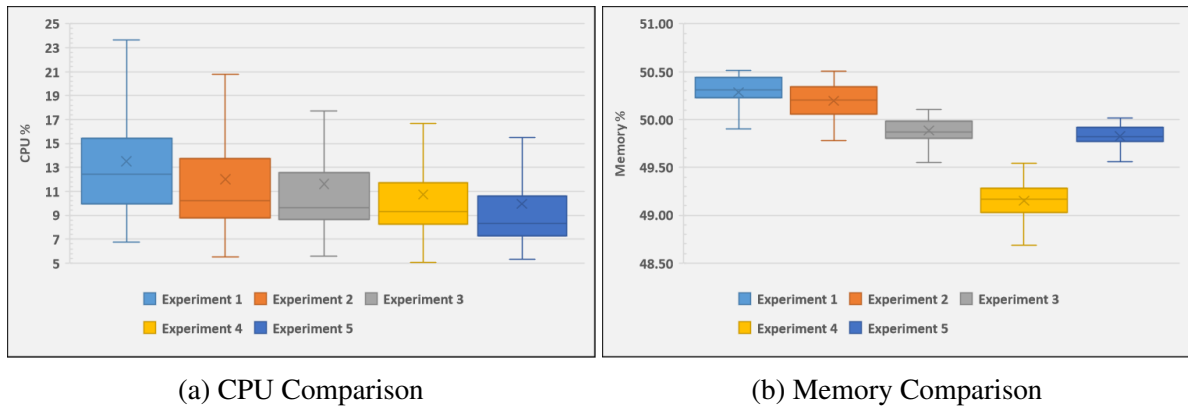


Figure 5.11: Scenario 1 - Experiment 5 - Frequency Performance Evaluation



(a) CPU Comparison

(b) Memory Comparison

Figure 5.12: Scenario 1 - Frequency Performance Comparison

virtual sensor with with the difference being the number of input sources. The virtual sensor was tested with 1, 10, 50, 100, and 150 input sources. As shown in table 5.4 we can notice that the CPU and memory utilization increased by increasing the number of input sources. This increase was due to increase in the number of topics declared at the message broke, since each input source represented an independent topic, which require the message broker to create an exchange for each topic. Furthermore, for each input source the virtual sensor needs to define a queue to store data received. We can infer that the more input sources that a virtual sensor have the more CPU and memory.

Table 5.4: Impact of number of inputs per virtual sensor over the performance

Scenario 2 - Experiment 1					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	6.78%	57.03%	13.51%	12.41%	6.02%
<b>Memory</b>	46.27%	51.73%	50.28%	50.31%	0.39%

Scenario 2 - Experiment 2					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	10	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	7.20%	66.84%	16.01%	15.19%	0.22%
<b>Memory</b>	44.47%	51.69%	49.65%	49.77%	0.60%

Scenario 2 - Experiment 3					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	50	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	15.80%	65.64%	27.22%	26.02%	7.10%
<b>Memory</b>	44.35%	56.54%	54.21%	54.60%	1.58%

Scenario 2 - Experiment 4					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	100	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	15.57%	82.52%	39.91%	38.17%	7.79%
<b>Memory</b>	48.34%	67.46%	63.25%	63.82%	2.99%

### 5.4.3 Scenario 3

The purpose of this scenario is to assess the impact of the total number of virtual sensor used on the performance. In this scenario we deploy a set of virtual sensors all at level 1. Each virtual sensor receives one input source and each has the same frequency rate. In this scenario we ran 5 different experiments using 1, 10, 50, 100, 150 virtual sensors. We noticed that as we increased the number of virtual sensors, higher the CPU and memory utilization. This behavior

Scenario 2 - Experiment 5					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	150	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	21.28%	85.90%	50.16%	48.30%	7.71%
<b>Memory</b>	44.91%	72.22%	66.85%	67.87%	4.81%

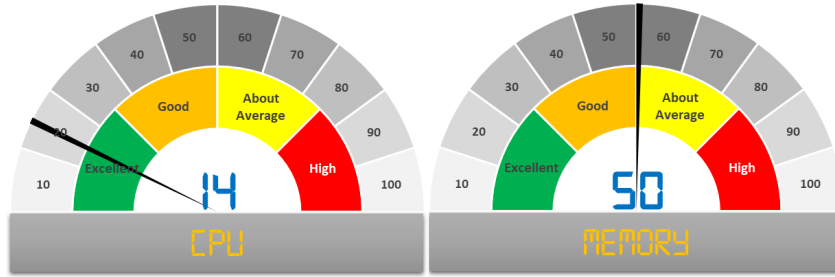


Figure 5.13: Scenario 2 - Experiment 1 - Number of Inputs per Virtual Sensor Performance Evaluation

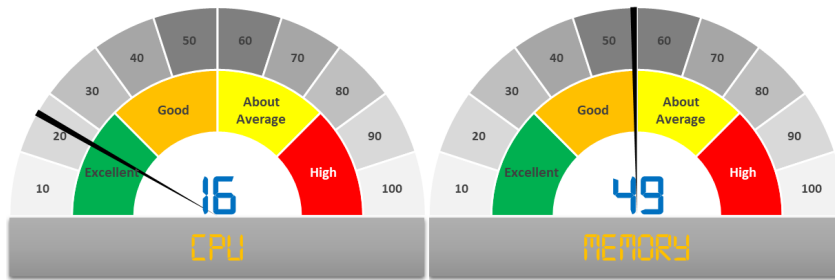


Figure 5.14: Scenario 2 - Experiment 2 - Number of Inputs per Virtual Sensor Performance Evaluation

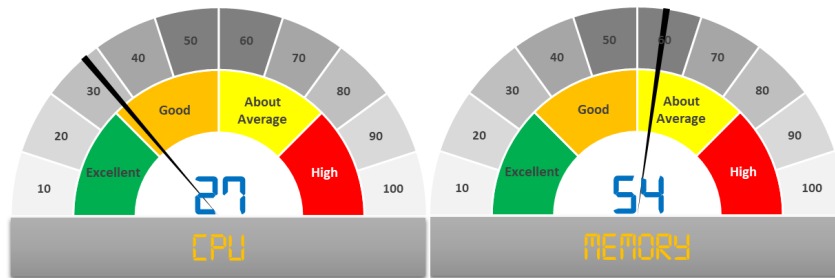


Figure 5.15: Scenario 2 - Experiment 3 - Number of Inputs per Virtual Sensor Performance Evaluation

is expected since each virtual sensor needs to define its input source and output destination.

