10-12-2017 11:00 AM

Middleware Architecture for Sensing as a Service

Muhamed Alarbi, *The University of Western Ontario*

Supervisor: Prof. Hanan Lutfiyya, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

Recommended Citation

Alarbi, Muhamed, "Middleware Architecture for Sensing as a Service" (2017). *Electronic Thesis and Dissertation Repository*. 4965.
https://ir.lib.uwo.ca/etd/4965

Abstract

The Internet of Things is a concept that envisions the world as a smart space in which physical objects embedded with sensors, actuators, and network connectivity can communicate and react to their surroundings. Recent advancements in information and communication technologies make it possible to make the IoT vision a reality. However, IoT devices and consumers of data from these IoT devices can be owned by different entities which make IoT data sharing a real challenge. Sensing as a Service is a concept that is influenced by the cloud computing term “Every Thing as a Service”. Sensing as a Service enables sensor data sharing. Sensing as a Service middleware enables IoT applications to access data generated by sensing devices owned by other entities. IoT applications are charged by the Sensing as a Service middleware for the amount of sensor data they use. This thesis addresses the architectural design of a cloud-based Sensing as Service middleware. The middleware enables sensor owners to sell their sensor data through the Internet. IoT applications can collect, and analyze sensors through the middleware API. We propose multitenancy algorithms for the middleware resource management. In addition, we propose a SQL-Like language that can be used by IoT applications for sensing service discovery, and sensor stream analytics. The evaluation of the middleware implementation shows the effectiveness of the algorithms.

Keywords

Sensing as a Service, Internet of Things, Stream Analytics, Continuous query, Cloud Computing, IoT Platform.

Acknowledgments

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

I would like to express my sincere gratitude to my advisor Prof. Hanan Lutfiyya for her constant guidance and supervision, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. Without her support, this work could not have been completed.

I would also thank my parents, brothers, and sisters, albeit I understand any amount of gratitude shown to them is woefully inadequate. No words are sufficient to describe my late father contribution to my life. Thank you mother, brothers and sisters for your love, understanding and their non-stop emotional support over the period of my postgraduate study away from home. It would not have been possible for me to make this achievement if was not for your support and motivation. Therefore, I am very grateful for the impact they had on my progression.

Finally, I would like to thank my friends those of whom that I met locally at and around my studying environment and those who are scattered around the world including my home land Libya. I would like to thank all of them for being there for me when I needed their opinions, consultations and even their quality leisure time.

Dedication

To the memory of my late Father Alarbi Aboalaed Alarbi.

To my Mother Zahra Alarbi.

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

1 Introduction

In this chapter, we introduce three concepts that form the basis of the proposed work: the Internet of Things, Cloud Computing, and Sensing as a Service Model. We then introduce the problem statement, thesis focus, and thesis outline.

1.1 Internet of Things

Over the last decade, IoT has been the focus of industry as well as academia because of its great functional and financial potential. The term (IoT) was coined by Kevin Ashton in 1998. He said "The IoT has the potential to change the world, just as the Internet did. Maybe even more so. [40]". The goal of IoT is to convert the physical world into a smart space in which physical objects, called things, are equipped with computing and communication capabilities. Those things can connect with anything, anyone at any time, any space via any network or service [32]. Harald et al. [41] defines IoT as "Things have identities and virtual personalities operating in smart spaces using intelligent interfaces to connect and communicate within social, environment, and user contexts". The European Union defines IoT as "IoT allows people and things to be connected Anytime, Anyplace, with Anything and Anyone, ideally using Any network and Any service [30]". Perera et al. [34] defines IoT as "The Internet of Things (IoT) is a network of networks where, typically, a massive number of objects, things, sensors and devices are connected through communications and information infrastructure to provide value-added services". Essentially, IoT enables the vision in which there is a connectivity to almost everything.

IoT is a result of a major advancement in Information and Communication Technology. More specifically, IoT emergence is attributed to the advancement of sensor networks. Over the last 10 years, the number of deployed sensors has significantly increased because of a substantial decrease in sensor production cost [43]. Figure 1 shows that, in 2008, the number of things that were equipped with Internet connectivity surpassed the population of earth [32]. Furthermore, the European Commission notes that the number of devices that will be equipped with Internet connectivity is predicted to reach 50 to 100 billion devices by 2020 [40].

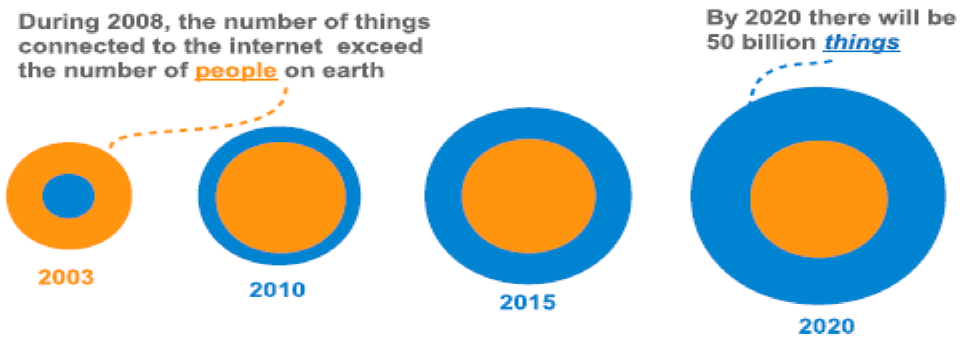


Figure 1 Growth of 'things' connected to the Internet [32].

Functionally, IoT opens doors to developing new applications in different domains such as traffic management, waste management, healthcare, smart home to name but few. The development of such applications is crucial to smart cities. Financially, the number of applications that is built on top of sensors is expected to have a positive impact on the economy. According to Cisco, IoT is predicted to create $14.4 trillion net profit value to the private sector by 2022 [3]. The amount of data generated by IoT devices is projected to be 44 zettabytes (e.g., 44 trillion gigabytes) by 2022. This immensely huge amount of data is a valuable asset that can be used to derive knowledge and detect patterns about our surroundings [3].

In summary, the Internet of Things is a concept that envisions the world as a smart space in which billions of sensors and actuators are attached to physical objects to enable them to communicate and interact with people and other things. The ability of things to communicate and react is a necessity for smart city applications that address challenges in modern cities such as traffic management, waste management, energy, education, smart home and some other challenges [37]. It's believed that coupling the IoT concept with modern computing technologies such as Cloud Computing will facilitate the development of IoT applications in many domains such as smart cities and agriculture to name a few [32, 4].

1.2 Cloud Computing

Cloud computing offers computing resources as services to its clients following the pay as you go business model. Cloud computing services include infrastructure as a service, platform as a service, and software as a service [32]. Recently, many organizations have shifted to using cloud services to reduce maintenance and operational cost. There are a number of commercial cloud platforms that provide computing resources to their clients over the Internet such as Amazon Web Services, IBM Bluemix, Microsoft Azure, Cloud Foundry, and Google Cloud Platform. The key advantage of cloud computing is that it provides its clients with elastic, and scalable resources that fits client resource needs.

1.3 Sensing as a Service Model

Sensing as a Service model [32, 37] is a new concept that is expected to be built on top of an IoT infrastructure and cloud computing services. The basic idea behind Sensing as a Service can be explained as follows: In Sensing as a Service, sensors are connected to a middleware solution, possibly hosted on a cloud platform. Data consumers are provided access to sensor data over the internet either for free or by paying a service fee to sensor owners [32]. The Sensing as a Service model is seen as a component that resides in between two IoT data sources and IoT applications in different domains such as smart cities, agriculture, manufacturing, and health care etc. Empowered by cloud computing, sensing as a service middleware solutions are expected to play a key role in delivering sensor data to IoT applications.

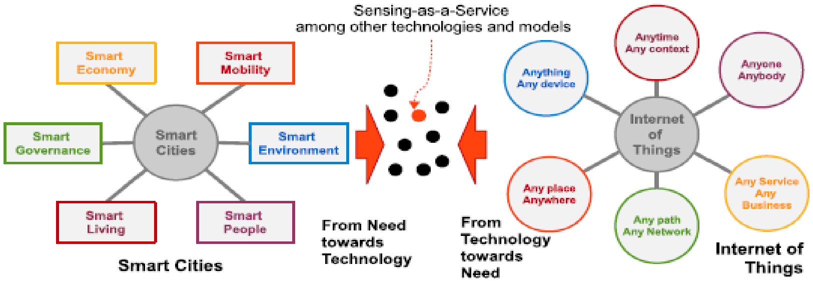


Figure 2 Sensing as service Model relation with IoT and Smart City [32].

1.4 Problem Statement

In the last several years, the number of sensors and actuators that have communication and computation capabilities has increased significantly. Those sensors generate an immensely huge amount of data that is considered meaningless unless it is used to derive knowledge [42]. The advancement in information and communication technologies made it possible to remotely access sensors over the Internet which opens the door to the development of many IoT applications that can analyze, control and react to sensory data in real-time. However, there are several challenges that must be addressed in order for IoT applications to benefit from the enormous number of deployed sensors.

To start with, connecting IoT applications to sensors is a major problem for IoT for a number reasons. First, sensors have limited power and computational resources, so they cannot directly deal with a large number of client applications. Second, dealing with sensor data entails processing a high volume of heterogeneous sensor data streams that cannot be done at the sensor device level. In some cases, an IoT application is also deployed at a resource constrained device. If the IoT application is interested in receiving processed data, stream processing should be done through an intermediary system that receives a sensor data stream, processes it and then delivers the processing result to the IoT application. Third, sensors, typically, belong to different organizations and traditionally are intentionally deployed to serve the sensor owner needs. Nevertheless, the ultimate goal is to share sensor data with other entities that are interested in the data. Having said that, there should be a way to attract sensor owners to participate in sensing operations.

There is a need for an intermediary system, known as a middleware, which decouples IoT applications from the underling physical infrastructure. This leads to a set of hardware resource requirements that should be considered in the intermediary system design. First, as the number of IoT devices continuously increases, the system must be able to connect to billions of devices. In addition, because of the heterogeneous nature of sensor data streams, the system must be generic to an extent that it is able to process any type of sensor data and deliver processing results to client applications in a near real-time manner. The system should contain an elastic, and scalable stream processing engine. Having said that, the system must have enough resources to ingest and process a tremendous amount of data and deal with spikes in client application requests.

Furthermore, there are other functional requirements that should be considered in the middleware design. First, adding a sensor to the middleware should be an easy process that even the average computer user can do, such that any sensor owner should be able to plug the sensor to the middleware. Second, sensor search is a major problem for IoT applications. As the number of sensors is in billions, there should a mechanism through which IoT applications can find sensors that fits their needs.

Recent development in cloud computing has resulted in several IoT platforms such Amazon IoT, Azure IoT, and IBM Bluemix IoT. Those platforms have the means to support IoT resource management requirements. However, those platforms lack support for sensor data sharing as only device owners can access sensor data. An IoT middleware can be built on top of those resource-rich platforms to collect, process, and enable sensor data sharing.

1.5 Thesis Focus

This work focuses on the architectural design of a Sensing as a Service IoT middleware that addresses some of the challenges discussed in the previous section. Although there has been a considerable surge of research in many aspects of IoT middleware solutions, this work relies on the emerging, resource-rich cloud based IoT platforms in the middleware design. In our design, we focus on three main aspects of middleware design. First, using a lightweight communication protocol between data sources and the middleware. Second, using resource sharing techniques when processing sensor data streams in order to reduce network traffic and cloud resource consumption. Finally, designing a programming interface that decouples IoT application from the underlying cloud and sensor infrastructure.

The primary contribution of this work is the design and implementation of Sensing as a Service IoT middleware that is built on top of a cloud platform. The middleware provides an easy, plug-and-play like approach to add sensors. In addition, the middleware provides IoT applications with an SQL-Like language that can be used through an Application Programming Interface (API) that abstracts the operations of sensing discovery, collecting and processing sensors data. Furthermore, we propose multitenancy buffer management algorithms that are used for memory management. Finally, our middleware adopts an incentive mechanism to attract sensor owners to participate in sensing operations.

1.6 Thesis Outline

The thesis is organized as follows: Chapter 2 describes the related and relevant work about this research area. Chapter 3 describes our proposed SQL-Like language and the data model used to describe sensors. Chapter 4 describes the proposed middleware architectural design and our proposed algorithms for sharing cloud resources. Chapter 5 describes the technologies used in the middleware implementation. Chapter 6 presents the results of evaluation experiments. Chapter 7 discusses conclusions and future work.

Chapter 2

2 Literature Review

In this chapter, we discuss the background of the proposed work. The foundation of the proposed work relies on Middleware Architecture, Sensor Networks, Sensing as a Service Model, Cloud-based IoT platforms, and Stream Processing Engines. Sections 2.1, 2.2 and 2.3 present the foundation of the work. Section 2.4 discusses the related work. Section 2.5 presents the gap analysis. Literature Review

2.1 Background on Middleware Architecture

A middleware can be defined as a software that resides in between the application layer and hardware layer. The middleware is designed to decouple applications from the underlying hardware infrastructure [4,37, 43]. More specifically, the middleware is usually used when there are applications that need to communicate with heterogeneous data sources. In addition to data source heterogeneity, middleware design addresses other problems such as security, interoperability, and dependability [24]. An important characteristic of middleware systems is that they are generic so that they can support applications in different domains. When using middleware, applications interact with the underlying hardware through a programming interface that abstracts the underlying infrastructure. Although this abstraction comes at an additional performance overhead as every interaction needs to go through the middleware, the reusability of its programming interface facilitates the development of new applications and makes it easier and faster.

The emergence of IoT requires a design of a middleware architecture that addresses problems beyond hardware abstraction. For example, IoT applications require a middleware that supports some other non-functional properties such as context-awareness and semantic interoperability [43] to name a few. To illustrate, for IoT applications, the context of a thing is not restricted to just its location. It has a much broader concept. For example, sensor information such as its accuracy, and capabilities are essential for IoT applications. As for semantic interoperability, it’s believed that IoT will connect billions of devices. With that being said, semantic technologies are thought of as useful tools to achieve this goal [43]. Corcho et al. [16] identify a set of challenges that can be addressed by semantic technologies. Those challenges include sensor configuration, context identification, complex sensor data querying, event detection and monitoring.

2.2 Background on Sensor Network

The emergence of IoT is attributed to the advancement in sensor networks. Over the last ten years, sensor production cost has significantly decreased. Furthermore, sensors become smaller and smarter as they are equipped with computation and communication resources. A sensor is defined as "as a device that detects or measures a physical phenomenon such as humidity, temperature, etc. [43]". A sensor network consists of several sensor nodes. A sensor node is a platform that can be connected to a number of sensors. Sensor nodes have the ability to sense, process sensor data, and communicate with each other through either wired or wireless connection [5].

2.3 Sensing as a Service

Sensing as a Service is a new concept that is expected to be built on top of IoT infrastructure and cloud computing services. The basic idea behind sensing as a service is to provide client applications access to sensors, managed and deployed by other entities, over the Internet. In Sensing as a Service, the interaction between sensor owner and sensor consumer is controlled by a pay-as-you-go model in which sensor consumers pay only for what they use. This service model benefits both sensor owners and sensor consumers [32]. From the sensor owner point of view, the sensor owner would be able to receive money in exchange for sensor data that is sold to consumers. On the other hand, sensor consumers reduce their operational cost. To illustrate, sensor consumers don't need to own physical sensor resources, yet they can use them in their applications. Thus, Sensing as a Service abstracts the underlying physical sensor network. This abstraction allows sensor consumers to focus more on their business instead of spending time and efforts dealing with sensor network infrastructure.

A Sensing as a Service middleware system is envisioned to be built on top of a cloud-based platform [4,37, 43]. The system consists of three main entities: Sensing devices, the middleware, and sensor consumers (e.g., client applications). Those entities interact as follows [37]: 1) *a client application* issues a sensing request through a programming interface to a cloud-based sensing server; 2) *the server* sends the request to sensing devices in an area of interest; 3) *the server* pushes sensing data received from sensing devices to the client application that issued the request.