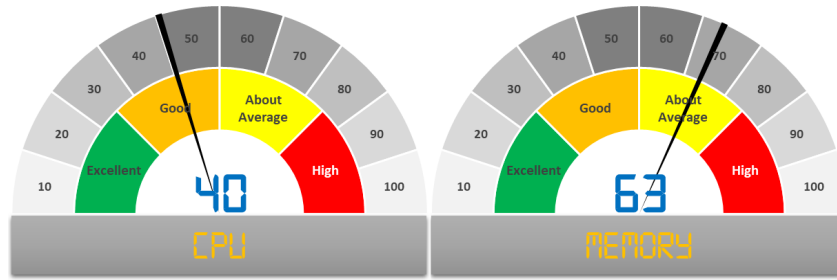


Figure 5.16: Scenario 2 - Experiment 4 - Number of Inputs per Virtual Sensor Performance Evaluation

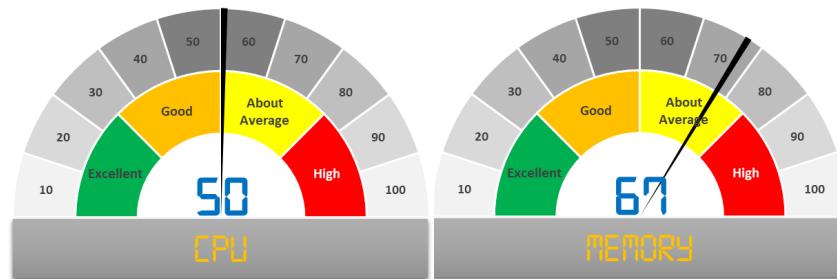
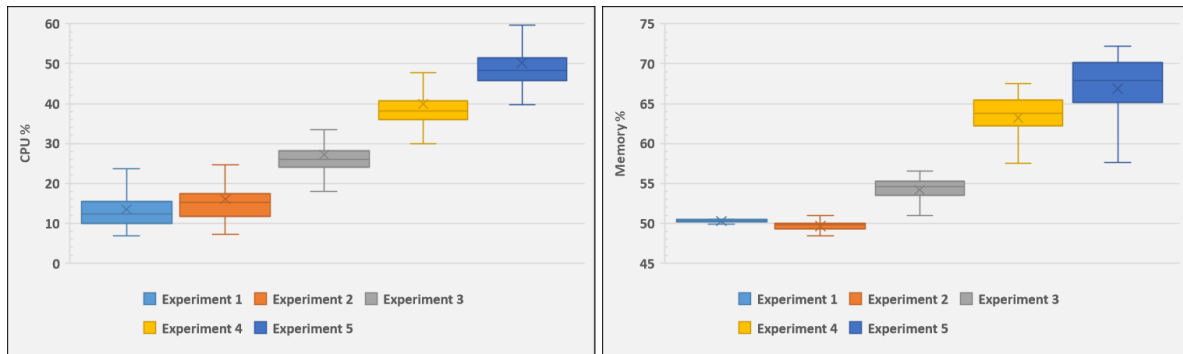


Figure 5.17: Scenario 2 - Experiment 5 - Number of Inputs per Virtual Sensor Performance Evaluation



(a) CPU Comparison

(b) Memory Comparison

Figure 5.18: Scenario 2 - Number of Inputs per Virtual Sensor Performance Comparison

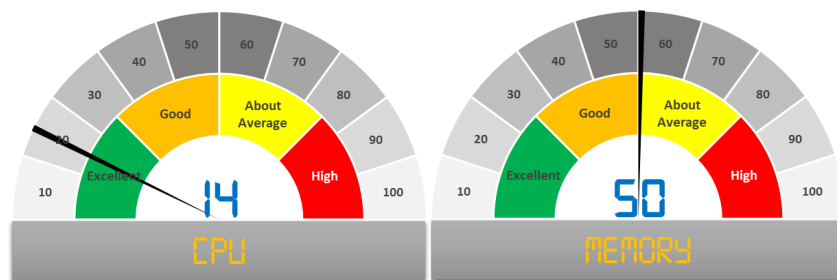


Figure 5.19: Scenario 3 - Experiment 1 - Number of Virtual Sensors Performance Evaluation

Table 5.5: Impact of the number of virtual sensors over performance

Scenario 3 - Experiment 1					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	1	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	6.78%	57.03%	13.51%	12.41%	6.02%
<b>Memory</b>	46.27%	51.73%	50.28%	50.31%	0.39%

Scenario 3 - Experiment 2					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	10	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	8.14%	85.90%	18.50%	17.77%	8.82%
<b>Memory</b>	45.71%	52.65%	50.71%	50.91%	0.63%

Scenario 3 - Experiment 3					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	50	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	17.75%	94.67%	31.14%	28.93%	10.75%
<b>Memory</b>	44.57%	59.14%	56.14%	56.43%	1.90%

Scenario 3 - Experiment 4					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	100	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	16.47%	95.90%	45.29%	45.26%	11.35%
<b>Memory</b>	45.05%	72.26%	67.08%	68.27%	4.39%

#### 5.4.4 Scenario 4

The purpose of this scenario is to evaluate the distribution of the virtual sensors across multiple levels on the same fog node. In this scenario we use 150 virtual sensors. In the first experiment we place all of them at level 1. In the second experiment we place 100 virtual sensor at level 1 and 50 virtual sensors at level 2. In experiment 3 we distribute the virtual sensors equally across three levels (50 virtual sensor each level). We noticed that in experiment 2 when we

Scenario 3 - Experiment 5					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	150	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	18.61%	98.22%	60.18%	57.25%	10.45%
<b>Memory</b>	45.38%	80.40%	73.25%	75.08%	6.29%

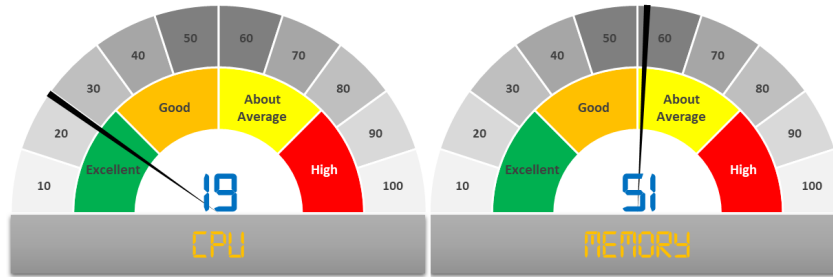


Figure 5.20: Scenario 3 - Experiment 2 - Number of Virtual Sensors Performance Evaluation

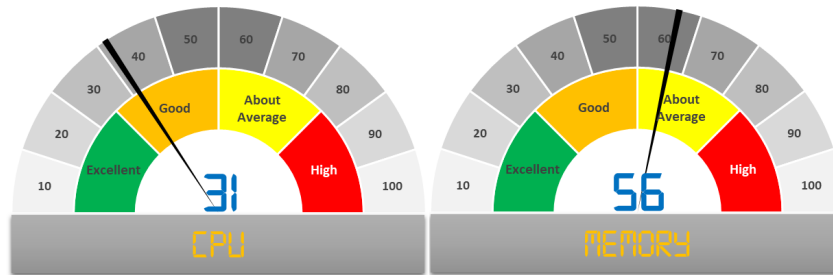


Figure 5.21: Scenario 3 - Experiment 3 - Number of Virtual Sensors Performance Evaluation

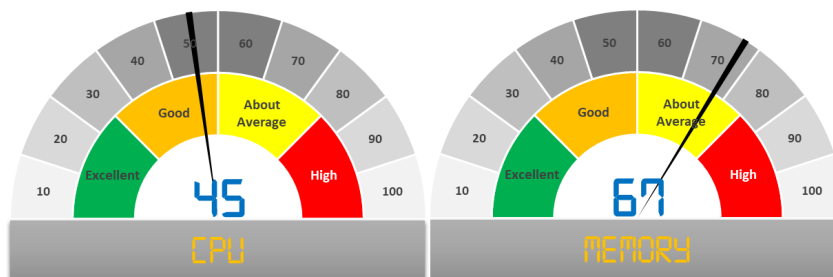


Figure 5.22: Scenario 3 - Experiment 4 - Number of Virtual Sensors Performance Evaluation

distribute the virtual sensors across 2 levels there was a slight increase in the CPU and memory utilization and this can be resulted from the fact that the number of input sources has increased at level 2 for each virtual sensor. However, when we distributed the virtual sensors into 3 levels we noticed that this has decreased the CPU utilization but the memory utilization almost remained the same compared to Experiment 1.