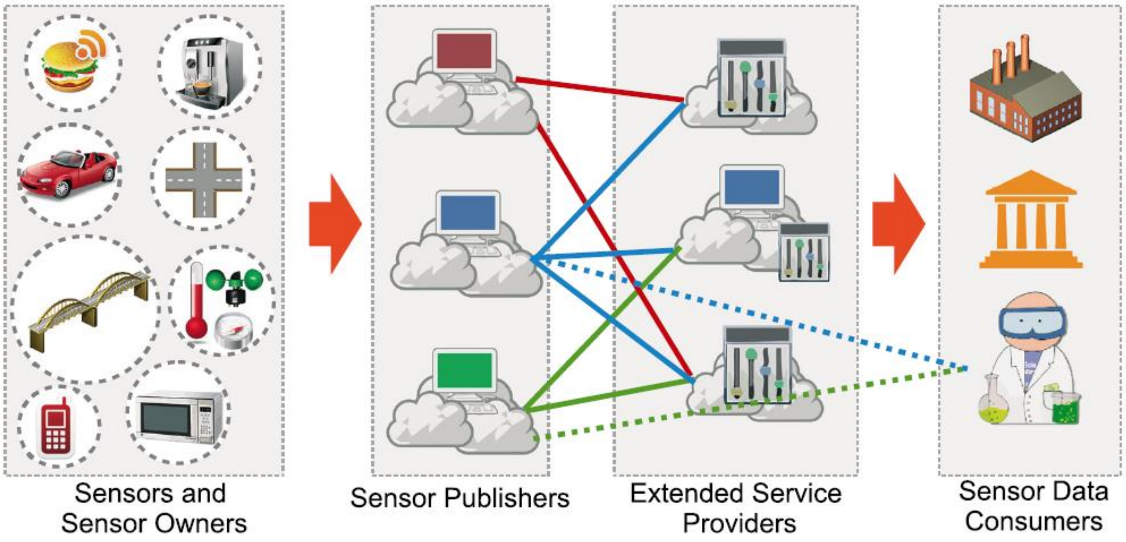


Figure 3 Sensing as a Service Model for IoT applications [32].

2.3.1 Sensing as a Service Model

It's envisioned that IoT applications would be provided access to the underlying IoT infrastructure through multiple IoT middleware solutions that adopt the Sensing as a Service concept. Charith et al. [32] proposed an IoT Sensing as a Service model that comprises four conceptual layers. The model is graphically depicted in Figure 3. First, with the sensors and sensor owners layer, sensor owners have full control over their sensors. They decide whether to share sensor data or not. If a sensor owner is willing to share sensor data, the owner specifies the terms of using the sensor. Furthermore, the owner selects the middleware through which sensor data is presented to client applications. Second, the sensor publisher layer consists of multiple cloud-based middleware solutions that manage sensors connectivity, process sensor data, and deliver it to software systems. Third, the extended service providers layer collects data from multiple sensor publishers on behalf of the consumers. It can provide domain-specific data analytics and provide sensor consumers the result. Finally, the sensor data consumers layer represents IoT applications.

2.3.2 Sensing as a Service Features

Empowered by cloud technology, sensing as a service model provides many features to client applications in different domains [4]:

1) *Decentralized data acquisition process* in which sensed data is collected from everywhere.
2) *Worldwide resource and data sharing* in which cloud and sensing resources are globally shared by different applications.
3) *Remotely accessing and analyzing real-time data* where sensed data can be accessed and analyzed in real time from anywhere.
4) *On-demand elastic resource provisioning and scaling* where users can scale the requested resources up and down based on the demand.
5) *Pay-as-you-go pricing model* in which client applications are just charged for the amount of sensor data and cloud resources they use.

2.3.3 Sensing as a Service Applications

Sensing as a Service middleware enables the development of IoT applications in a wide range of domains. In this section, we present IoT applications that can be enabled by Sensing as a Service Middleware.

1) *Remote Tracking and Monitoring:* Sensing as a Service middleware can be used to remotely monitor objects of interest. Therefore, the middleware can be used to raise alarms, and react to occurring actions in a real-time manner. Applications of remote tracking and monitoring include [4]: environmental conditions, animal behaviors, vehicles, patient-health conditions, building surveillance and security, vegetation production quality and smart-grid operations just to name a few.

2) *Real-Time Resource Management:* Sensing as a Service Middleware can be used for on-line resource control and optimization to ensure cost reduction and improve system performance. In real-time resource management, the middleware can support applications in various domains such as guided navigation, traffic control, smart parking, waste management and water/irrigation management [4, 32].

3) *Smart Troubleshooting:* Sensing as a service middleware can be used to remotely detect problems in IT systems in several domains that include: network systems, Automotive, Aviation and Aerospace, smart grids, and oil and gas pipelines [4].

2.4 Related Work

In this section, we present the related work. Section 2.4.1 describes middleware solutions, section 2.4.2 describes IoT cloud platforms, and section 2.4.3 describes Stream processing.

2.4.1 Middleware Solutions

This section discusses research efforts in designing middleware solutions.

Sense Cloud [26] is a sensing as a service middleware that is built on top of Amazon AWS cloud platform. Sense Cloud is a general-purpose middleware that addresses a set of middleware challenges such as dynamic resources provisioning, sensor virtualization, load balancing, and multitenancy mechanisms. For each sensor owner, Sensor cloud creates a virtual machine through which the sensor owners connect their sensors to the platform. Furthermore, Sense Cloud provides sensor consumers with a web application, hosted on a server instance, through which they can create virtual sensors to accesses sensors data that is placed in a cloud database. Sense Cloud can dynamically provision new server instances when the usage of the currently running instances surpasses a predefined threshold. Moreover, when Sense Cloud receives a sensing request, the load balancer is triggered to select the server instance that has the smallest outstanding request queue.

Linked Sensor middleware (LSM) [28] addresses sensor semantic interoperability. LSM uses web semantic technologies to link raw sensor data to its semantics. This process is known as Linked Stream Data. The goal of Linked Stream Data is to facilitate integrating sensor data streams into existing web technologies. LSM transforms raw sensor data into linked data represented using Resource Description Framework, known as RDF. RDF is used to process metadata; it provides interoperability between applications that exchange data on the web. RDF data store is queried using a query language called SPARQL. LSM receives sensor data through a set of wrappers that provide access to physical sensors and sensor data presented by other applications. The raw sensor data is then annotated with

Linked Stream Data Layout, which provides information such as observed property and unit of measurement. Sensor data consumers use SPARQL to query live and historical linked sensor data through an Ajax-based web application.

OpenIoT [38] is an open source, cloud-based IoT middleware that supports semantic interoperability among IoT services. Open IoT relies on W3C Semantic Sensor Network Ontology (SSN) to provide a unified metadata model for physical and virtual sensor representations. SNN ontology describes sensor accuracy, capabilities, observations, sensing method, performance, and infield deployment structure [15]. The OpenIoT ontology is an extension of SNN ontology as it doesn't restrict sensor definition to physical sensing devices since a sensor can be a device, a program, or a combination of a device and a program that can observe a phenomenon. This ontology enriches sensor description terminologies with vocabularies that facilitate IoT and cloud integration. Furthermore, OpenIoT relies on LSM [28] to transform raw sensor data into Linked Data. In addition to stationary sensors, OpenIoT supports mobile crowd sensing in which sensing operations are carried out by mobile devices.

Da Rocha et al. [19] proposed a semantic middleware for wireless sensor networks. The work addresses the Structural Health Monitoring (SHM) application domain in which semantic sensor networks can be used to enable using semantic information for monitoring and handling the environment. The middleware was developed using a low-level language called NesC, a C language extension that is used for embedded programming. The middleware uses ontologies to describe sensor information such as sensor capabilities and battery power level. Furthermore, the proposed ontologies define concepts related to other services. The middleware intelligently shares semantic information among the deployed sensors based on the semantic knowledge that controls the information sharing process. To illustrate, when the measurements of two sensors complement each other (e.g., humidity and corrosion), the sensors are allowed to share their observed values and combine their values to do reasoning. When many sensors provide the same sensing service, a few of them can be turned off to reduce energy consumption. The middleware uses a rule-based reasoning engine that employs the proposed ontologies.

Zafeiropoulos et al. [45] proposed a middleware architectural design that addresses data aggregation, management, and querying. The work focus on using semantic technologies to extract knowledge from raw sensor data. To achieve this goal, the system should employ a set of semantic technologies such as annotation frameworks, query languages and content description languages. The proposed architecture consists of three layers. First, the Data Layer in which raw sensor data is collected using polling-based or event-based mechanisms. Second, the Processing Layer which saves raw sensor data into XML files. Finally, the Semantic Layer maps sensor data stored in XML files to their semantic model. After mapping sensors data to their semantic, they system can analyze the mapped data via a semantic query language.

The Hydra project [20] proposed a domain-specific middleware that addresses applications in home automation, health-care, and agriculture domains. Hydra connects several sensor devices together to detect interesting events. Hydra is designed based on the Service Oriented Architecture and Model Driven Architecture. The middleware architecture consists of a network manager, discovery manager, event manager, storage manager, and ontology manager. The middleware uses web services to encapsulate sensors. Sensor semantic interoperability is enabled by the ontology that describes sensor devices. It is important to note that Hydra does not annotate raw sensor data with its semantic.

Lee et al. [27] proposed a hybrid middleware which consists of a server-side middleware and an in-network middleware. The server-side middleware is responsible for handling context-aware stream processing, querying and event detection. The in-network middleware is responsible for handling energy-efficient data transmission. Furthermore. In-network middleware can intelligently identify false and in-complete data values. In this work, more focus was given to in-network middleware. For this reason, the server-side middleware has limited capabilities in terms of processing sensor data.

SWASN [23] is a server-side middleware. SWASN stands for Semantic Web Architecture for Sensor Networks. SWASN employs semantic technologies to enhance sensor data processing. SWASN can connect many sensor networks by taking advantage of ontologies each network. The local ontology is used to map sensor data to a common RDF data model that can be queried using SPARQL. The SWASN architecture comprises four layers:

sensor networks, data sources, ontology, semantic web processing and, the application layer. SWASN is a domain-specific middleware with a focus on handling data for building fire emergency applications.

2.4.2 Commercial IoT platforms (State of Art)

This section describes industry solutions.

2.4.2.1 AWS IoT

AWS IoT [48] is the Amazon Web Services IoT platform. This platform was launched in October 2015. AWS IoT provides an easy, and secure way to connect IoT devices to the AWS platform and deliver device data streams to AWS Cloud services such as AWS S3, AWS Dynamo DB, and AWS Kinesis to name a few. AWS IoT can connect to billions of devices and deliver trillions of messages. AWS IoT consists of four components: Device gateway, Rule-based Engine, Device Registry, and Message Broker. The device gateway is an application that knows how to connect and send data to AWS IoT. The rule-based engine allows developers to write rules that can be used to route data streams to other AWS services such as AWS Lambda or Dynamo DB. The Registry keeps information about IoT devices and their status. AWS IoT communicates with IoT devices through a messaging broker that uses a lightweight communication protocol called MQTT. AWS IoT provides developers an easy way to connect to IoT devices, and integrate them with other services within AWS ecosystem.

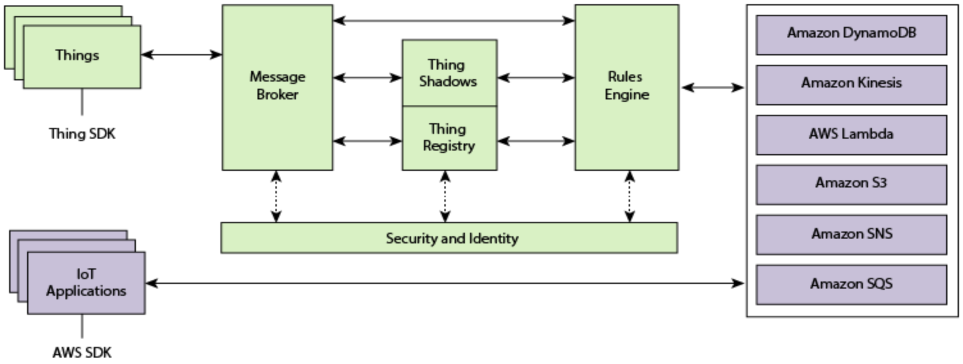


Figure 4 AWS IoT [48]

2.4.2.2 IBM IoT Foundation

IBM IoT Foundation [47,49] is a cloud-based IoT platform for managing IoT devices. IBM IoT Foundation is part of the IBM Bluemix Cloud platform. The IBM IoT Foundation provides an easy way to manage and connect IoT devices. With the IBM IoT Foundation, an IoT device can be a sensor, an actuator, or a gateway. A gateway is a device that is connected to multiple sensors, or actuators, and it's responsible for publishing sensor data to the cloud. IBM IoT foundation provides developers with a powerful web interface to add IoT devices, control access to IoT services, monitor usage, and perform device management tasks such as firmware update. Furthermore, IBM IoT foundation delivers IoT devices data to developer applications, other IBM Bluemix, storage services and IBM Bluemix Analytics services through the industry-standard MQTT protocol.

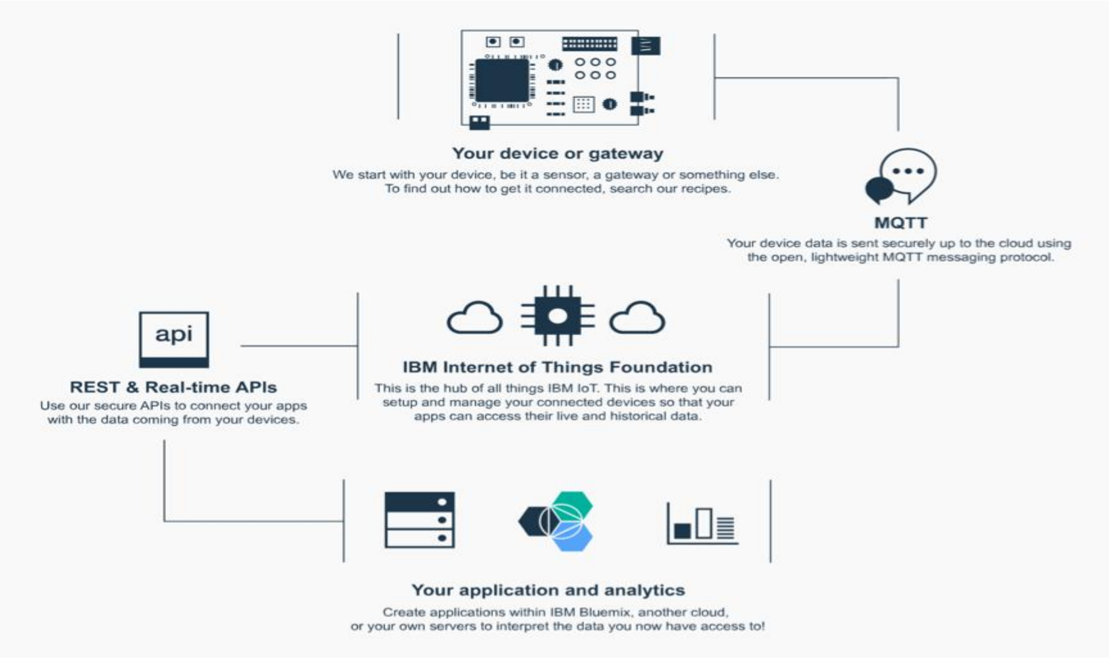

Your device or gateway
We start with your device, be it a sensor, a gateway or something else.
To find out how to get it connected, search our recipes.
MQTT
Your device data is sent securely up to the cloud using
the open, lightweight MQTT messaging protocol.
api
REST & Real-time APIs
Use our secure APIs to connect your apps
with the data coming from your devices.
IBM Internet of Things Foundation
This is the hub of all things IBM IoT. This is where you can
setup and manage your connected devices so that your
apps can access their live and historical data.
Your application and analytics
Create applications within IBM Bluemix, another cloud,
or your own servers to interpret the data you now have access to!

Figure 5 IBM IoT Foundation [47].

Recently, IBM empowered its IoT service with another IoT platform called Watson IoT [49]. Watson IoT relies on IBM IoT Foundation to manage IoT devices. Watson IoT adds cognitive capabilities to IoT applications to produce new insights and intelligence.

2.4.2.3 Azure IoT Suite

Azure IoT Suite [50] is Microsoft's cloud-based IoT platform. Azure IoT suite is an enterprise-grade solution that enables developers to create and deploy a set of extensible preconfigured solutions that address common IoT scenarios such as predictive maintenance, remote monitoring, and connected factory. Those solutions are complete, working, production-ready solutions which comprise simulated devices to produce data streams, preconfigured Azure services such as Azure IoT Hub, Stream Analytics, Machine learning, and storage services. Developers can download the source code of a preconfigured solution, customize it, and extend it to meet their specific IoT application requirements. Azure IoT Suite relies on Azure IoT Hub to manage IoT devices, and collect IoT device streams. Figure 6 shows a preconfigured IoT solution for a remote monitoring domain.

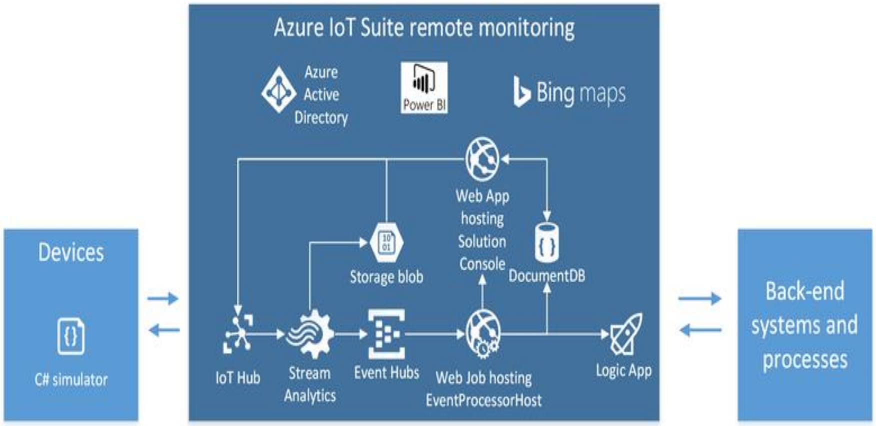

Azure IoT Suite remote monitoring
Azure Active Directory
Power BI
Bing maps
Devices
C# simulator
Web App hosting Solution Console
Storage blob
DocumentDB
IoT Hub
Stream Analytics
Event Hubs
Web Job hosting EventProcessorHost
Logic App
Back-end systems and processes

Figure 6 Azure IoT Suite Preconfigured IoT solution [50].

2.4.3 Stream Processing

Stream processing is a necessity for IoT applications. In this section, we present the foundation of stream processing and discuss research efforts in stream processing.

2.4.3.1 Stream Definition

Abstractly, a stream, S, is a set of relational tuples, possibly infinite. The tuples share the same structure. Each tuple is characterized by a set of attribute names $\{A_0,\ldots, A_{n-1}\}$ [31]. The tuples may be generated by one or more data sources. A timestamp is associated with each tuple. This timestamp can be regarded as a supplementary tuple attribute that is denoted by A_t. Another way to define a stream is as a big bag of pair elements <r,t> where r is a tuple characterized by a set of attribute names and t is a timestamp associated with the tuple [8]. Regardless of the formalism, a tuple has a unique timestamp, but there can be multiple tuples associated with a timestamp. The rest of this section assumes the use of a supplementary tuple attribute.

At a given timestamp t_i, the current stream content is defined as follows [31]:

$$S(t_i) = \{s \in S: s.A_t \leq t_i\}$$

Streams satisfy the following properties:

- Existence: $\forall\ s \in S,\ s.A_t \neq \text{NULL}$
- Monotonicity: If $t_i < t_j$ then $S(t_i) \subseteq S(t_j)$.

2.4.3.2 Window Semantics

A stream may be very large or possibly infinite. This makes it difficult to execute queries especially those with aggregation operators (e.g., average, maximum) and stateful operators (e.g., intersection and join which use multiple streams). These operators cannot generate output before the entire input is read. However, with streams, it is not always possible to know when or if a stream ends. To address this problem, queries can specify a window of time that represents a subset of a stream [31, 39,1]. A window is a mechanism to specify, dynamically, moving boundaries over stream tuples in order to extract a finite, yet always changing, set of tuples to be used as an input for blocking and stateful operators such as aggregations, join, and merge operators [31].

Window attributes are used to specify the upper bound, lower bound, extent, and mode of adjustment. These are described below.

- *Upper bound* is a value that specifies the most recent tuple that should be included in the window subset.
- *Lower bound* is a value that specifies the oldest tuple that that should be included in the window subset.
- *Extent* is a value that specifies the size of the window that could be a number of elements or a time interval.

- *Mode of adjustment* specifies the way in which the window changes as time advances.

Representative examples of windows are briefly described in the rest of this section.

A *time-based window* is defined by a time interval. A time-based window can be represented by a start time (t_s) and an end time (t_e). These represent the lower and upper bounds of the window as well as the extent. For a stream, S, a query that uses time-based windows would apply query operators to the set represented by the following [8]:

$$S(t_e)-S(t_s)$$

Possible adjustment modes include the following [31]:

- *Landmark:* One of the window boundaries is kept equal to a specific time, while the other window boundary incrementally changes as time advances. This is referred to as a *landmark window*. A *lower-bounded landmark window* is where the lower bound stays fixed at a specific time while the upper bound advances with time. An *upper-bounded landmark window* is where the upper bound is set to a fixed value while the lower bound advances with time.
- *Sliding Window:* Both the start and end times may change. Boundaries proceed based on a predefined progression step β and a fixed temporal size ω. β is always set to be less than ω. As a result, an overlap between successive windows is always observed (see Figure 7 c). An overlap is prevented by the condition $\beta \geq \omega$ (Figure 7 d).

A *count-based window* is defined by a timestamp, t, and N which represents the N most recent tuples with a timestamp less than or equal to t. This is more formally defined by the following [31]:

$$\{s \in S(t) : \exists\, t_1 \in T\, (t_1 \leq t \;\wedge | \{ s \in S(t) : t_1 \leq s.A_t \;\leq t \} \; | \leq N) \;\wedge$$
$$\forall t_2 \in T\, (\, t_2 < t_1 \wedge | \;\{ s \in S(t) : t_2 \leq s.A_t \leq t \}| > N)\}$$

The upper bound is defined by the timestamp, t and N is the extent. One possible adjustment mode varies t.

With *partitioned windows,* the stream tuples are first partitioned into different sub streams based on the values of specified grouping attributes. At each time, t, the N most recent tuples are taken from the sub streams.

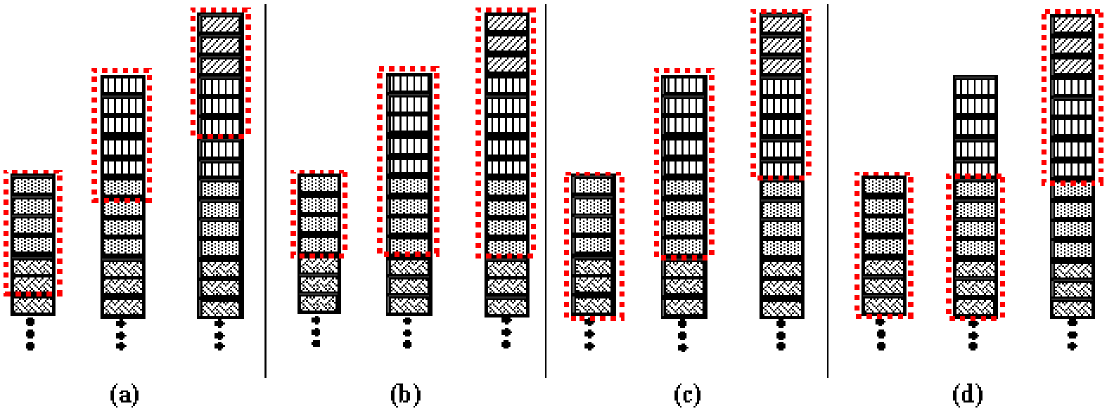


Figure 7 Window Types [31]

2.4.3.3 Stream Processing Engines

Stream processing Engine (SPE) is a term this is used to refer to an application that is designed to process a massive amount of streaming data on the fly in a near real-time manner [39]. An SPE is an intermediary between data sources and client applications. SPEs execute client application queries over live, possibly unbounded, data streams presented by data sources. Unlike database systems that execute queries over stored data, SPEs analyze stream tuples as they move through the system due to the high volume of input messages that discourages the use of persistent storage [22]. Abstractly, SPEs are similar to DBMSs in the sense that both apply relational algebra operations (e.g., select, project, aggregate, filter, etc.) over a dataset. However, their implementation of these operations is substantially different [8]. The operations need to consider the unbounded nature of data streams [31,8]. Figure 8 shows the general concept of a stream processing engine.

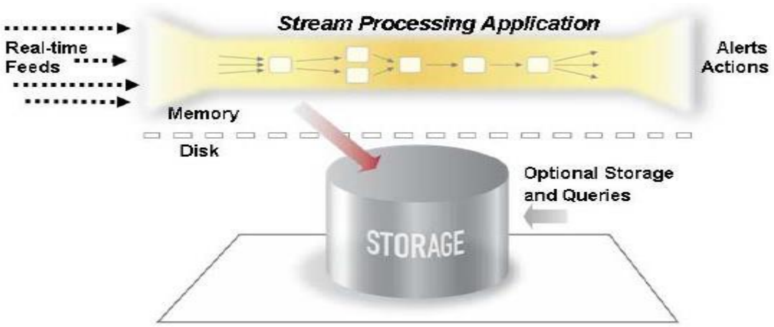


Figure 8: General Concept of Stream Processing Engine [39]

2.4.3.4 Database Limitation for Real-Time Stream Processing

SPEs are designed to overcome DBMSs limitations in supporting real-time stream-based applications. DBMSs follow the store then process programming paradigm. A traditional database model is inappropriate for real-time stream-based applications [1,25, 39]. This section briefly describes the limitation of DBMSs in supporting real-time stream-based applications.

First, in traditional database systems, a query processor reads data from a disk. In a traditional database system store, input tuples are stored and indexed before they are made available for query activity. Disk storage introduces latency which makes it difficult for applications to receive the data in real-time or even in near real-time.

Second, the query processor in SPEs needs to consider that the data arrival rate can be extremely high and thus a query processing for stream applications should employ scheduling and load shading techniques to control CPU and memory usage by active queries [12].

Third, typically queries from stream-based applications are long running queries [8,31]. Blocking operators such as aggregation operators (e.g., avg, max) need special treatment by running them over a portion of the stream which is referred to as a window. Blocking operators return their results at the end of each window i.e., essentially all the input has to be read before output can be produced. Similarly, stateful operators like join and group also should work on stream subsets. A system that manages streaming data must consider reducing operator state accumulated by continuous queries.

2.4.3.5 Stream Processing Engine Requirements

A stream processing engine must provide the following features [39]:

1. **Keep the Data Moving**: A query may consist of one or more operators. To provide low latency, an SPE must be able to process stream messages without using storage operations throughout the processing path of query operators that may include blocking and stateful operators. However, an SPE should have a special treatment for blocking and stateful operators. In addition, to reducing latency, SPEs should adopt an active processing model (e.g., non-polling) in which query output is constantly delivered to client applications rather than waiting for client applications to make requests to poll results because polling increases the system overhead and the processing delay.
2. **Query using SQL on Streams (StreamSQL):** There should be a mechanism through which client applications can express operations to be executed over data streams. An SPE should support a high-level query language such as SQL that supports stream operations [8, 18, 7, 10]. The language operators should be extensible to allow developers to define new streaming functionalities [1,10].
3. **Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data):** Unlike traditional database systems, stream data is queried before it's presented. For this reason, an SPE must provide a mechanism to provide flexibility to deal with stream imperfection situations that include delayed, missing, and out-of-order data [2].
4. **Integrate Stored and Streaming Data**: As some streaming applications demand access to stored data in order to compare past and present stream data, an SPE should be able to execute queries over stream data, stored data, or a combination of the both. The system should provide a uniform query language that can be used when querying either data source.
5. **Guarantee Data Safety and Availability:** An SPE must always be up and running despite the workload spikes that might happen at run-time. In case of system failure, a backup hardware device should take over tasks assigned to failed devices in order to keep going.
6. **Partition and Scale Applications Automatically:** An SPE should be able to support parallel query execution in which query processing is distributed among a set of

machines in a cluster. In addition, the system should load balance computation across its nodes.

7. **Process and Respond Instantaneously:** In order to provide low latency when processing high-volume input streams, an SPE should be able to process hundreds of thousands of messages per second. To do so, the SPE should have a highly-optimized, minimal-overhead execution engine.

2.4.3.6 Single-Site Stream Processing Engines

Aurora [1] is a stream processing engine (SPE) that was a result of collaborative research efforts of students and professors at MIT, Brandies, and Brown University. Aurora's main task is to perform data analytics on inbound stream messages in a way specified by an application administrator. Aurora follows a dataflow-style paradigm in which operations on a stream are represented by boxes, and arrows indicate the order in which operations are applied. Basically, an Aurora query is an acyclic directed graph where the nodes represent operators, and the edges represent data flow. Aurora provides a graphical user interface to specify the query. In addition to online stream processing, Aurora supports historical analytics in which Aurora stores stream tuples for a certain amount of time and uses this storage to answer ad-hoc queries. Aurora translates the query graph into a data structure that is saved into a database. At run-time, Aurora loads the query graph from the database to be used to direct the input stream to the relevant operators. In addition to on the fly query execution, Aurora addresses stream processing challenges such as query optimization [1], load shedding (i.e., load reduction) [1], and scheduling [12].

STREAM [7] is a single site stream processing engine developed at Stanford. STREAM stands for STanford StREam Data Manager. Stream was designed to run queries over a combination of data streams and static relations (e.g., tables). STREAM was developed with the intention to minimize memory allocation. Furthermore, STREAM supports query plan modification runtime [6, 7]. In addition, due to resource limitations, STREAM provides a way to compute approximate query results. Furthermore, STREAM clients use a SQL-Like query language to define continuous queries over incoming data stream and static tables. The STREAM query language is known as CQL which stands for Continuous Query Language [8]. As CQL uses SQL-99 standards, it supports relational operators such as select, project, filter, duplicate elimination and aggregation functions. In addition, CQL provides window operators that partition streams into small chunks.

COUGAR [10] is a stream processing engine developed at Cornell. COUGAR is a prototype sensor database system that performs long running queries over unbounded streams and static relations. COUGAR was developed as an extension for an object-relational database called PREDATOR. COUGAR defines a sensor database system as a mix of stored relations that consists of sensor deployment information and sensor streams which are represented as a time series. A continuous query over sensor streams runs for a user defined time interval and defines a persistent view that is updated with the query results during the query time interval. The COUGAR sensor database system model consists of the Remote Sensor Model, Stream Model, Query Language, and Query processing. Although COUGAR supports long running queries, it does not support window operations over stream data.

Gigascope [18] is a Data Stream Management System that runs queries over a continuous stream of data emitted by network card interfaces (NICs). Gigascope serves as an intermediary between data sources (e.g., NICs) and network monitoring applications. It receives queries from user applications and executes the queries over the incoming data streams from network card interfaces. Gigascope does not run queries over stored data, so all Gigascope inputs and output are streams. Gigascope can analyze high-speed streams presented by communication networks without using expensive processors. Furthermore, Gigascope provides its clients with a query interface that uses a SQL-Like language called GSQL.

2.4.3.7 Distributed Stream Processing

Stream processing in a distributed environment offers several advantages [46]. To begin with, it enables stream processing to be scaled over many nodes. It increases the availability of Stream Processing Engines as the processing nodes monitor each other and take actions to keep the system running in case of node failure. Moreover, it makes it easier for nodes to cope with sharp increases in load by having nodes cooperate in sharing the load until each node has the required resources to handle its assigned tasks. In this section, we first present concepts related to stream processing in a distributed environment. We then present a number of multi-site stream processing engines.

2.4.3.7.1 Parallel Query Execution

The parallel query execution paradigm states that queries are distributed among multiple nodes that collaboratively interact to produce output. Parallel query execution can be classified into two categories [22]. First, inter-query parallelism in which different queries are assigned to different nodes. Second, inter-operator parallelism in which query operators are distributed across different nodes.

Despite which parallelism technique an SPE follows, query parallelization implementation should satisfy two transparency conditions: syntactic transparency and semantic transparency [22]. Syntactic transparency states that users should be unaware of the query parallelization process. Semantic transparency states that using a given input data stream, the result of a parallel query must be the same as its centralized counterpart.