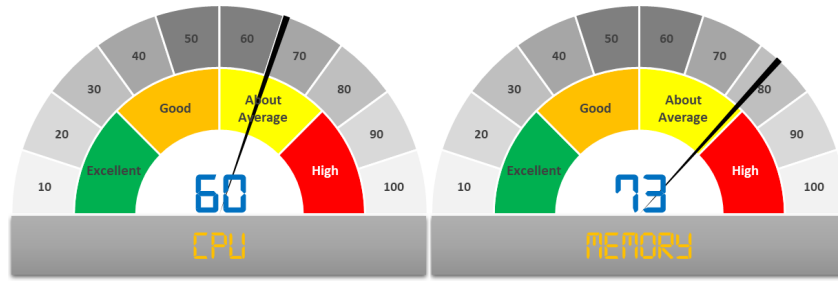
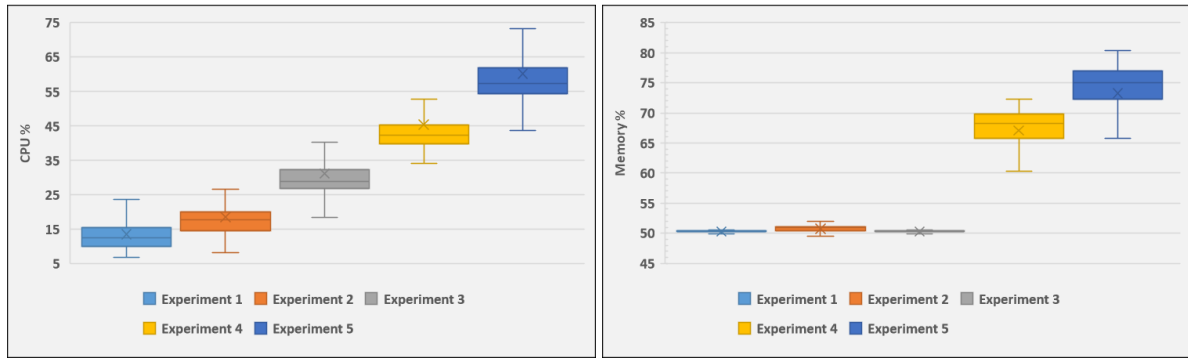


Figure 5.23: Scenario 3 - Experiment 5 - Number of Virtual Sensors Performance Evaluation



(a) CPU Comparison

(b) Memory Comparison

Figure 5.24: Scenario 3 - Number of Virtual Sensors Performance Comparison

Table 5.6: Impact of the number of levels over performance

Scenario 4 - Experiment 1					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	150	-	-	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	1	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	18.61%	98.22%	60.18%	57.25%	10.45%
<b>Memory</b>	45.38%	80.40%	73.25%	75.08%	6.29%

Scenario 4 - Experiment 2					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	100	50	-	-	-
<b>VS Inputs</b>	1	2	-	-	-
<b>Frequency</b>	1	1	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	15.09%	99.25%	60.20%	57.44%	10.90%
<b>Memory</b>	44.98%	84.37%	76.38%	77.95%	7.30%

Scenario 4 - Experiment 3					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	50	50	50	-	-
<b>VS Inputs</b>	1	1	1	-	-
<b>Frequency</b>	1	1	1	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	16.43%	98.00%	47.33%	47.33%	13.58%
<b>Memory</b>	48.12%	80.58%	74.02%	75.75%	5.83%

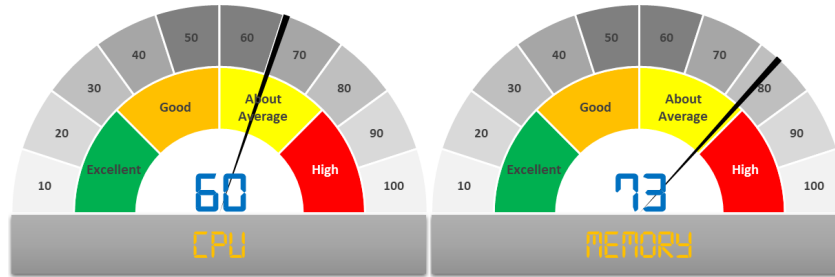


Figure 5.25: Scenario 4 - Experiment 1 - Number of Levels Performance Evaluation

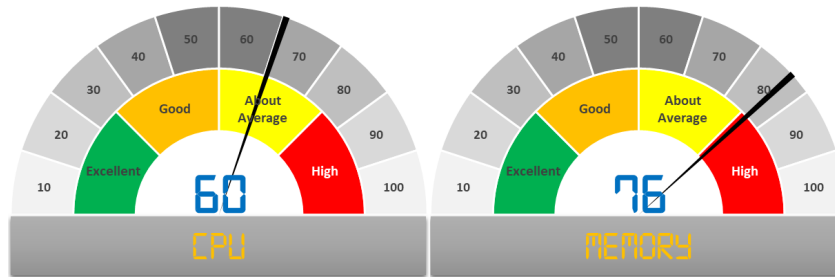


Figure 5.26: Scenario 4 - Experiment 2 - Number of Levels Performance Evaluation

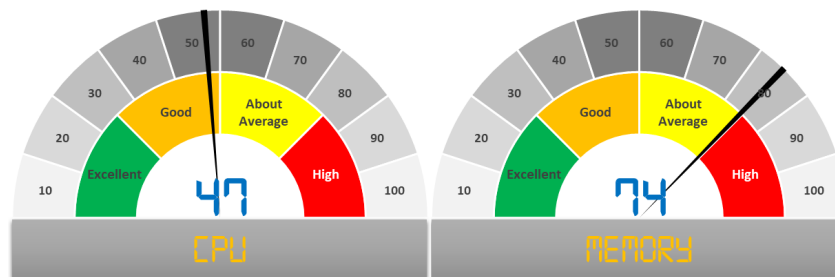


Figure 5.27: Scenario 4 - Experiment 3 - Number of Levels Performance Evaluation

### 5.4.5 Scenario 5

The purpose of this scenario is to show the impact of distributing processing of virtual sensors on the performance. For this scenario, we ran three experiments. The first experiment has 150 virtual sensors at level 1 in a single fog node. In the second experiment there are 100 virtual sensors hosted on fog 1 and 50 virtual sensors hosted on fog 2, receiving inputs from the 100