2.4.3.7.2 Elastic vs Static SPE Configuration

Elastic and static SPE configurations are terms used to describe SPE resource management mechanisms in distributed environments [22]. In static configurations, an SPE employs a fixed number of nodes for handling query processing. In contrast, in an Elastic SPE configuration, the number of running nodes dynamically changes at run-time in response to the current workload. Although static SPEs have the means to support parallel query execution, their resource management poses two major disadvantages [22]: resource under-provisioning, and resource over-provisioning. Under-provisioning occurs when the number of provisioned nodes is not sufficient to cope with the workload. Over-provisioning occurs when the size of workload can be handled with a fewer number of nodes which results in a waste of resources. In contrast, the elastic configuration makes an SPE able to adjust its amount of allocated resources to the level that serves the current workload.

2.4.3.7.3 Multi-Site Stream Processing Engines

Medusa [46] is a distributed version of Aurora [1]. Medusa extends Aurora’s functionality by distributing queries across multiple single site stream processing engines (e.g., nodes) that cooperate at runtime to process stream tuples and deliver processing results to client applications [46]. Medusa nodes are called participants who might belong to a single organization or be part of a coupled federation of nodes in which nodes are controlled by independent owners. Medusa adopts a financial incentive model in which nodes receive payments for rules they play in query processing and load sharing [9]. The Medusa incentive model is called the bounded-price mechanism in which inter-node load sharing agreement is negotiated before participating in the runtime environment. At run-time, a node can only move the load to another node if and only if there is a contract between them. Moreover, Medusa brings another significant change to the way Aurora works. Instead of using Aurora as a stand-alone system, Medusa allows client applications to communicate with the system through an API that wraps the system functions. The API facilitates the integration of Aurora with client applications.

Borealis [2] follows the work-flow paradigm that Aurora [1] implements, so a Borealis query is a directed graph of boxes, arcs, and arrows that collectively describes the order of stream processing steps. Borealis adopts the Medusa [46] query distribution mechanism. Moreover, Borealis extends the Aurora query model by allowing client applications to change or update operators at run-time. Furthermore, Borealis proposes a data model that allows dynamic query revision. Dynamic query revision allows a stream processing engine to correct mistakes in previously generated output messages. Wrong output values are generated as a result of incorrect input values generated by data sources.

Stream Cloud [22] is a distributed stream processing engine that processes continuous queries by distributing those queries among multiple nodes. Stream Cloud adopts a parallelization mechanism in which queries are split into small chunks, called subqueries, which are assigned to a set of independent Stream Cloud instances, e.g., nodes. Stream Cloud query distribution approach aims to minimize distribution overhead. Furthermore, Stream Cloud is an elastic and scalable SPE that is capable of handling large volumes of stream data. Stream Cloud was built on top of a Borealis [2], so Stream Cloud inherits Borealis query model in which a client application query is represented as an acyclic directed graph where nodes represent operators and edges represent data workflow. However, Stream cloud approaches for parallel query execution, scalability, and elasticity are substantially different from Borealis. This is due to the different query parallelization techniques used in the two systems. While Borealis distributes query operators among multiple nodes, Stream Cloud supports three different query parallelization techniques that are categorized based on the granularity of the parallelization units which are Query-cloud strategy, Operator-cloud strategy, and Operator-set-cloud strategy [22].

TelegraphCQ [14] is a stream processing engine developed at UC Berkeley. TelegraphCQ directs stream tuples through a set of query operators that handle tuples in the same fashion used in traditional databases. Furthermore, TelegraphCQ has a special set of routing modules which are used to route tuples among query operators. TelegraphCQ was built to support shared and adaptive continuous query processing over, possibly unbounded, data streams. Adaptive query processing states that an SPE should have the ability to adjust its processing dynamically in response to unexpected changes in data availability or as a response to changes in client needs [14]. The Shared query processing targets commonality among client queries. Instead of processing each query separately, shared processing states that stream tuples should be processed simultaneously by all active client queries as the tuples pass through the system. In addition, TelegraphCQ addresses resource management, scheduling, and distributed parallel query execution. TelegraphCQ inherits most of its functions from predecessor projects: Telegraph [36], CACQ [29] and PSoup [13]. Telegraph provided adaptive query processing, yet it did not target commonality among active queries. The latter two projects addressed shared query processing. However, their implementation showed significant limitations [14]. For instance, the limit of data that they can process depends on the memory size. Moreover, they did not provide solutions to resource management and scheduling. For those reasons, TelegraphCQ was developed to address challenges that were considered to be drawbacks of its predecessors Telegraph [36], CACQ [29] and PSoup [13]. Furthermore, TelegraphCQ supports distributed parallel query processing through an extension project called FLuX [35]. TelegraphCQ uses FLuX as a routing module that routes tuples across multiple nodes in a cluster. The key feature of FLuX that it’s able to redistribute query operators alongside their internal state with minimal impact on query processing.

2.5 Gap Analysis

Over the last decade, researchers spent a great deal of time working on building IoT solutions to support IoT applications in different domains. Those solutions focus on the architectural design of a middleware that decouples the underlying IoT infrastructure from IoT applications. The proposed work is devoted to address aspects of middleware design that increase interoperability such as sensor ontology, sensor data semantic, sensor query language, sensor virtualization, data acquisition wrappers, and programming interfaces. However, the proposed middleware solutions have some drawbacks. First, the proposed middlewares cannot support a large number of IoT devices as they don't have the means to connect to billions of devices. Although some of the proposed solutions use cloud services, those solutions mainly depend on cloud services for hosting and storage services which make connecting to IoT devices a system bottleneck. Second, some of those middlewares are domain-specific middlewares that only support applications in specific domains. Third, to the best of our knowledge, none of the proposed systems have multitenancy mechanisms that enable multiple client applications to share resources for sensor connections and sensor data streams. For example, when different client applications request a middleware to perform aggregations over sensor stream. In most cases, processing sensor data streams dictates storing stream elements in a buffer for a period of time, processing stream elements and delivering the results to client applications. With that being said, the lack of buffer management poses significant problem to the middleware resource consumption as stream buffers are stored in the RAM, which is a limited resource. Thus, there is need for a multitenancy mechanism to manage stream buffers. Finally, the proposed middleware solutions present sensor data streams to client applications at the same frequency that the sensor uses when it sends its data to the middleware. However, client applications might be interested in receiving sensor data at different frequencies, and they should not pay for a frequency they don't need. For Example, a sensor sends its data every second, yet a client application wants to receive sensor data every 10 seconds, the sensor should have multiple pricing policies for different frequencies. To the best of our knowledge, none of the proposed middleware solutions considered the idea pricing policies.

Commercial cloud-based IoT solutions are resource-rich platforms that enable the connection of billions of devices that can send trillions of messages. The commercial platforms provide powerful analytics and stream engine services. Basically, commercial cloud-based IoT platforms have the means to build IoT middleware solutions. However, they are more focused on providing tools and services that enable developers to easily integrate their IoT devices into the cloud platforms and facilitate the process of routing device data streams to different cloud services within the same platform or at a different one. Essentially, sensors are tied to a cloud platform account. Only developers who have access to that account would be able to access the sensor data. This contradicts the IoT data and resource sharing concepts. Moreover, an IoT middleware should also abstract the underlying cloud structure, so software developers can focus on building their application. To summarize, commercial cloud-based IoT platforms have the required infrastructure to build IoT middleware solutions. However, they are more focused on providing the services that enable developers to create their own IoT applications.

Finally, Stream Processing Engines (SPEs) can analyze sensor streams on the fly, and present analysis results in a near real-time manner. Moreover, SPEs support continuous query semantic. However, SPEs are focused on providing algorithms for stream analysis, reducing memory usage, and supporting adaptive query processing. SPEs don’t consider the idea of sharing sensor data. SPEs replicate stream data for queries issued by different client applications. SPEs don’t consider that the relation between a stream and its consumer applications is a one-to-many relationship. For this reason, SPEs cannot be considered as a middleware solution that provides sensing as a service concept.

To summarize, middleware solutions, stream processing engines, and cloud-based IoT platforms have some drawbacks that imply that they cannot be used as an IoT solution that enables sharing sensor data. There is an apparent need for an IoT solution that supports sensor data sharing. This solution can be built on top of a combination of the aforementioned technologies.

Chapter 3

3 Sensing as a Service Query Language

Sensing as a Service Query Language is a SQL-like declarative language used to execute consumer queries over sensor metadata, and sensor data streams. As noted in the previous chapter there is a considerable number of published research papers in the area of stream processing. In addition to focusing on stream management techniques, researchers also focused on building query languages that have the capability of executing stream operations on windows of data. However, these languages are mostly focused on stream processing. Moreover, some of the middleware solutions proposed search functionalities that facilitate sensing search [28, 38]. However, those search functionalities are limited and use query language called SPARQL which is not user-friendly for non-technical users [33]. Furthermore, to the best of our knowledge, there is no query language that supports the Sensing as a Service concept. In this section, we propose a SQL-like declarative query language. The proposed query language allows client applications to search for sensors using a metadata model built on top of the SSN ontology [15]. Moreover, the language supports Sensing as a Service model by allowing client applications to specify query parameters that indicate the frequency of the requested data and the pricing policy of the Pay-As-You-Go Incentive Model.

3.1 Stream Query Language Requirements

Querying sensor data streams is quite different from querying data stored in static relations. This is due to the volatile nature of streaming data. Unlike static relations, data streams cannot be stored in a disk. This is due to the fact that streams may be very large or possibly infinite which discourages stream storage. This makes it difficult to execute queries especially those with aggregation operators (e.g., average, maximum) and stateful operators (e.g., intersection and join which use multiple streams) since these operators cannot generate output before the entire input is read. Furthermore, client application queries might be executed over generated data from a future time. Subsequently, queries must be applied as data streams flow through the stream query processing engine. With that being said, operations like stream filtering and aggregations should be treated in a special way. Therefore, special terms should be added to the stream query language to capture concepts like query session, sensing discovery, windowing, and pricing policies.

- **Query Session** is a term used to refer to the time frame which a consumer wants to execute their query over the stream data generated during the time frame.
- **Windowing** is the process of splitting the data stream into smaller subsets of data in order to run a query over those subsets separately. Windowing is a technique used to unblock stateful (i.e., Join, Merge) and blocking operators (i.e., Aggregation functions).
- **Pricing policies** form the commercial model regarding what consumers pay for their sensor data usage.
- **Sensing Discovery** is the process that enables software systems to locate sensors that fit their needs.

3.2 Query Language

In this section, we present a query language for a Sensing as a Service middleware. Using this language, the middleware clients can perform sensing discovery tasks. In addition, clients can use the proposed SQL-Like query language to perform stream analytics operations.

3.2.1 Information Model

Sensing discovery is crucial for IoT applications. As the number of IoT devices is in the billions, it will not be easy for IoT applications to identify sensors that fit their needs out of billions deployed sensors. Sensing discovery provides a searching mechanism which narrows down the range of sensors that fit client needs. However, the fact that sensors are owned by different entities poses a problem for integrating sensors with IoT applications because sensor owners might describe their sensors in different ways. For this reason, there is a need for an information model that provides a unified method for describing the deployed sensors. In our design, the proposed middleware employs sensor metadata model that describes sensor capabilities and deployment features. Our sensor metadata model relies on a semantic sensor network ontology (SSN) in describing sensors [15]. The SSN ontology includes the most common context properties, such as accuracy, precision, drift, sensitivity, selectivity, measurement range, detection limit, response time, frequency and latency. In addition to sensor context properties, a sensor location description is also important for the Sensing Discovery process. We use a hierarchical model to capture sensor location description. At the top level, we have countries, then cities. In a city, sensors might be deployed outside or inside a property. In both cases, we store the sensor longitude and latitude. For sensors located inside a property, we store more location information such as property type (e.g., apartment building, flat, house). Each property type consists of a set of components: room, hallway, bathroom, kitchen, entrance, parking lot, backyard, etc. Within a property component, a sensor might be attached to an object which might be a wall, a door, or a window. When a sensor owner adds a sensor, the owner specifies detailed location information which includes the property, the property component, to which object the sensor is attached to within the property component. This information model provides a precise sensor location description information. Outdoor sensors also can have more location information. For example, a sensor in a parking lot might have a location description that indicates where in the parking lot the sensor is located (e.g., in which spot in which floor). Moreover, a sensor deployed in a highway might have a description that tells which lane this sensor monitors and to which object the sensor is attached. It’s important to note that currently we limit outdoor sensor description to GPS location. A full outdoor location modeling is beyond this work. The proposed information model is graphically depicted in Figure 9.

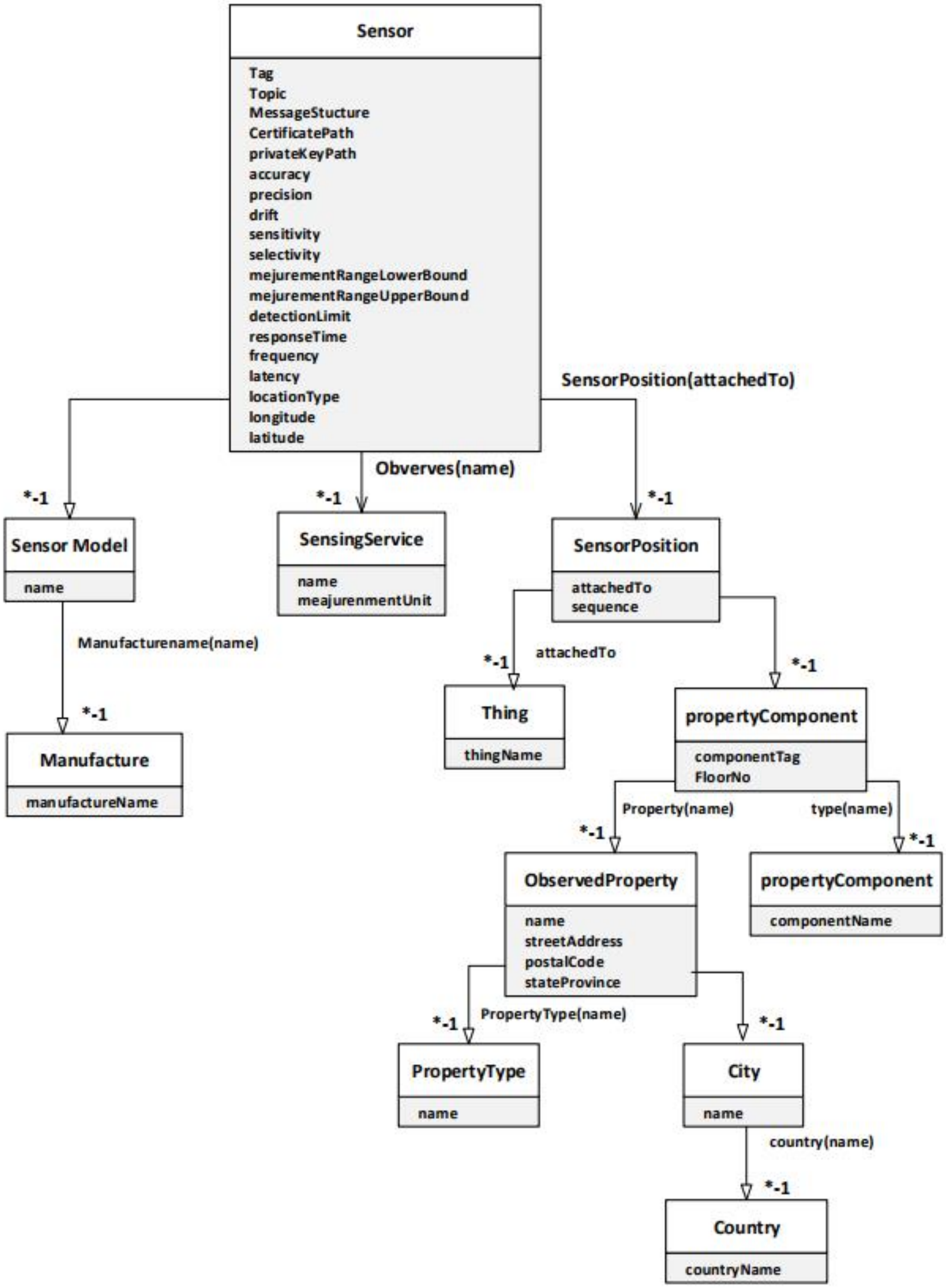


Figure 9 Information Model.

3.2.2 Sensing Discovery Query

A Sensing Discovery query is a query that client applications submit to the middleware to construct sensors deployment knowledge. Sensing Discovery queries are executed against the sensor information model. The goal of Sensing Discovery Queries is to help consumers discover sensors that meet their needs. It's important to note that in the sensor discovery process, the middleware does not make a decision about which sensor satisfies the consumer needs the best. Rather it is the consumer who makes this decision based on the middleware answers for sensing discovery queries. Essentially, the middleware narrows down the range of selection.