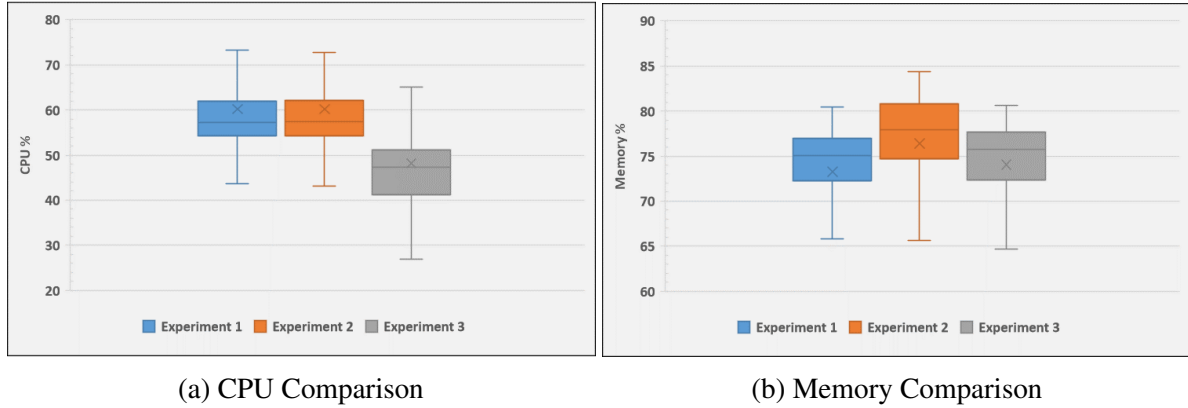


Figure 5.28: Scenario 4 - Performance Comparison

virtual sensor at fog 1, which means each virtual sensor would receive two inputs. In the third experiments, there are 50 virtual sensors hosted on a fog 1 on level 1, 50 virtual sensors hosted on a fog 2 on level 2 and 50 virtual sensors hosted on a fog 3 at level 3. The virtual sensors at fog 2 receive data from the virtual sensors at fog 1, and the virtual sensors at fog 3 receive data from virtual sensors at fog 2. Table 5.7 shows the results of the three experiments. We can notice that both the CPU and memory utilization is reduced when we distribute the processing on other nodes.

Table 5.7: Impact of distribute processing over the performance

Scenario 5 - Experiment 1						
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>	
Node 1	<b>VS Used</b>	150	-	-	-	-
	<b>VS Inputs</b>	1	-	-	-	-
	<b>Frequency</b>	1	-	-	-	-
		<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
	<b>CPU</b>	15.45%	96.44%	57.81%	54.77%	10.93%
	<b>Memory</b>	44.12%	79.44%	73.08%	74.08%	6.38%

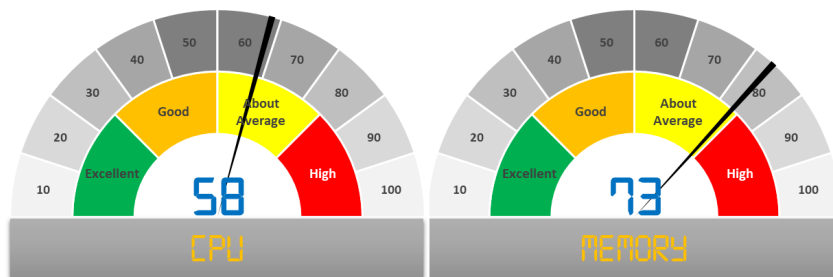
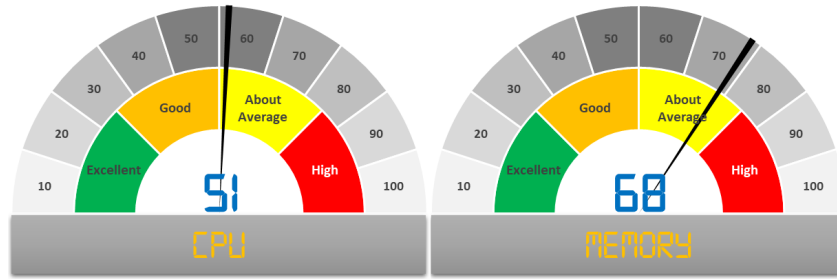


Figure 5.29: Scenario 5 - Experiment 1 - Performance Evaluation for 150 VS Running on Single Node

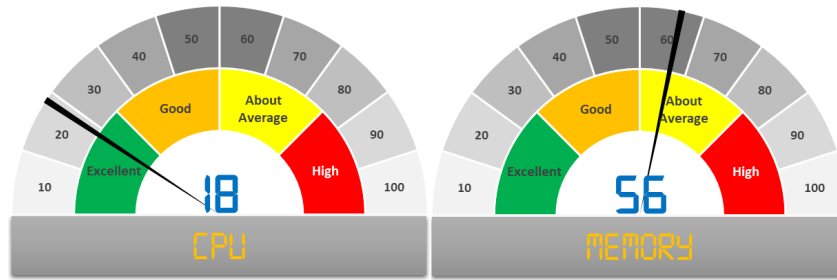
Scenario 5 - Experiment 2						
Node 1	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>	
	<b>VS Used</b>	100	-	-	-	
	<b>VS Inputs</b>	1	-	-	-	
	<b>Frequency</b>	1	-	-	-	
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>S.D.</u>	
	<b>CPU</b>	11.93%	96.70%	50.67%	48.96%	11.42%
	<b>Memory</b>	44.95%	73.72%	67.86%	69.16%	4.85%
Node 2	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>	
	<b>VS Used</b>	-	50	-	-	
	<b>VS Inputs</b>	-	1	-	-	
	<b>Frequency</b>	-	1	-	-	
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>S.D.</u>	
	<b>CPU</b>	7.32%	87.88%	18.16%	17.13%	7.05%
	<b>Memory</b>	43.44%	59.64%	56.47%	56.70%	1.75%
Scenario 5 - Experiment 3						
Node 1	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>	
	<b>VS Used</b>	50	-	-	-	
	<b>VS Inputs</b>	1	-	-	-	
	<b>Frequency</b>	1	-	-	-	
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>S.D.</u>	
	<b>CPU</b>	9.65%	92.98%	31.78%	29.06%	11.93%
	<b>Memory</b>	45.19%	63.17%	59.26%	59.74%	2.49%
Node 2	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>	
	<b>VS Used</b>	-	50	-	-	
	<b>VS Inputs</b>	-	1	-	-	
	<b>Frequency</b>	-	1	-	-	
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>S.D.</u>	
	<b>CPU</b>	6.13%	91.88%	16.18%	14.68%	0.44%
	<b>Memory</b>	44.70%	59.11%	56.76%	57.27%	1.70%
Node 3	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>	
	<b>VS Used</b>	-	-	50	-	
	<b>VS Inputs</b>	-	-	1	-	
	<b>Frequency</b>	-	-	1	-	
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>S.D.</u>	
	<b>CPU</b>	4.57%	89.06%	12.43%	11.17%	8.35%
	<b>Memory</b>	46.27%	57.63%	55.54%	55.67%	1.10%