3.2.2.1 General Knowledge Sensing Discovery

General Knowledge Sensing Discovery provides consumers with general knowledge about sensing services in a given country. The middleware's response for these queries is to provide general deployment information. For example, in which cities temperature sensors are deployed, in which buildings in a given city there are air quality sensors, or what are the available sensing services in a given city. Using EBNF, we define a General Knowledge Sensing Discovery Query as follows:

```
<SelectStatement> ::=
    SELECT <SelectOptions>
     FROM <CountryIdentifier>
     [IN <CityIdentifier>]
     [PROVIDES <sensingService>]
     [MANUFACTUREDBY <VendorName>] ;

<SelectOptions> ::= ‘City’ | ‘SensingService’ | ‘Building’
<CountryIdentifier> ::= String
<CityIdentifire> ::= String
<sensingService>::=<sensorType>{‘,’<sensorType>}*
<sensorType> ::= String
<VendorName> ::=String
```

A General Knowledge Sensing Discovery Query has the following format:

SELECT < *City | Sensing Service |Building*>
FROM <*CountryIdentifier*>
[**IN** *CityIdentifer*]
[**PROVIDES** *sensingService*]
[**MANUFACTUREDBY** vendor];

Select: The select expression takes three possible keyword values: *City*, *SensingService*, or *Building*. The keyword *City* indicates that the consumer wants to receive a list of cities (e.g. <CityId, CityName>). The keyword *sensingType* indicates that the consumer wants to receive a list of Sensing Types (e.g, temperature, humidity, traffic) available in a country specified in the Country Identifier part of the query. The keyword *Building* indicates that the consumer wants to know the addresses of buildings that provide sensing services. *CountryIdentifier***:** This part of the query indicates which country the consumer wants to search to find general sensor deployment information.

IN *CityIdentifer*: This is an optional part of the query that can be used alongside the *SensingService*, and *Building* keywords. This means that the consumer wants a list of the available sensing services, or building addresses in a given city as identified by *CityIdentifier*.

PROVIDES *sensingService*: This is an optional part of the query that can be used alongside the *City* and *Building* keywords. This part tells the middleware that the consumer wants to know in which cities or buildings a giving sensing service is available.

MANUFACTURED BY vendor: This is an optional part of the query which indicates that the consumer wants to find sensing services or cities in which the deployed sensors are manufactured by a given vendor.

We will now provide several example queries.

Example 1: This query is used to discover names of cities in which the temperature sensing service is available.

```
SELECT City
FROM Canada
PROVIDES Temperature;
```

Example 2: This query is used to discover names of cities in which *Humidity* sensors manufactured by OMEGA are deployed.

```
SELECT City
FROM Canada
PROVIDES Humidity
MANUFACTUREDBY OMEGA;
```

Example 3: This query is used to discover names of sensing services associated with sensors that are manufactured by OMEGA.

```
SELECT SensingService
FROM Canada
MANUFACTUREDBY OMEGA;
```

Example 4: This query is used to discover cities in which there are deployed sensors manufactured by OMEGA.

```
SELECT City
FROM Canada
MANUFACTUREDBY OMEGA;
```

Example 5: This query is used to discover the addresses of buildings that provide humidity sensing services in London.

```
SELECT Building
FROM Canada
IN London
PROVIDES Humidity;
```

This query states that a consumer is looking for address of building that provides humidity sensing services in London.

3.2.2.2 Discovery at the Sensor Level

Sensing discovery at the sensor level provides consumers with detailed information about the deployed sensors based on context and location features. The middleware responds to queries with a list of detailed sensor information. For each sensor, the response consists of sensor identifier, sensor data scheme (e.g., sensor data attribute names and data types), Sensing Service (e.g., temperature, humidity), sensor context features, manufacturer name, pricing policies, and location description.

EBNF notation:

```
<SelectStatement> ::=
      SELECT <SelectOptions>
       FROM <CountryIdentifier>
       [ WHERE <search_condition> ]
       [IN <CityIdentifier>]
       [AT <BuildingAddress>]
       [WITHIN <DistanceAttributes>]
       [MANUFACTUREDBY <VendorName>] ;


<SelectOptions> ::= <sensorType>{‘,’sensorType}*
<CountryIdentifier> ::= String
<search_condition>:= <logicalExpression> [{<logicalOp><logicalExpression>}*]
<logicalOp> ::= (And | Or)
<LogicalExpression> ::= <Expression> <OP> <Expression>
<OP> ::=(> | >= | < | <= | = | !=)
<Expression> ::= <term>[(+|-) <term>]*
<term> ::= <factor> [ ( * | / ) <factor>]*
<factor> :=’(‘< Expression>’)’| attributeName | number
<CityIdentifire> ::= String
<BuildingAddress> ::= String
<DistanceAttributes> ::= number, number, number
```

<sensorType> ::= String

<VendorName> ::=String

A discovery query has the following format:

SELECT <SelectOptions>
FROM <CountryIdentifier>
WHERE <search_condition >
IN <CityIdentifier>
[**AT** Building Address]
[WITHIN distance, longitude, latitude**]**
[MANUFACTUREDBY vendor];

SELECT: The SELECT expression indicates the sensing service the consumer is looking for which might be one or many sensing services such as temperature, humidity, air quality …etc.

FROM: This part of the query specifies the country in which the consumer wants to discover sensors.

WHERE: This part of the query specifies sensor context features.

IN: This part of the query indicates which city the consumer wants to search for sensors.

AT *Building Address*: This an optional part of the query that is used when the consumer is looking for sensors information at a specific building.

WITHIN *:* This an optional part of the query that is used when the consumer is interested in finding sensors that are located within a specific distance from a given longitude and latitude.

MANUFACTUREDBY: This indicates that the consumer wants to find sensing services in which the deployed sensors are manufactured by a given vendor.

We now present an example of a query.

Example 6: This query returns full description of temperature sensors which are deployed at building 740 Proudfoot Lane in London and satisfy the condition accuracy= 90.

```
SELECT Temperature
FROM Canada
Where accuracy= 90
IN London
At 740 Proudfoot Lane;
```

3.2.3 Stream Analytics Query

Stream Analytics Query is the query used to analyze sensor data streams on the fly. The middleware provides a SQL-like query language that provides consumers the ability to query a sensor as they would query a table in a relational database. Using this language, consumers can specify sensor data attributes they want to receive in case the sensor provides a set of attributes. For example, a sensor that measures temperature and humidity provides two attributes that are named temp and hum. Moreover, consumer can use aggregation functions such as average, minimum, maximum, sum and count. Furthermore, consumers have the ability to filter the data streams provided by sensors.

It’s important to note that in our design we adopt the formal definitions presented in sections 2.4.3.1 and 2.4.3.2 for sensor stream, stream tuple, and window semantics.

We formally define the Stream Analytic Query using the EBNF notation as follows:

```
<SelectStatement> ::=
    SELECT <SelectOptions>
     FROM <Sensor Identifier>
    WHEN <SessionFilter>
    [WINDOW (“number | “unbounded”)]
    USING number
    FOR (number | “unbounded”);
```

<SelectOptions> ::=
(‘*’| attributeName {‘,’attributeName}* | < aggregationFunction > ‘(’attributeName ‘)’ {‘,’<aggregationFunction> ‘(’ attributeName ‘)’}*)

<aggregationFunction> ::= ‘avg’ | ‘sum’ | ‘count’ | ‘min ‘| ‘max’ | ‘std’

<SensorIdentifier> ::= number
<SessionFilter>:= <logicalExpression> [{<logicalOp><logicalExpression>}*]
<logicalOp> ::= (And | Or)
<LogicalExpression> ::= <Expression> <OP> <Expression>
<OP> ::=(> | >= | < | <= | = | !=)
<Expression> ::= <term>[(+|-) <term>]*
<term> ::= <factor> [(* | /) <factor>]*
<factor> :=’(‘< Expression>’)’| attributeName | number

The Stream Analytics Query has the following format:

SELECT *<Select Options>*
FROM *<sensorIdentifier>*
WHEN*<sessionFilter>*
[**WINDOW** *<WindowSize>*]
USING *<pricingPolicyIdentifier>*
FOR *<sessionDuration> ;*

Select Options: This refers to sensor data attributes that the consumer wants to receive. Generally, the attributes list can take one of the following values. First, the symbol “*” is used to indicate that the consumer wants to receive all data attributes the sensor provides. Second, one or more sensor data attributes separated by comma “,” e.g., temperature, humidity. Finally, the attribute list might represent aggregation functions over the sensor’s data attributes. The query syntax does not allow multiple attribute-list types in a query, so consumers cannot mix aggregation functions with “*” or data attributes.

Sensor Identifier: The sensor identifier is a value that uniquely identifies the sensor which a consumer wants to receive its data. This value can be extracted from the sensing discovery stage.

Session Filter: The session filter is a logical expression set by the consumer to filter sensor data. The use of the session filter means that the consumer is interested in receiving a subset of sensor data that can satisfy the filter. The filter can be a simple condition such as WHEN temp > 25 or a much more complicated logical expression in which conditions are connected by logical operators such as AND, and OR.

WINDOW *windowSize*: This is an optional part of the query which is used when consumers want to perform stream analytics over a subset of the stream data. There are two possible values for *windowSize*: an *integer value* representing the size of the window in seconds, or the keyword *unbounded*. The integer value is used when the consumer wants to run a Time-based Tumbling Window query. When a consumer uses a Time-based Tumbling Window Query, the middleware retains sensor data for the window size. After each window period, the middleware runs the query over the buffered data, cleans the buffer and pushes the result to the consumer. The middleware repeats this process until the session expire time is due. The keyword, *unbounded*, is used when the consumer wants the middleware to buffer all sensor data throughout the session to help run INSTANTANEOUS queries over the buffered data. An INSTANTANEOUS query is a query that the consumer sends at any point of time during a session to analyze the buffered data.

USING *pricing Policy identifier*: This represents the sensor pricing policy which the consumer wants to use in the session. As mentioned previously, every sensor has multiple pricing policies.

FOR *sessionDuration*: The session duration has two possible values: An integer value that represents the session lifetime in seconds, or the keyword *unbounded* which means that the session duration is uncertain, yet the client application that opened the session can send a request to terminate the session.

We will now present several examples.

Example 1:

```
SELECT temp, hum
FROM Sensor i
USING Pricing Policy j
FOR 360;
```

This query states that a consumer wants to receive the sensor data attributes temperature and humidity from sensor *i* using a pricing policy *j* for the next 360 seconds.

Example 2:

```
SELECT *
FROM Sensor i
WHEN temp>25
USING Pricing Policy j
FOR 360;
```

This query states that a consumer wants to receive all data attributes from sensor *i* if and only if the value of sensor data attribute temp is greater than 25 using a pricing policy *j* for the next 360 seconds.

Example 3:

```
SELECT avg(temp),min(temp),max(temp), avg(hum),min(hum),max(hum)
FROM Sensor i
WHEN temp>25
WINDOW 20
USING Pricing Policy j
FOR 360;
```

This query states that a consumer wants to receive the result of aggregation functions over sensor *i* data attributes every 20 time units. The session filter temp>25 indicates that only sensor stream data tuples that have temp > 25 will be processed in the aggregation functions. The consumer wants to use a pricing policy *j* for the next 360 seconds.

Example 4:

SELECT *temp, hum*

FROM *Sensor i*

WHEN *temp>25*

WINDOW *unbounded*

USING *Pricing policy j*

FOR *360;*

This query states that a consumer wants to the middleware to buffer the specified sensor *i* data attributes for the entire session period to allow the consumer to run INSTANTANEOUS queries over the buffered data. The session filter temp>25 indicates that only sensor stream data tuples that have temp > 25 will be buffered. The consumer wants to use a pricing policy *j* for the next 360 seconds. An INSTANTANEOUS query is any query that the consumer sends during the session, and it takes the following format:

SELECT *<attributeList|aggregats>*

FROM *<sensor I >*

WHEN *<filter>*

As an example of INSTANTANEOUS queries, we might consider the following query:

SELECT *avg (temp),std(temp)*

FROM *sensor I*

WHEN *temp > 20 and hum >80*

This query calculates the average and the standard deviation of the temperature attribute. The middleware runs this query over the buffered data generated by the first query of the session. The middleware sends back the query results to the consumer.

Chapter 4

4 Middleware Architecture

In this chapter, we present the architecture of a Sensing as a Service (SaaS) middleware, which is graphically depicted in Figure 10. Basically, the middleware has three primary layers: The Service Interface layer, the Stream Processor layer, and the Sensors Management and Representation layer. We assume that this middleware is hosted on a cloud platform and interacts with both sensors and client applications. We presume that sensor owners specify policies for how frequently a consumer should receive sensor data and the cost of this frequency. To illustrate, a sensor owner might specify the following policy: To receive the sensor's readings every 10 seconds, a client must be charged $X every minute. The cost increases as the frequency increases. The price policy cannot use a frequency, which is higher than the physical sensor's actual frequency of sending data. The following is a brief description of the middleware layers.

4.1 Architecture overview

The Service Interface layer receives client requests, directs these requests to the Stream Processor for stream operations handling. Stream processor, then, replies the requests through the Service Interface. For client applications, the Service Interface hides the underlying knowledge and technology required to deploy, manage, and transfer sensor data, so that client applications need only to know which service interface method to use.

Secondly, the Stream Processor is the core of the proposed middleware. It is responsible for executing client application queries and charging client applications based on the chosen sensor pricing policy and data consumption. After executing a query over a stream, the Stream Processor returns the query result to the client application through the Service Interface.

Finally, the Sensor Management and Representation layer manages the connection between the Stream Processor and physical sensors. Furthermore, this layer maintains the sensor deployment and pricing policy database.

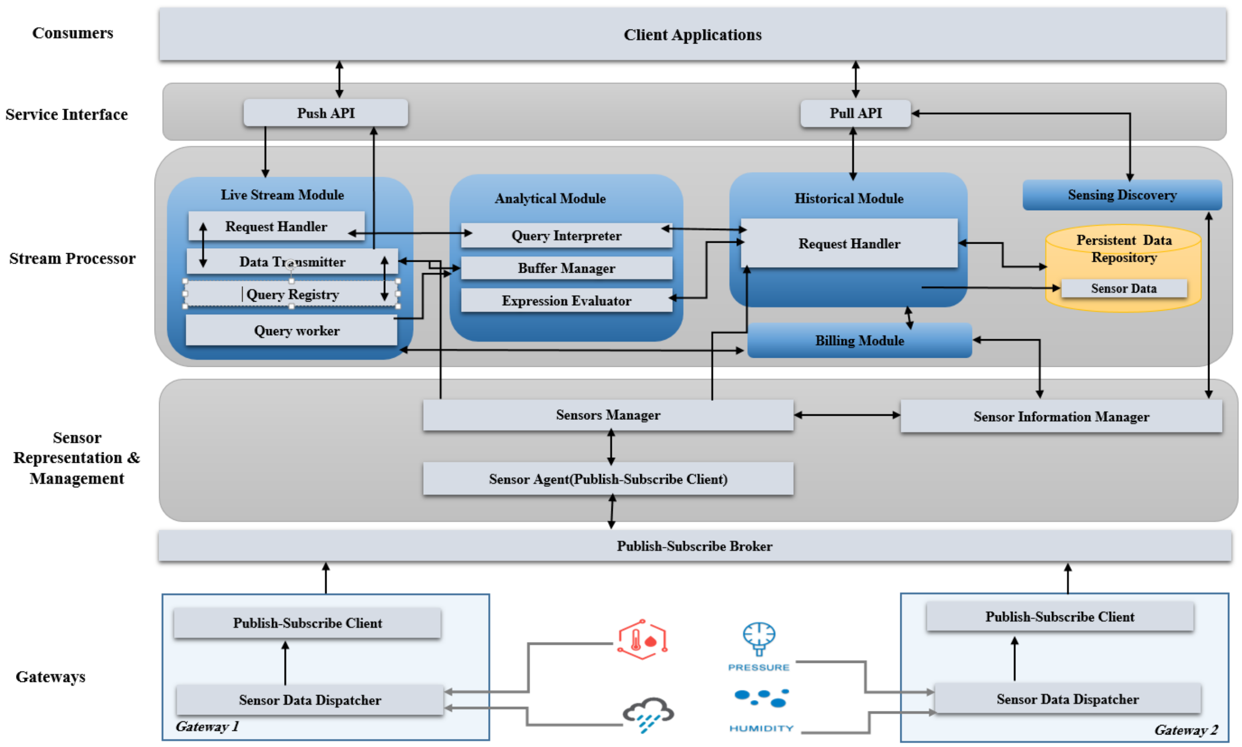


Figure 10 Middleware Architecture.