### 5.4.6 Scenario 6

The purpose of this scenario is to evaluate the affect of the virtual sensor frequency. We defined in section 4.3.4 the use of *vsFrequency* which determines the rate at which virtual sensor checks the input queues. This scenario uses 111 virtual sensors that is deployed on three levels. The



(a) Node 1



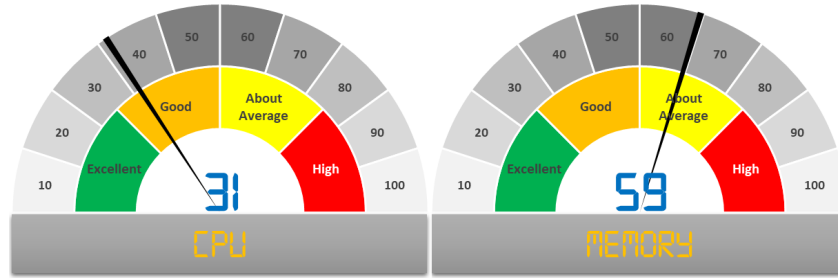
(b) Node 2

Figure 5.30: Scenario 5 - Experiment 2 - Performance Evaluation for 150 VS distributed into two nodes

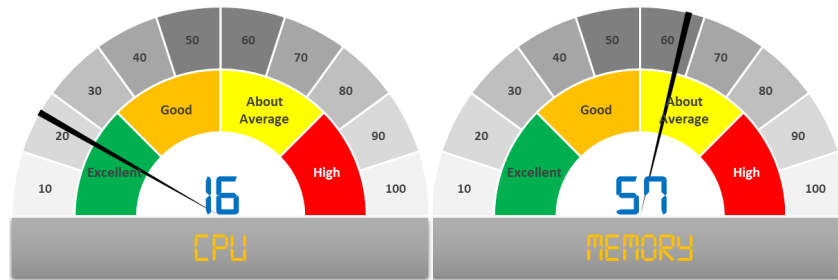
same number of virtual sensors will be used in all experiments of this scenario with one difference is the frequency rate at which the virtual sensor will be running. We ran the virtual sensor at a frequency rate of 5, 10, 15, and 60 seconds. The results show that the CPU and memory utilization decreased by increasing the frequency rate of the virtual sensor. When we increased the frequency from 5 to 10 there was an enhancement in the CPU utilization from 36.47% to 34.65% or 2% enhancement in the CPU performance. When we increased the frequency rate from 15 seconds to 60 seconds there was an enhancement in the CPU utilization from 31.66% to 28.66% or 3% enhancement in the CPU performance. Figure 5.32 through 5.35 show the performance gauge for CPU and memory for the experiments 1-4, which shows an improvement in CPU and memory utilization when we increased the virtual sensor frequency.

Table 5.8: Impact of the frequency using 111 virtual sensors over the performance

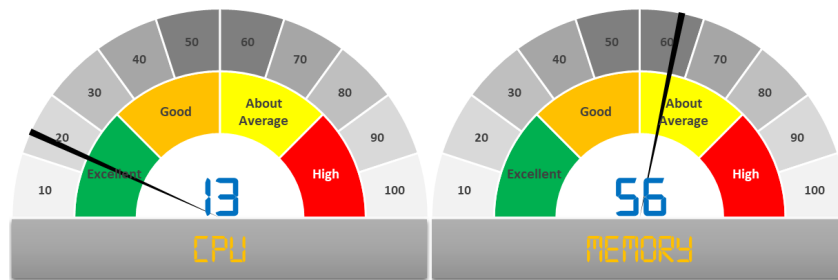
Scenario 6 - Experiment 1					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	100	10	1	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	5	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	16.80%	95.89%	36.47%	32.36%	15.40%
<b>Memory</b>	49.79%	84.01%	78.48%	80.26%	6.30%



(a) Node 1



(b) Node 2



(c) Node 3

Figure 5.31: Scenario 5 - Experiment 3 - Performance Evaluation for 150 VS distributed into three nodes

Scenario 6 - Experiment 2					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	100	10	1	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	10	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	11.39%	95.68%	34.65%	31.04%	14.68%
<b>Memory</b>	50.65%	81.45%	76.15%	78.46%	5.71%

## 5.5 Discussion of Results

When the middleware starts, there are several tasks that take place. These include reading all sensor configurations (JSON), instantiation of virtual sensors, and create topics associated with each virtual sensor. This cause the middleware to reach the maximum number displayed in the results. Once the middleware finished processing and instanctiating all virtual sensor the load

Scenario 6 - Experiment 3					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	100	10	1	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	15	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	11.92%	96.68%	31.66%	26.65%	15.87%
<b>Memory</b>	45.32%	74.55%	69.54%	71.48%	5.37%

Scenario 6 - Experiment 4					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
<b>VS Used</b>	100	10	1	-	-
<b>VS Inputs</b>	1	-	-	-	-
<b>Frequency</b>	60	-	-	-	-
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>SD.</u>
<b>CPU</b>	9.85%	95.67%	28.66%	23.66%	15.99%
<b>Memory</b>	46.55%	71.31%	65.96%	67.11%	4.14%

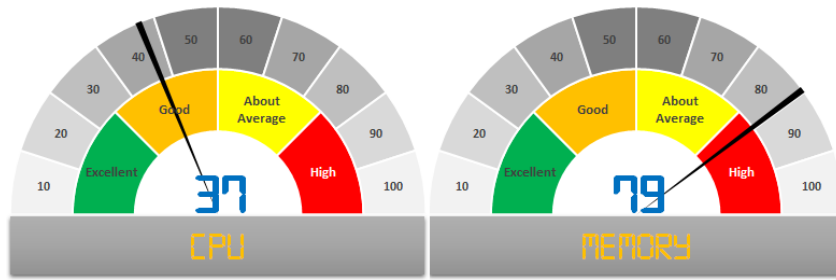


Figure 5.32: Scenario 6 - Experiment 1 - Frequency Performance Evaluation of 111 Virtual Sensors

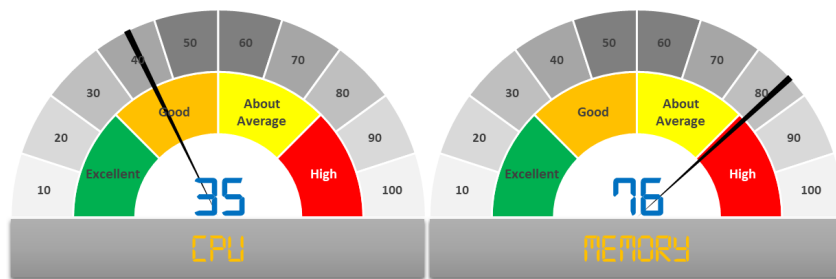


Figure 5.33: Scenario 6 - Experiment 2 - Frequency Performance Evaluation of 111 Virtual Sensors

on CPU and Memory is decreased.