4.2 Service Interface

The Service Interface layer receives client requests and invokes relevant services after verifying a consumer's right to invoke a service. This section describes the communication paradigms used and the format of request messages.

4.2.1 Service Interface Communication Protocols

The push and pull models designate two ways of exchanging data between two distinct entities. In this work, the entities are client applications and the middleware. The pull model is based on the request/response paradigm. The response may be sent synchronously or asynchronously. With the push model, a client subscribes to data providers and new content is automatically sent to the client. The push protocol is graphically depicted in Figure 11. Since push allows multiple responses per request, it is preferred over pull when data volume and velocity are high. To handle different ways of client-server communications, the middleware provides two kinds of interfaces through which the clients can interact with the middleware: an interface that provides an API that uses pull communication protocol used for single response per request communication paradigm and an interface that provides an API that uses push communication protocol for multiple responses per request communication paradigm.

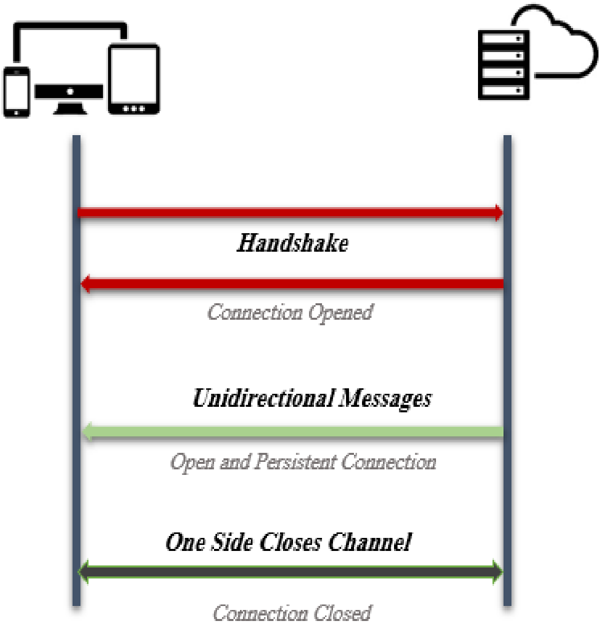


Figure 11 Push Protocol.

4.2.2 Request Format

A service interface request consists of two variables. The *Authentication Token* is a unique token generated by the middleware and is provided to clients at the end of the client registration process. Client applications must submit this token in every request to ensure that the client application is allowed to carry out the request. The *command* variable represents the query.

4.3 Query Interpreter

The Query Interpreter breaks down queries into smaller elements where each element is translated into a command that is understandable by the other components of the middleware. For example, there are queries that require the buffering of data. In the middleware, the Buffer Manager uses an in-memory NoSQL database. The Query Interpreter translates the query to commands that can be understood by the NoSQL database. For example, consider the following query to be executed over a buffer:

*SELECT * FROM SensorId WHEN temperature >=0 and temperature <= 10 Using P_i For 3600.*

The Query Interpreter generates the following code to be passed to the Buffer Manager (described in Section 4.5.2) to execute:

query.sensorBufferName.find(Filter.and(Filter.gte(“temperature”,0),Filter.lte(“temperature”,10)))

To do this task, the Query Interpreter generates an Abstract Syntax Tree (AST) and then traverses the tree to generate the code for commands that can be understood by other middleware components. Figure 12 shows the structure of the AST. Figure 13 shows the Abstract Syntax Tree created by the Query Interpreter to translate the query provided in the example.

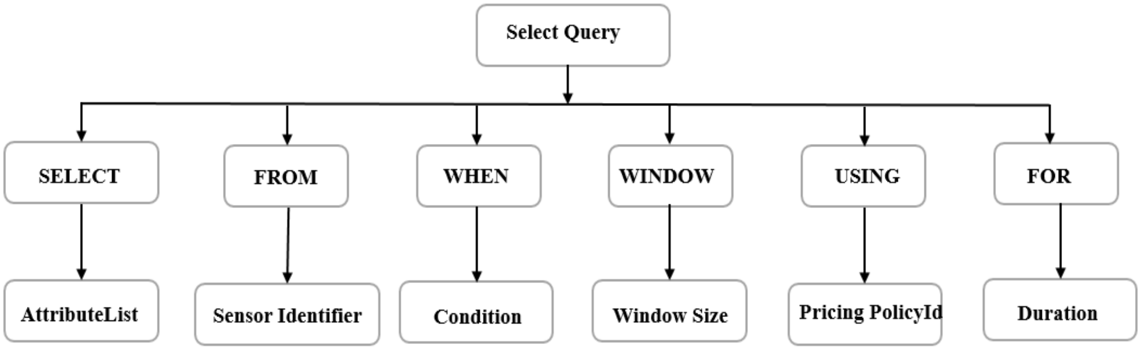


Figure 12 Abstract Syntax Tree Structure.

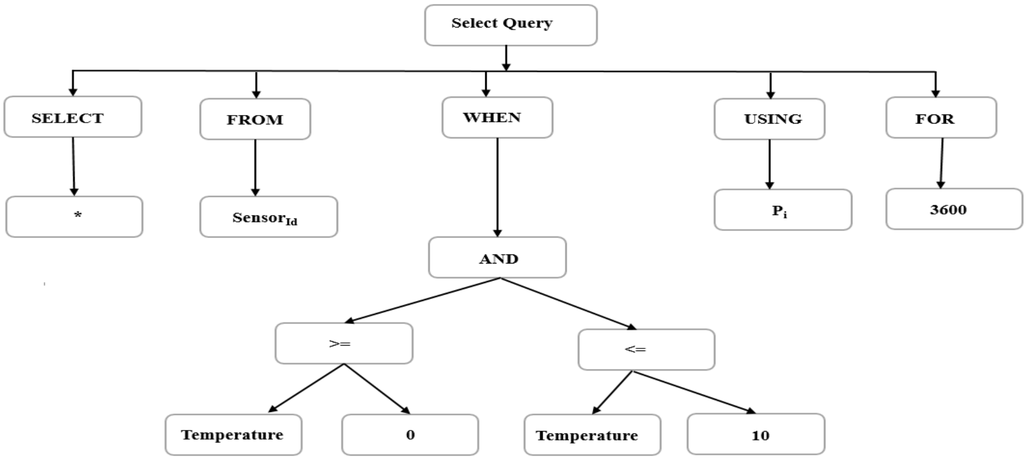


Figure 13 AST Generated to translate a query.

4.3.1 Expression Evaluator

This component evaluates a combination of arithmetic and logical expressions specified in the query filter (i.e., WHEN clause). The Expression Evaluator is used when a query has a filter in order to check whether a sensor data tuple satisfies a condition. Basically, the Data Transmitter (see section 4.5.1.3) passes the sensor data and the filter to the Expression Evaluator which replies with the comparison result. If a sensor data tuple satisfies the condition, the tuple is pushed to the client application. To understand how the Expression Evaluator works, let us consider the following example:

A sensor S sends data represented by four data attributes that are referred to as a, b, c and d. A client application wants to execute a query with the following filter:

((a=9) and (b=c) Or ((c*(a+9)) > (9+a)) and (d<=c))

When the Data Transmitter receives sensor data from the sensor, the Data Transmitter passes sensor data and the parse tree of the client application's filter to the Expression Evaluator. The Expression Evaluator modifies the tree by substituting sensor data attribute names with their values in the message. After that, the Expression Evaluator traverses the tree to evaluate the session condition and then sends evaluation results to the Data Transmitter. The result of the evaluation is either true or false. Figure 14 shows the session condition parse tree that is built by the Query Interpreter.

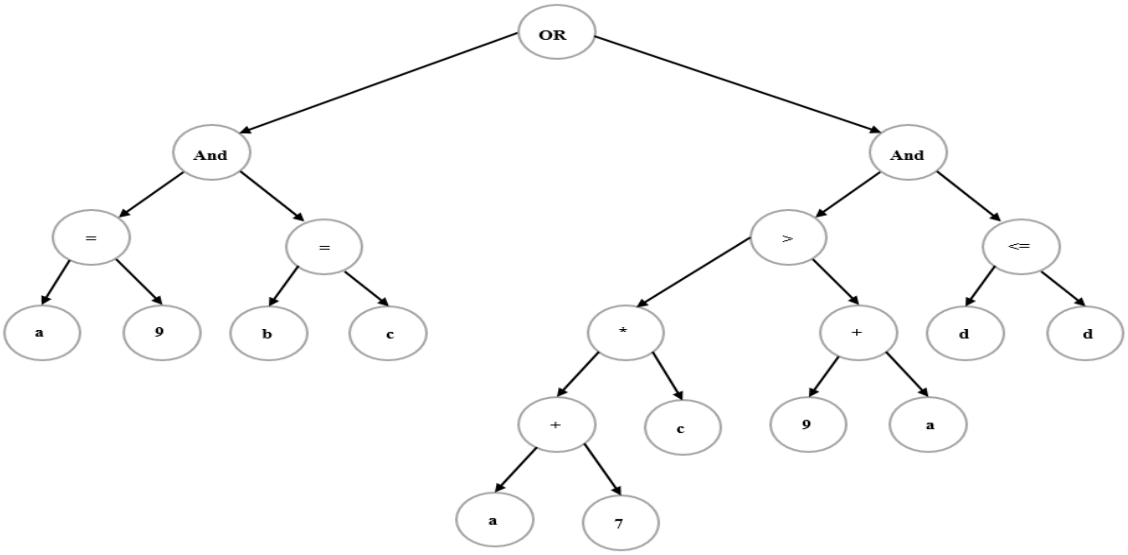


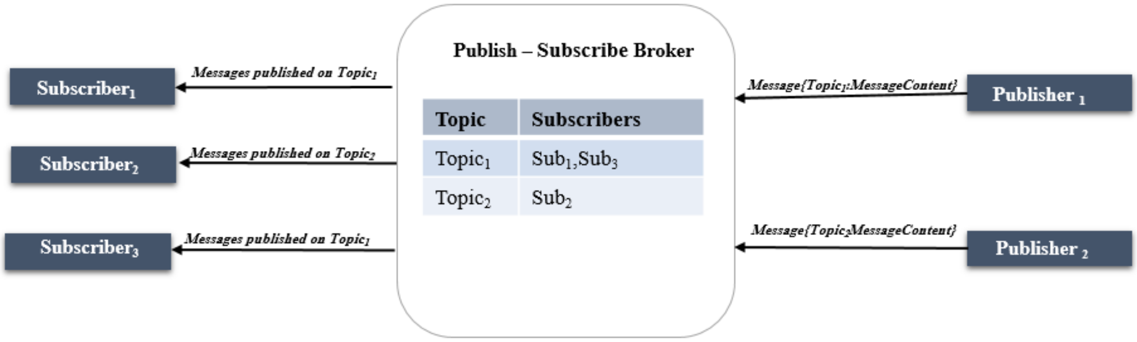
Figure 14 Session Condition Tree.

4.4 Sensor Management and Representation

This layer handles the connection between the middleware and the sensors. More specifically, the Stream Processor layer connects to sensors through this layer. Abstractly, the connection between the sensors and the middleware is managed by a cloud-based Publish-Subscribe Broker. In our design, both middleware components and the sensors are clients of the Publish-Subscribe Broker. These clients have to know how to communicate with the broker. For this reason, we split this layer into three different, yet related components: Sensors Manager, Sensor Agent, and Sensor Data Dispatcher.

4.4.1 Publish-Subscribe Pattern

The Publish-Subscribe is messaging pattern that supports a bidirectional messaging approach in which data sources publish data on a topic and potential data consumers subscribe to that topic. Typically, the data publishers and consumers do not directly communicate with each other. The interaction is done through an intermediary system which is referred to as a “message broker.” In this work, a Publish-Subscribe communication pattern is used for managing the connection between sensors and the middleware. Sensors publish data to a messaging broker using a Publish-Subscribe client, and the middleware receives sensor data from the messaging broker using a Publish-Subscribe client. The rationale of using a Publish-Subscribe protocol as a communication protocol between the middleware and the sensors is that the Publish-Subscribe protocol is a lightweight communication protocol that is designed for devices with limited power, and computational resources [64].


Publish – Subscribe Broker
Subscriber$_1$
Subscriber$_2$
Subscriber$_3$
Messages published on Topic$_1$
Messages published on Topic$_2$
Messages published on Topic$_1$
Message{Topic$_1$:MessageContent}
Message{Topic$_2$MessageContent}
Publisher $_1$
Publisher $_2$

Topic	Subscribers
Topic$_1$	Sub$_1$,Sub$_3$
Topic$_2$	Sub$_2$

Figure 15 Publish Subscribe Protocol.

When a sender publishes a message to the broker, the sender tags the message with a topic that identifies the message. A receiver sends a tag asking the broker to send any message that has that tag attached to it. The tag is called a *message topic*. The receiver must know the topic that the sender uses for tagging their messages. Once the broker receives a message from the sender, the broker sends the message to all clients who requested messages tagged with the topic associated with the sender. An example of Publish-Subscribe connection is graphically depicted in Figure 15. In our middleware, the Sensor Manager generates a unique topic for each sensor. This topic is used by the middleware and Sensor Data Dispatcher (see section 4.4.2 and 4.4.4), which publishes sensor data on behalf of a sensor.

4.4.2 Sensor Manager

The Sensor Manager manages sensor registration and connectivity. For sensor registration, the sensor owner specifies the sensor’s data scheme template which describes the sensor’s data attributes and their data types. Generally, a sensor data scheme is more formally defined as a set of pairs where a pair is of the form (attribute: datatype).

For example, a sensor that measures temperature and humidity might have the following data scheme:

{Temperature: double, Humidity: double}

At the end of the sensor registration process, the Sensor Manager stores the sensor data scheme in a database alongside a unique topic identifier to be used by the Publish-Subscribe clients. After that, the Sensor Manager generates a Sensor Data Dispatcher that can be deployed in the gateway that the sensor is associated with. Furthermore, the Sensor Manager manages the connection between a Sensor Data Dispatcher and the Stream Processor through a Sensor Agent. The Sensor Manager is responsible for creating the Sensor Agent and providing it with the required information to receive sensor data from the Publish-Subscribe Broker.

4.4.3 Sensor Agent

The Sensor Agent delivers inbound sensor data streams to the Stream Processor. As previously mentioned, sensors publish their readings to the Publish-Subscribe Broker which passes the readings to the middleware. In order to deliver a sensor data stream to the Stream Processor, the Sensor Manager creates a Sensor Agent which is a client of the Publish-Subscribe Broker and provides the Sensor Agent with the unique topic that is used by the sensor when publishing its data. The Sensor Agent subscribes to the Publish-Subscribe Broker using the given sensor topic. The Sensor Agent supplies sensor data to the Historical Module and the Live Stream Session Module.

4.4.4 Sensor Data Dispatcher

A Sensor Data Dispatcher is software generated by the Sensor Manager at the end of the sensor registration process. This software runs on a gateway associated with one or more sensors. A gateway is an embedded device with computing and networking capabilities where software can be executed to fetch the data from sensors as well as send the data to a remote server. Gateways are usually placed in close vicinity to the sensors. The connection between the sensors and the gateway can be wired or wireless. The Sensor Data Dispatcher software communicates with the Publish-Subscribe Broker through a Publish-Subscribe Client and knows how to structure the sensor's data in the format that the service broker understands. The Sensor Data Dispatcher knows the topic to be used for publishing sensor readings, and the sensor data scheme. It is important to note that we assume the sensor owner deploys the generated Sensor Data Dispatcher in the gateway to which the sensor is attached. In addition, we also assume that the sensor owner modifies the generated Sensor Data Dispatcher to have it read information from the physical sensor. The rationale for this decision is that there is no unified approach that can be followed to pull readings from a physical sensor. To illustrate, the connection between a sensor and a gateway can be wired or wireless. If the connection is wired, then the way the wiring is done is different from one sensor to another. The same problem applies to the wireless connections. Furthermore, there are many sensor vendors, and the way sensor connection is handled differs among vendors. For example, let us assume $Sensor_i$ and $Sensor_j$ are both air quality sensors manufactured by vendors *A* and *B*. The connection for both sensors is a wired connection. Vendor *A* specifies that the connection requires three wires. The wires have to be connected to the gateway GPIO pins 0, 6 and 7 respectively. On the other hand, vendor *B* specifies that the connection requires two wires. The wires have to be connected to the gateway

GPIO pins 1and 5, respectively. Even though both sensors provide the same service, the way they are connected to the gateway is completely different. For this reason, the Sensor Manager generates a software, Sensor Data Dispatcher, that knows how to connect to the Publish-Subscribe Broker and assumes that the sensor owner can update the software to pull data from the physical sensor.

4.5 Stream Processor

The Stream Processor is responsible for executing client application queries over sensor data streams. A sensor data may need to be analyzed quickly in order to send a notification which we call a Live Stream Session query. However, sensor data could be analyzed to detect long-term trends which we call a Historical Session query. A Stream Processor receives client applications queries through the Service Interface. It then takes action based on the request type. To distinguish a Live Stream query from a Historical query, the Service Interface provides different API functions for Live and Historical queries. In addition, the Stream Processor responds to sensing discovery queries in which queries are executed against the Sensor Database that has the knowledge of sensor locations, data scheme, and pricing policies. Once the Stream Processor executes a query, the result is sent to a client application through the Service Interface. The Stream Processor consists of four modules: Live Stream Session Module, Historical Session Module, Belling Module, and Sensing Discovery Module.

4.5.1 Live Stream Session Module

The Live Stream Session Module handles consumer real-time data stream requests. We use the term Live Session to refer to a request in which a client application is interested in receiving a live stream of data from a sensor for a period of time. For example, a client application might be interested in receiving a sensor stream of data for the next 30 minutes. The Live Stream Session Module receives client requests from the service interface, identifies which sensor the consumer wants to query, collects sensor information, executes the query, and sends the query result to the consumer. To handle those tasks, the Live Stream Session Module has four main components**:** Request Handler**,** Query Registry, Data Transmitter, and Sensor Stream Buffer.

4.5.1.1 Request Handler

The Request Handler is responsible for managing Live Stream Session Module components. It creates the Query Registry and Sensor Data Transmitter. It communicates with the Query Interpreter to parse client application requests and with the Sensor Representation & Management Layer to open a connection with a sensor.

4.5.1.2 Query Registry

Each sensor is associated with a Query Registry that is used to maintain information for each active query application that is using the sensor. A query registry tuple representing an active query application consists of information, extracted by the Query Interpreter, and includes the sensor identifier, the session filter, message receiving frequency (i.e. how frequently a client application should receive sensor data), pricing policy, attribute list (e.g., the part of the query right after the SELECT keyword) and query type. Once a consumer session is over, its tuple is removed from the Query Registry.

4.5.1.3 Data Transmitter

The Data Transmitter is responsible for delivering query results to consumers. Upon receiving sensor data from the Sensor Agent, The Data Transmitter uses the Query Registry to determine the consumers of the sensor data. For each consumer, the Data Transmitter determines if the consumer should receive the sensor data, based on the frequency of the requested pricing policy. The Data Transmitter ensures that the consumer receives a single sensor observation for every time frame specified in the pricing policy, e.g., every 20 seconds. If the consumer is allowed to receive the sensor data, the transmitter follows the consumer's query execution strategy to deliver the message. The Live Stream Session query execution strategies are discussed in Section 4.5.1.4.

To reduce network traffic and communication load on the broker, only one Sensor Agent is created per sensor to serve all ongoing live sessions. This is graphically depicted in Figure 16. Upon receiving sensor data, the Sensor Agent passes the data to the Data Transmitter which takes care of the rest of the delivery process. This Sensor Agent connection with the Publish-Subscribe Broker is terminated when there is no open live session with the sensor. It is important to note, that the Request Handler creates one Data Transmitter per sensor as shown in Figure 16. When the first live stream session request arrives at the Request Handler for a sensor, the Request Handler creates a Data Transmitter and uses the session duration of the first request as the Data Transmitter's expiration time. Whenever a new live stream session request for the same sensor arrives, the Request Handler does not create another Data Transmitter. Instead, it updates the Data Transmitter expiration time if needed.

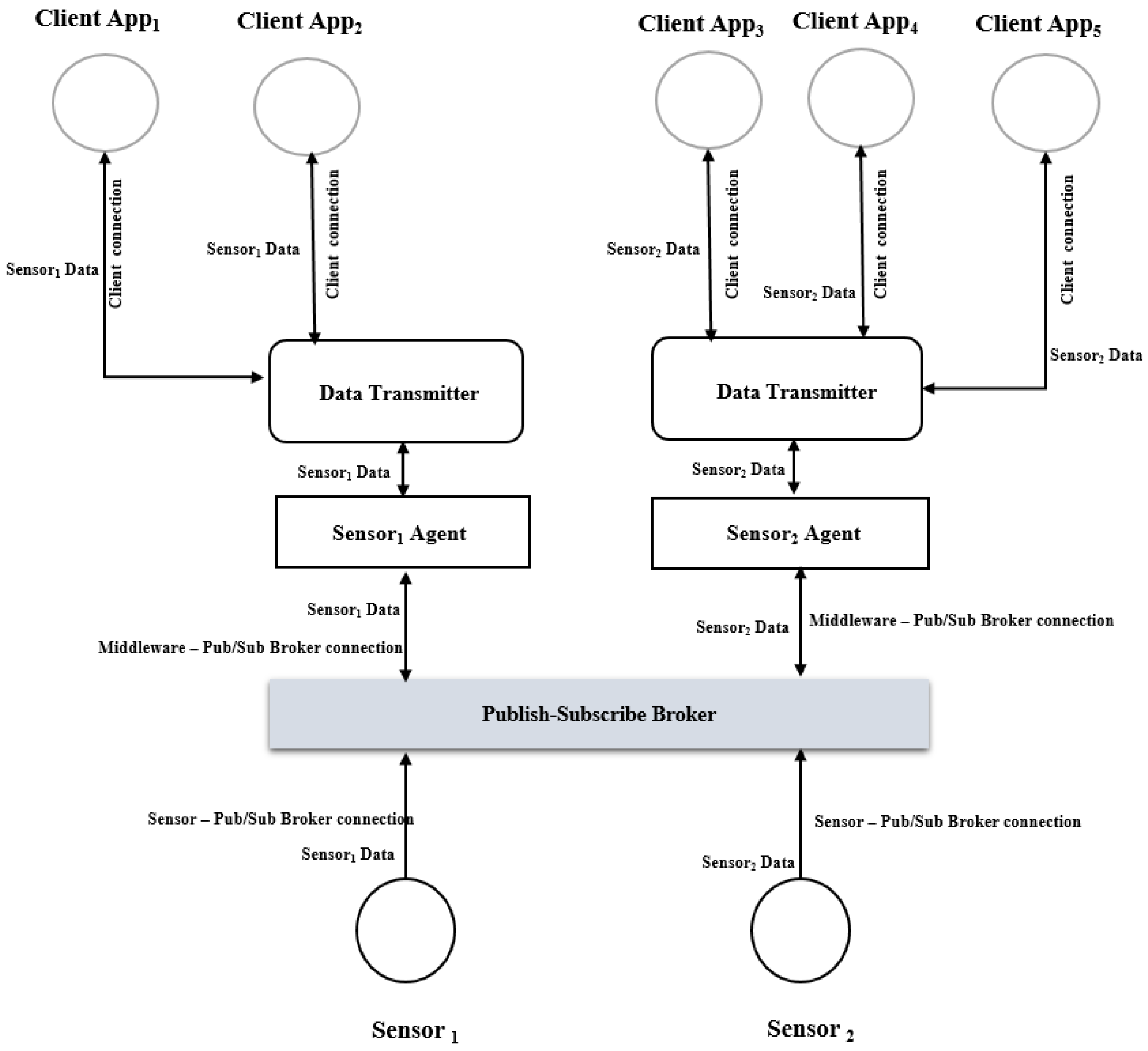


Figure 16 Data Transmitter.

4.5.1.4 Query Execution Strategies

A Query Execution Strategy is the way Data Transmitter executes client applications queries over inbound sensor data streams. The Query Execution Strategy is determined by the type of query which is determined by the query interpreter based on the query structure. Currently, the middleware supports four types of query execution strategies: a raw data query, filtered raw data query, Time-based Tumbling Window query, and instantaneous query.

4.5.1.4.1 Raw Data Query

A Raw Data Query is a query that does not specify a filter nor require any data processing operations (e.g., aggregation functions). A client application query that takes the following format is considered a raw data query:

SELECT *<attribute List>* ***FROM*** *<SensorIdentifier>* ***USING*** *<pricingPolicy>* ***FOR*** *<sessionDuration>;*

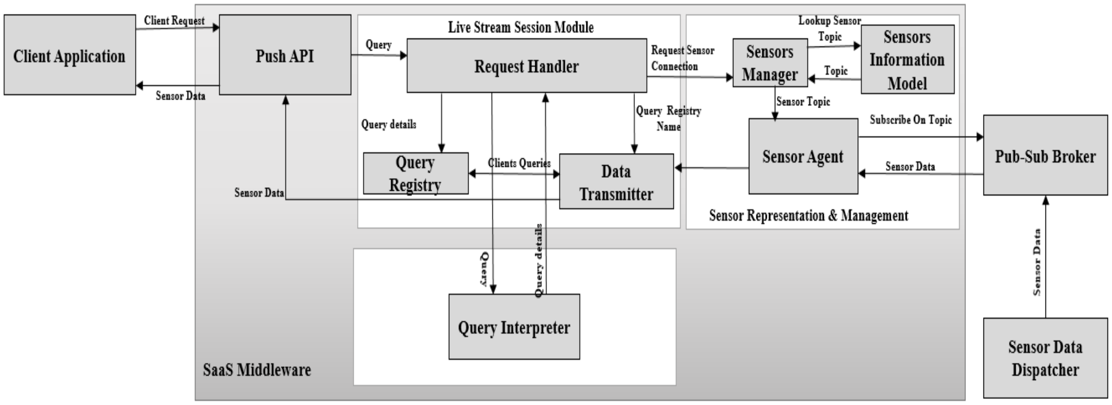


Figure 17 Raw Data Query Execution.

As we can see the query does not have a filter (i.e., WHEN clause) and does not specify a window. The query is executed as follows during the session period. Whenever, the Data Transmitter receives sensor data, if the client can receive the message based on the pricing policy frequency, the Data Transmitter pushes the sensor message to the client application. When the client application session duration expires, the Data Transmitter removes the client application query tuple for the sensor from the Query Registry. Stream Processor components interaction for a Raw Data Query Execution in Figure 17.

4.5.1.4.2 Filtered Raw Data Query

A filtered raw data query is a query that specifies a filter but does not require data processing operations (e.g., aggregation functions). A client application query uses the WHEN clause is considered a filtered raw data query.

SELECT <*attributeList*>
FROM <*SensorIdentifier*>
WHEN <*SessionFilter*>
USING <*pricingPolicy*>
FOR <*sessionDuration*>

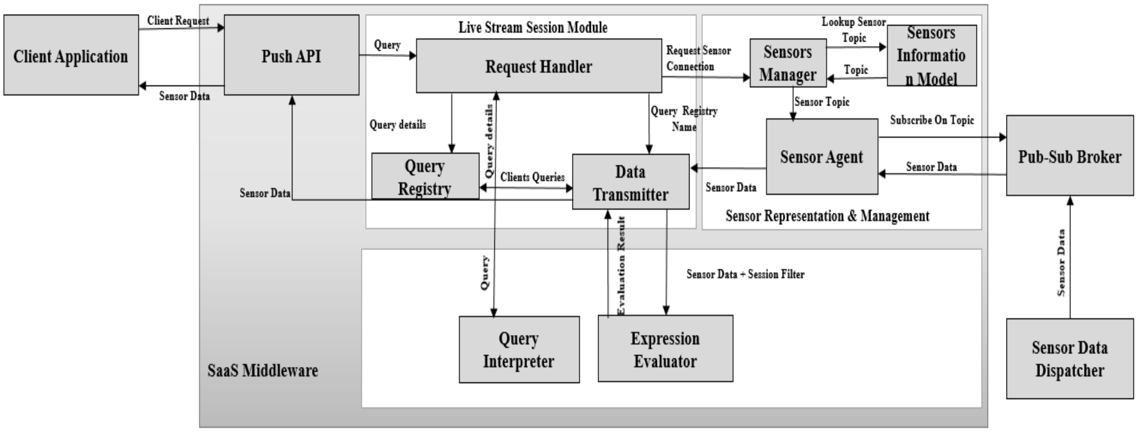


Figure 18 Filtered Raw Data Query Execution.

This query follows the same execution strategy as the raw data query in the sense that the Data Transmitter passes the sensor data to the client application without any further processing. The presence of a filter implies that the client application is only interested in receiving sensor data that satisfies the session filter, so the Data Transmitter evaluates the session filter and pushes the data to the client only if the result of condition evaluation is *TRUE*. The session filter evaluation is done by the Expression Evaluator. The Stream Processor components interaction for a Filtered Raw Data Query Execution is graphically depicted in Figure 18.

4.5.1.4.3 Time-based Tumbling Window Query

A Time-based Tumbling Window Query (TTW) is a query that requests the execution of aggregation functions over a subset of the sensor data stream. Time-based Tumbling

Window splits the stream into non-overlapping portions and executes the aggregation functions over stream portions in a successive manner [31]. The subset size is defined by the window size. This query re-executes itself every N seconds where N is specified in the WINDOW clause. A TTW query remains active for the number of seconds specified in the FOR clause. A client application query that takes the following format is considered a continuous query:

SELECT avg(temp), min(hum), max(temp)
FROM sensorId
WHEN temp > 25
WINDOW 120
USING policyId
FOR 3600

This request means the user ***U*** wants to open a session with a Sensor ***S***, identified by sensorId which is known from the sensor discovery stage, for the next 3600 seconds using a pricing policy ***P***, identified by policyId. The keyword **WINDOW** followed by an integer number indicates that the query type for this session is a Time-based Tumbling Window query (TTW) which requires that the Data Transmitter buffers sensor data for a certain period of time specified in the window clause. For the example query, this is 120 seconds. The Data Transmitter executes the aggregation function and pushes the result to the client application. The Data Transmitter re-executes the query every ***N*** seconds (e.g., 120 in the example query), specified in window clause, by using a query worker that remains active for the number of seconds specified in FOR clause. A Time-based Tumbling Window query requires buffering the sensor data for a specified period of time and then executing aggregation functions, sending the result to the client, and then cleaning the buffer for another execution cycle. To handle a continuous query, the Data Transmitter uses two components: A Sensor Stream Buffer and a Query Worker.

- **Sensor Stream Buffer:** Sensor data is held for a certain amount of time in the Sensor Stream Buffer. The Data Transmitter is a producer of data for the buffer while the Query Worker consumes the buffer. The Buffer Manager creates a single buffer per sensor regardless of the number of client applications that use the buffer. Sensor stream buffer management is discussed in section 4.5.2.1.

- **Query Worker**: This represents a user application of the data in Sensor Stream Buffer. Essentially, there is a query worker for each user application. The Query Worker is responsible for running the query every ***N*** seconds and pushing the query results to the client application. After each run, the Query Worker sleeps until the next query execution cycle. A Query Worker is a thread that executes time-based tumbling window query over the sensor stream buffer. The query worker uses two variables $\text{Window}_{\text{lowerBound}}$ and $\text{Window}_{\text{upperBound}}$ to slide over the sensor stream buffer. On each run, the query worker advances the window boundaries by the value of window size specified in the Query Registry tuple. Subsequently, the Query Worker sends the window boundaries and the Query Registry tuple to Buffer Manager to stream buffer data that falls within the window boundaries. Window boundaries specify a portion of the buffer to be used for as a query dataset. The Query Worker then pushes the query result to the client application and prepares for the next execution cycle by advancing the window's upper and lower boundaries. Essentially, $\text{Window}_{\text{lowerBound}}$ is set to $\text{Window}_{\text{upperBound}} + 1$, and $\text{Window}_{\text{upperBound}}$ is incremented by the window size.