The collection of evaluation results for all experiments executed only once manually using Linux commands for 6 minutes. The commands were started directly after starting the middle-ware. However, it is hard to tell the exact start point of the results collection compared to the start point of the middleware. Therefore, if multiple runs executed, there might be a slight vari-

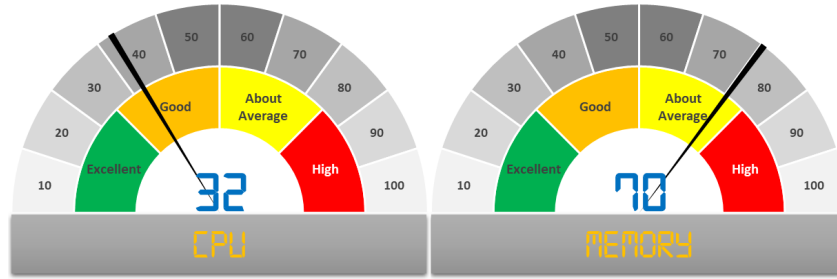


Figure 5.34: Scenario 6 - Experiment 3 - Frequency Performance Evaluation of 111 Virtual Sensors

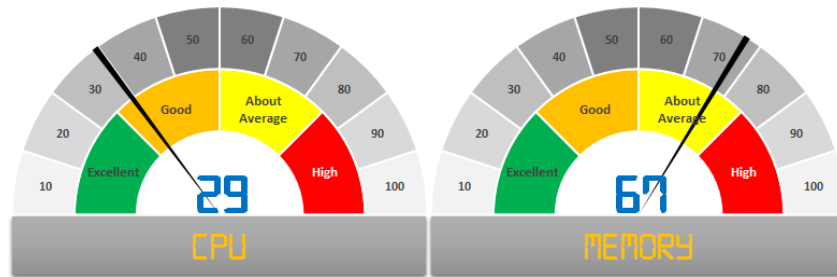
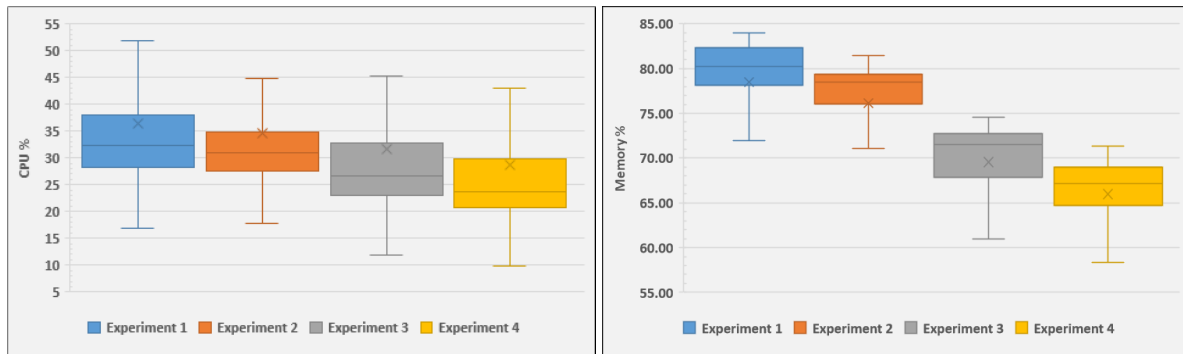


Figure 5.35: Scenario 6 - Experiment 4 - Frequency Performance Evaluation of 111 Virtual Sensors



(a) CPU Comparison

(b) Memory Comparison

Figure 5.36: Scenario 6 - Frequency Performance Comparison for 111 Virtual Sensors

ation in the minimum, maximum, and average of the CPU and memory performance. Another reason behind this slight variation is that the operating system has other software packages running in the background which might have some processes running that might impact the evaluation numbers.

# Chapter 6

## Conclusion and Future Work

### 6.1 Conclusion

The Internet of Things (IoT) is a massively growing field where billions of devices are expected to be connected to the internet. These devices co-exists in a fast changing heterogeneous environment that uses different protocols, operating systems, and hardware. Yet, these devices need to interact with different applications to process sensor data and perform tasks efficiently. Moreover, most applications in a rapid growing world require a real-time processing of data to provide real-time responses. This produce a burden on application developers to communicate with IoT devices which use different protocols and produce large amounts of data. The middleware was able to reduce the development efforts by extending the physical sensor activities to provide extended functions performed on the fog nodes rather than on the cloud or let the applications do the actual processing of sensor data. The middleware provided the abstraction required by application through the use of adapters to connect to the physical sensors and through the use of the publish-subscribe design pattern. Furthermore, the middleware considered the different needs of the applications by providing the choice for the applications to subscribe to sensor data at fog nodes or at the cloud. Therefore, it helped applications to acquire sensor data in a real time or near-real time. In addition, the use of the fog computing helped the middleware to distribute the processing of sensor data to produce results quickly and minimize the need for powerful node to do the processing. Finally, the middleware through the use of fog computing reduced the amount of the traffic transferred to the cloud by processing sensor data on the fog nodes and allowing applications to subscribe to the produced data.

In this thesis, we propose virtual sensor middleware architecture to overcome the different challenges. The middleware focused on the following:

- Reduce the development efforts required by application developers through the use of virtual sensors, which extends the functionalities of the IoT devices needed by the applications. For example, a virtual sensor can be used to connect to multiple temperature sensors and compute the average temperature. Therefore, it reduced the development efforts to establish and maintain connections to the physical sensors then compute the average temperature.
- Provide a real-time or near real-time sensor data by processing the data at the fog node and allowing applications to receive sensor data from the fog node.

- Eliminate the dependency between IoT devices and applications through the use of the publish-subscribe design pattern, where data producers don't need to maintain information about data consumers.
- Introduce new IoT devices to the applications easily regardless of the underlying protocol through the use of adapters or by allowing capable IoT devices to send data to the message broker.
- Bring up an adaptive middleware that allows modifications of virtual sensor configuration and introduce new virtual sensor through configurable settings that can be updated during the run-time of the applications.
- Introduce the use of a fault-handling policy to deal with the absence of sensor data when a physical sensor is down

## 6.2 Future Work

In this thesis, we built virtual sensor middleware to correspond to different challenges discussed earlier. The middleware was tested and operated well in different scenarios. Despite the functionalities provided by the middleware, there is always a space for improvement. The middleware can be extended to support more features and enhance existing functions

First, the middleware uses the publish-subscribe design pattern to exchange information between virtual sensors and applications. The publish-subscribe design pattern currently supports the use of topics to exchange information. This can be extended to make virtual sensors content-aware by allowing virtual sensors to apply filters to receive data selectively based on the content. This feature would allow virtual sensors to filter data and would minimize the traffic sent over the network.

Second, sensor data processing can be a very sophisticated problem. Especially due to errors produced by physical sensors. The middleware currently applies a simple fault-handling technique that might not be sufficient to cover all cases. Therefore, in more complex scenarios data/sensor fusion [27] [20] might be required to process sensor data to make sure the produced data have a better quality and to produce new derived data type from the input sources. For example, if sensors of different types are deployed inside a cargo container that transfers food, the container should preserve the food under certain conditions. Suppose, humidity and temperature sensors are used. By applying sensor fusion on the data we can improve the quality of the readings and produce a new reading that tells us if the conditions inside the container are suitable to preserve the food.

Third, currently applications should have knowledge about the published topics in order to subscribe to these topics. Therefore, applications should be able to allocate topics by searching a topics-catalog which contains a description of each topic, which allows applications to independently select the topics and subscribe to.

Fourth, subscribers and publishers require username and password to access the message broker. This can cause a security issue and can be difficult to manage since it requires the maintenance of different applications. We will explore the use of a public-private key.



Fifth, the current architecture assumes the administrator specifies the deployment location of the virtual sensors. It would be more convenient if the middleware is able to determine the best location to deploy the virtual sensor based on multiple factors: The desired response time, the distance from the physical sensor, the function executed by the virtual sensor, and the processing power of the fog node.

Sixth, a more sophisticated dashboards should be developed to support advanced monitoring and provide a GUI and advanced reporting tools.

# Bibliography

- [1] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A Middleware For Fast and Flexible Sensor Network Deployment. *Proceedings of the 32nd international conference on Very large data bases*, pages 1199–1202, 2006.
- [2] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, 17(4):2347–2376, 2015.
- [3] Sarfraz Alam, Mohammad MR Chowdhury, and Josef Noll. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–6. IEEE, 2010.
- [4] ZigBee Alliance. What is zigbee? 2015. <https://www.zigbee.org/what-is-zigbee/494-2/>.
- [5] Apache Foundation. Apache commons net. Product documentation, 2019. <https://commons.apache.org/proper/commons-net/> [March 15, 2019].
- [6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
- [7] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Dave Langworthy, Ashok Malhotra, et al. Web services policy 1.2 - framework (ws-policy). W3C specification, 2006. <http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/>.
- [8] Bootstrap. Bootstrap. Product documentation, 2019. <https://getbootstrap.com/docs/3.3/> [March 14, 2019].
- [9] Carsten Bormann, Angelo P Castellani, and Zach Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.
- [10] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: a survey. *Future generation computer systems*, 56:684–700, 2016.

- [11] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, et al. Web services addressing (ws-addressing). W3C specification, 2004. <http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/>.
- [12] Doug Davis, Ashok Malhotra, Wu Chou, and Katy Warr. Web services metadata exchange (WS-metadataexchange). W3C recommendation, 2011. <http://www.w3.org/TR/2011/REC-ws-metadata-exchange-20111213/>.
- [13] JSON Editor. Json editor. Product documentation, 2019. <https://github.com/josdejong/jsoneditor> [March 14, 2019].
- [14] Patrick Th Eugster, Rachid Guerraoui, and Joe Sventek. Type-based publish/subscribe. Technical report, 2000.
- [15] FIWARE. Introduction - quick fiware tour guide. FIWARE documentation, 2018. <http://fiwaretourguide.readthedocs.io/en/latest/connection-to-the-internet-of-things/introduction/>.
- [16] FIWARE. Reading data from iot devices - quick fiware tour guide. FIWARE documentation, 2018. <http://fiwaretourguide.readthedocs.io/en/latest/connection-to-the-internet-of-things/how-to-read-measures-captured-from-iot-devices/>.
- [17] Apache Foundation. Apache log4j 2. Apache website, 2019. <https://logging.apache.org/log4j/2.x/> [Feb 25, 2019].
- [18] Raspberry Pi Foundation. Raspberry pi 3 model b+. Product documentation, 2019. <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/> [Feb 25, 2019].
- [19] Raspberry Pi Foundation. Raspbian. Product documentation, 2019. <https://www.raspberrypi.org/downloads/raspbian/> [March 14, 2019].
- [20] David L Hall and James Llinas. An introduction to multisensor data fusion. *Proceedings of the IEEE*, 85(1):6–23, 1997.
- [21] Daniel Happ, Niels Karowski, Thomas Menzel, Vlado Handziski, and Adam Wolisz. Meeting iot platform requirements with open pub/sub solutions. *Annals of Telecommunications*, 72(1-2):41–52, 2017.
- [22] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*, pages 791–798. IEEE, 2008.
- [23] François Jammes, Antoine Mensch, and Harm Smit. Service-oriented device communications using the devices profile for web services. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–8. ACM, 2005.

- [24] Ram Jeyaraman, Vipul Modi, Dan Driscoll, Geoff Bullen, and Toby Nixon. Understanding devices profile for web services, ws-discovery, and soap-over-udp. 2008. [http://msdn.microsoft.com/en-us/library/dd179231.aspx#\\_Toc208829801](http://msdn.microsoft.com/en-us/library/dd179231.aspx#_Toc208829801).
- [25] The jQuery Foundation. What is jquery. Product documentation, 2019. <https://jquery.com/> [March 14, 2019].
- [26] Danh Le-Phuoc, Hoan Quoc Nguyen-Mau, Josiane Xavier Parreira, and Manfred Hauswirth. A middleware framework for scalable management of linked streams. *Web Semantics: Science, Services and Agents on the World Wide Web*, 16:42–51, 2012.
- [27] Theresa W Long, Emil L Hanzevack, and William L Bynum. Sensor fusion and failure detection using virtual sensors. In *Proceedings of the 1999 American Control Conference (Cat. No. 99CH36251)*, volume 4, pages 2417–2421. IEEE, 1999.
- [28] Sanjay Madria, Vimal Kumar, and Rashmi Dalvi. Sensor cloud: A cloud of virtual sensors. *IEEE software*, 31(2):70–77, 2014.
- [29] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.
- [30] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82. ACM, 2007.
- [31] Mark Nottingham. Web linking. IETF standards track, 2010. <http://tools.ietf.org/html/rfc5988>.
- [32] Oracle. Java api for json processing. JSR 374 specification, 2015. <https://javaee.github.io/jsonp/> [Feb 25, 2019].
- [33] R. Salz P. Leach, M. Mealling. A universally unique identifier (uuid) urn namespace. IETF standards track, 2005. <https://tools.ietf.org/html/rfc4122> [Feb 25, 2019].
- [34] RabbitMQ. Amqp 0-9-1 model explained. RabbitMQ documentation, 2019. <https://www.rabbitmq.com/tutorials/amqp-concepts.html#amqp-model> [Feb 25, 2019].
- [35] RabbitMQ. Rabbitmq java client library. Technical report, 2019. <https://www.rabbitmq.com/java-client.html> [Feb 25, 2019].
- [36] RabbitMQ. Which protocols does rabbitmq support? Technical report, 2019. <https://www.rabbitmq.com/protocols.html> [Feb 25, 2019].
- [37] Sajjad Hussain Shah and Ilyas Yaqoob. A survey: Internet of things (iot) technologies, applications and challenges. In *Smart Energy Grid Engineering (SEGE), 2016 IEEE*, pages 381–385. IEEE, 2016.
- [38] Zach Shelby. Constrained restful environments (CoRE) link format. IETF standards track, 2012. <http://tools.ietf.org/html/rfc6690.html>.

- [39] Zach Shelby. The constrained application protocol (CoAP). IETF standards track, 2016. <http://tools.ietf.org/html/rfc7252>.
- [40] Haiying Shen. Content-based publish/subscribe systems. In *Handbook of Peer-to-Peer Networking*. Springer US, 2010.
- [41] Ayman Sleman and Reinhard Moeller. Integration of wireless sensor network services into other home and industrial networks; using device profile for web services (dpws). In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1–5. IEEE, 2008.

# Appendix A

## Virtual Sensor Configuration File

```
1 {
2     "vsName" : "VS-01-000001",
3     "vsID" : "01-000001",
4     "vsType" : "Processor",
5     "vsFrequency" : "1",
6     "vsInitialDelay" : "1",
7     "vsAggregateFunction" : "AVG",
8     "vsNode" : {
9         "hostname" : "vsm-fognode01",
10        "hostip" : "172.18.17.11"
11    },
12    "publishExchange" : {
13        "exchName" : "PE-01-000001",
14        "exchType" : "fanout",
15        "exchNode" : {
16            "exchHostName" : "vsm-fognode01",
17            "exchHostIP" : "172.18.17.11"
18        },
19        "exchUsername" : "admin",
20        "exchPassword" : "admin",
21    },
22    "subscribeExchanges" : [
23        {
24            "exchName" : "PE-00-000001",
25            "exchType" : "fanout",
26            "exchQueueName" : "SE-Queue-01-000001
27            --00-000001",
28            "exchNode" : {
29                "exchHostName" : "vsm-fognode0
30                1",
31                "exchHostIP" : "172.18.17.11"
32            },
33        }
34    ]
35 }
```

```
31         "exchUsername" : "admin",
32         "exchPassword" : "admin",
33     },
34     {
35         "exchName" : "PE-00-000002",
36         "exchType" : "fanout",
37         "exchQueueName" : "SE-Queue-01-000001
38             --00-000002",
39         "exchNode" : {
40             "exchHostName" : "vsm-fognode01",
41             "exchHostIP" : "172.18.17.11"
42         },
43         "exchUsername" : "admin",
44         "exchPassword" : "admin",
45     }
46 ],
47 "faultHandlerPolicyID" : "policy1",
48 "saveToDB" : true,
49 "database" : {
50     "dbName" : "mydb",
51     "driverURL" : "jdbc:sqlserver://localhost:3306",
52     "dbUsername" : "admin",
53     "dbPassword" : "admin"
54 }
```

# Appendix B

## Reading Evaluations Tables and Figures

Appendix B explains how to read the tables used in chapter 5 and explains the *Box and Whisker* Diagram.

### B.1 Reading Evaluation Table

Figure B.1 shows the evaluation table, which consists of two parts:

- **Part 1:** This part is illustrated in figure B.2 describes the different configurations used to setup the virtual sensor at each level. There are three configurations used at each level:
  - *VS Used* refers to the number of virtual sensor used at the mentioned level.
  - *VS Inputs* refers to the number of input sources that any given virtual sensor have at the mentioned level.
  - *Frequency* refers to the frequency rate at which any given sensor in the mentioned level checks the input queues
- **Part 2:** This part illustrated in figure B.3 describes the statistical data collected from the full experiment for both the CPU and Memory utilization.

Scenario 1 - Experiment 1					
	<u>L1</u>	<u>L2</u>	<u>L3</u>	<u>L4</u>	<u>L5</u>
VS Used					
VS Inputs					
Frequency					
	<u>Min</u>	<u>Max</u>	<u>Average</u>	<u>Median</u>	<u>S.D.</u>
CPU					
Memory					

Part 1

Part 2

Figure B.1: Evaluation Table Parts



Scenario 1 - Experiment 1					
	L1	L2	L3	L4	L5
VS Used	1	-	Not used	-	-
VS Inputs	1	-	-	-	-
Frequency	1	-	-	-	-
	Min	Max	Average	Median	S.D.
CPU					
Memory					

Figure B.2: Evaluation Table Part 1

Scenario 1 - Experiment 1					
	L1	L2	L3	L4	L5
VS Used	1	-	-	-	-
VS Inputs	1	-	-	-	-
Frequency	1	-	-	-	-
	Min	Max	Average	Median	S.D.
CPU	6.78%	57.03%	13.51%	12.41%	6.02%
Memory	46.27%	51.73%	50.28%	50.31%	0.39%

Figure B.3: Evaluation Table Part 2

## B.2 Box and Whisker Plot

Box and Whisker is a diagram consist of a box with two lines drafted from the middle of the box called whisker, at the end of each line there is a cross-hair that forms a T-shape along with the line. Each part of the diagram represents a statistical information in a data set as shown in figure B.4.

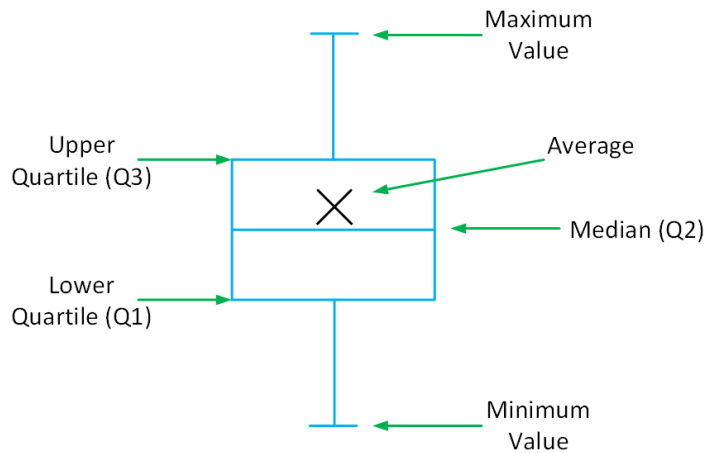


Figure B.4: Box and Whisker Plot Explanation

- **Maximum Value:** The maximum value is displayed at the top of the box using a cross-hair.
- **Minimum Value:** The minimum value is displayed at the bottom of the box using a cross-hair.
- **Upper Quartile (Q3):** Upper quartile is defined as the median for the upper half of the data set and presented with the top line in the box.
- **Lower Quartile (Q1):** Lower quartile is defined as the median for the lower half of the data set and presented with lower line in the box.
- **Median:** The median is presented with the middle-line in the box which is calculated for the whole data set.
- **Average:** The average is presented with **X** inside the box

## Curriculum Vitae

**Name:** Fadi AlMahamid

**Post-Secondary Education and Degrees:** Princess Sumaya University for Technology  
Amman, Jordan  
2007 - 2001 B.Sc.

New York Institute of Technology  
Amman, Jordan  
2002 - 2003 M.Sc.

University of Western Ontario  
London, Ontario, Canada  
2017 - 2019 M.Sc.

**Related Work Experience:** Research & Teaching Assistant  
The University of Western Ontario  
2017 - 2019

Team Lead & Senior Consultant  
Xerox  
2012 - 2016

Senior Consultant & Solution Architect  
eSolutions Information Management  
2009 - 2011

ECM Consultant  
Xerox  
2007 - 2009

IT Lecturer  
PSUT, NYIT, Colleges of Technology, and ECT  
2001 - 2007

**Honours and Awards:** Princess Sumaya Excellence Award  
2001

Graduate Student Teaching Assistant Award – Nomination  
2017 & 2019