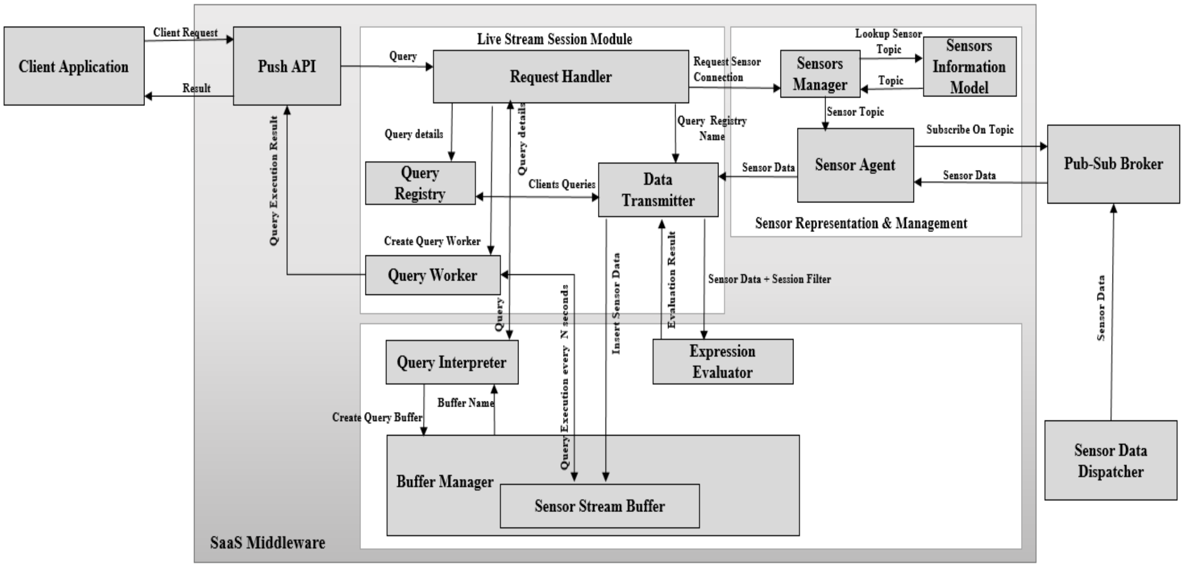


Figure 19 : Time-based Tumbling Window Query Execution.

The Stream Processor components interaction for a Time-based Tumbling Window query execution is graphically depicted in Figure 19. As with any other query, the query arrives at the Request Handler through the Service Interface. The Request Handler passes the query to the Query Interpreter which returns a tuple of query details to the Request Handler. It then adds the tuple to the Query Registry and starts the Data Transmitter if it has not already been started. After that, when the Data Transmitter finds that the query type is a Time-based Tumbling Window query, the Data Transmitter creates a Query Worker. The Query Worker expiration time is the same as the Query Registry tuple expiration time. After that, when the Data Transmitter receives sensor data, the Data Transmitter checks if there are active queries in the Query Registry that require buffering. If so, the Data Transmitter instructs that the Buffer Manager creates a Sensor Stream Buffer if the buffer has not already been created. The Data Transmitter then puts the sensor data in the Sensor Stream Buffer. Furthermore, the Query Worker periodically wakes up to execute the query based on the window clause. The query is executed over a subset of the Sensor Stream Buffer. The way a subset of the Sensor Stream Buffer is extracted for analysis by the Query Worker is discussed in section 4.5.2.1.

4.5.1.4.4 Instantaneous Query

Instantaneous queries give client applications the ability to analyze sensor data stream using multiple queries throughout the session. An Instantaneous query shares some similarities with a Time-based Tumbling Window query in the sense that it defines a window and might use aggregation functions. However, the window semantics for Instantaneous queries is quite different from Time-based Tumbling Window queries. Instantaneous queries use a lower-bounded landmark window which is a window that has its lower bound fixed, but its upper bound advances with time (see section 2.4.3.2). Another major difference is that Instantaneous queries allow client applications to analyze the window subset using different queries (e.g., they can change the aggregation functions and the session filter) whereas client application cannot update a Time-based Tumbling Window once a session started. To better understand Instantaneous queries, let us consider a client application query that takes the following format:

SELECT temp, hum
FROM sensorId
WHEN temp > 25
WINDOW unbounded
USING policy ID
FOR 3600

This query is called a session opening query. It tells the middleware that user ***U*** wants to open a session with a Sensor ***S*** for ***N*** seconds using pricing policy ***P***. The keyword *window* followed by the keyword *unbounded* indicates that the client application requests a window of a type lower-bounded landmark window.

An Instantaneous Query is a query that the client application submits during the session to be executed over the buffered data. This query is answered directly by the Request Handler. Throughout the session, the client application can send as many instantaneous queries as possible. Instantaneous queries have the following format:

SELECT <attributeList|aggregates**>** **FROM** sesnsorId **WHEN** <Condition>

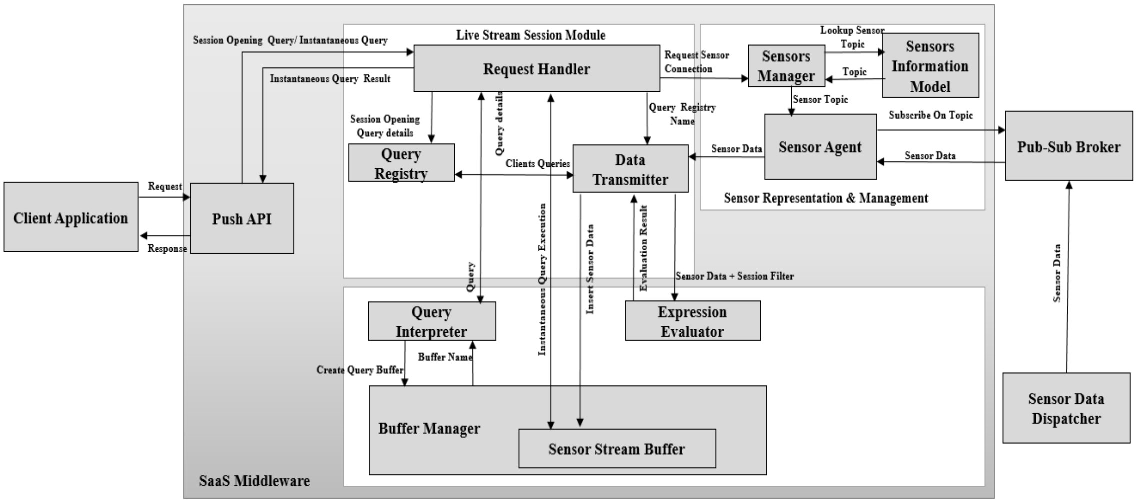


Figure 20 Instantaneous Query Execution.

Stream Processor components interaction for an Instantaneous query execution is graphically depicted in Figure 20. When the Request Handler recognizes that the user wants to run an Instantaneous query, it inserts the query details in the Query Registry. When the client application sends an Instantaneous Query, the Request Handler executes the query over a subset of the Sensor Stream buffer and asks the service interface to push the results to the client application. When the Client Application Session is completed, the Data Transmitter deletes the client application query from the Query Registry and asks the Buffer Manager to clean the buffer if no other client application needs the sensor data that is stored in the buffer.

4.5.2 Buffer Manager

The Buffer Manager is an in-memory database engine used to handle sensor stream buffer operations (e.g., create, insert, select, delete). This is used to avoid disk I/O operation overhead. Aggregation functions over buffered data are applied by the buffer manager.

4.5.2.1 Sensor Stream Buffer Management

Execution strategies for Time-based Tumbling Window query and Instantaneous query require buffering sensor data for a certain amount of time. This section describes the management of the Sensor Stream buffer that satisfies these characteristics: (1) The sensor data should be buffered in memory to avoid disk I/O overhead; (2) Sensor data should not be buffered unless there are ongoing client application sessions that require executing continuous or instantaneous query; (3) As memory storage is expensive due to the size limitation, a single buffer per sensor should be created, so that data redundancy is avoided.

However, using a single buffer per sensor poses other challenges. First, we assume that client applications receive sensor data at different frequencies. For example, let us assume that sensor *S* sends data every second. Client application C_i and C_j are interested in receiving sensor *S's* data. However, C_i uses a pricing policy that states that the client must receive sensor data once every 10 seconds while C_j uses a pricing policy that states that the client must receive sensor data once every 30 seconds. If sensor data is stored in a single buffer, there must be a way to extract subsets of the sensor stream buffer that can be used to execute the queries from C_i and C_j. Second, sensor stream buffer tuples that no client application can use should be deleted. To illustrate, let us assume that client application C_i started its session with sensor *S* at 10.00 am, and the session ends at 11.00 am. Let us then assume C_j started its session with the same sensor *S* at 10.30 am and the session ends at 11.30 am. Both client applications ask to execute continuous queries which require buffering sensor data. The sensor stream buffer should hold sensor data that arrived between 10.00 am, and 11.30 am in order to execute client applications queries. When the session of client C_i is completed, we know C_j will not benefit from the data stored in the buffer for the period between 10.00 am, and 10.29 am because the C_j session started at 10.30 am, so the portion of the buffer that C_j is not using should be deleted. When the client C_j's session is completed, the buffer can be deleted as no other client application is currently running a query that requires buffering sensors data.

We manage the Sensor Stream Buffer in a way that addresses these challenges. First, in order to extract a subset of the Sensor Stream Buffer that can be used to execute client $C_i{}'s$ query, we take the following steps. Sensor data messages (e.g., sensor tuples) are tagged with a timestamp upon arrival at the Data Transmitter. The Data Transmitter then checks the Query Registry to see if there is at least one client application query that requires buffering sensor data. If so, the sensor data tuple is placed in the buffer. We propose a multitenancy algorithm to extract client application subset from sensor buffer. The proposed algorithm is to be used when executing client application queries over sensor data buffers.

4.5.2.2 Subset Extraction using Modulo Operator

In this algorithm, we employ remainder after division operation (modulo operation) to extract a subset of the Sensor Stream Buffer that a client application query should be executed over. The timestamp of the first sensor message that arrives after a client application session has begun is essential in extracting the client application subset of the Sensor Stream Buffer.

For each tuple in the sensor stream buffer, a tuple belongs to the client application subset if and only if the tuple satisfies the following condition:

$$(t_{ij} - t_{i0})\ Mod\ (f_q) = 0$$

where

t_{ij} represents the arrival timestamp of tuple t_i

t_{i0} is the timestamp of the first tuple that arrived after the start of the client session.

f_q is the pricing policy message arrival frequency

We formally define a client application buffer subset as follows:

$$C_{subset} = \{s_i \mid s_i \in S \wedge t_{ij} - t_{i0} \bmod f_q = 0\}$$

If the client application uses a window operation over a stream, then a client application subset for a window that starts at W_{t1} and ends at W_{t2} is defined as follows:

$$CW_{subset} = \{s_i \mid s_i \in S \wedge t_{ij} - t_{i0} \bmod f_q = 0 \wedge t_{ij} \geq w_{t1} \wedge t_{ij} \leq w_{t2}\}$$

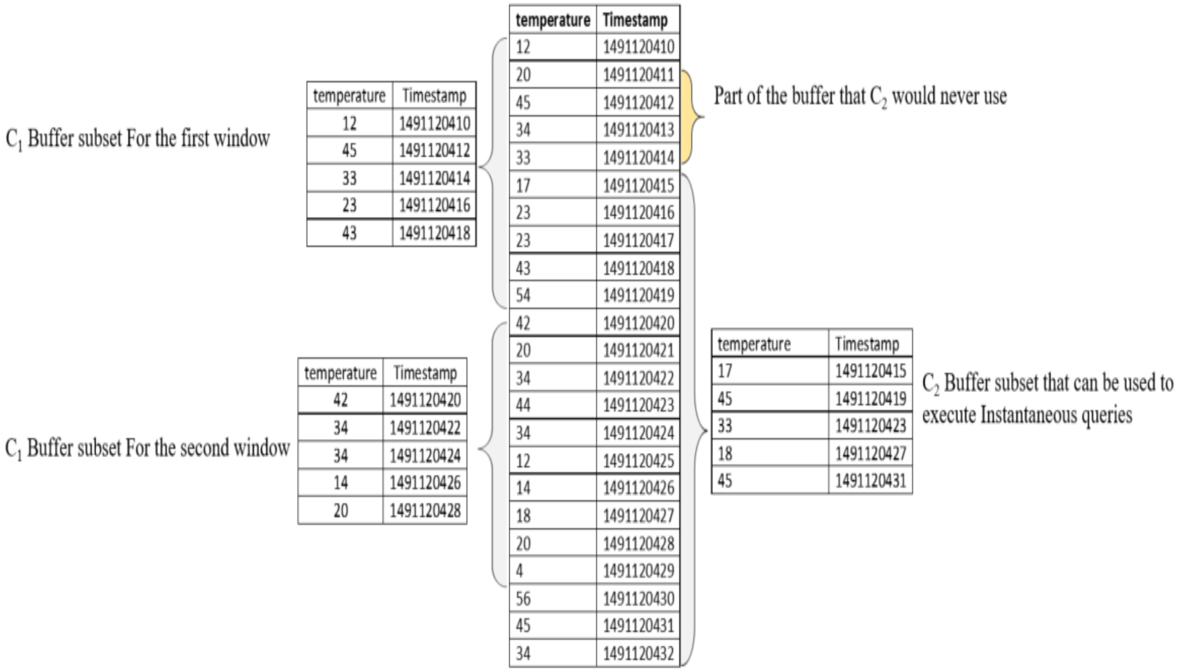
Example:

Sensor *S* sends messages every x seconds. The sensor data scheme is {temperature: double}. Clients C_1, and C_2 are interested in receiving sensor S's data. C_1 and C_2 submitted the following queries:

C_1 query: Select avg(temperature) from Sensor *S* window 10 using P_1 For 120
C_2 query: Select temperature from Sensor *S* window unbounded using P_2 For 120

C_1's session starts at 10.00.00 AM and ends at 10.02.00 AM. P_1 is a pricing policy stating that C_1 wants to get a message every 2 seconds. C_1 wants the middleware to run a Time-based Tumbling Window query with a window size 10 seconds which means buffer the sensor data for 10 seconds and then send the average of temperature to C_1. On the other hand, C_2's session starts at 10.00.15 AM and ends at 10.03.15 AM. P_2 is a pricing policy that states C_2 wants to get a message every 4 seconds. C_2 's query states that the middleware buffers all sensor data for the next two minutes so C_2 can execute instantaneous queries over the buffered data.

Figure 21 shows a snapshot of the sensor stream buffer that shows how the middleware manages the sensor stream buffer. The middleware creates a single buffer. When the middleware executes a client application query, the middleware executes the query over a subset of the buffer. As shown in Figure 21, 5 tuples are extracted from the buffer to calculate the temperature average for the first window in C_1 query. As for C_2 client application, whenever *C₂* submits an instantaneous query, the middleware extracts a subset to be used to answer the query.


C$_1$ Buffer subset For the first window
Part of the buffer that C$_2$ would never use
C$_1$ Buffer subset For the second window
C$_2$ Buffer subset that can be used to execute Instantaneous queries

temperature	Timestamp
12	1491120410
20	1491120411
45	1491120412
34	1491120413
33	1491120414
17	1491120415
23	1491120416
23	1491120417
43	1491120418
54	1491120419
42	1491120420
20	1491120421
34	1491120422
44	1491120423
34	1491120424
12	1491120425
14	1491120426
18	1491120427
20	1491120428
4	1491120429
56	1491120430
45	1491120431
34	1491120432

temperature	Timestamp
12	1491120410
45	1491120412
33	1491120414
23	1491120416
43	1491120418

temperature	Timestamp
42	1491120420
34	1491120422
34	1491120424
14	1491120426
20	1491120428

temperature	Timestamp
17	1491120415
45	1491120419
33	1491120423
18	1491120427
45	1491120431

Figure 21 Client Application Buffer Subset Extraction

After the execution of *C₁* ’s first window, the query worker asks the buffer manager to delete the first five tuples in the buffer. This is graphically depicted in Figure 21 with tuples highlighted by yellow color since no other client application can use the first five tuples other than *C₁* in the first window as *C₂*’ session started fifteen seconds after *C₁*’s session. In other words, Buffer elements that arrived before *C₂*’s sessions had begun, should be deleted when *C₁* no longer needs them.

4.5.2.3 Deleting unused buffer tuples

Deleting buffer tuples that cannot be used by any client application is crucial for the middleware resource management because buffers are stored in the memory. As client applications share the buffer for a certain amount of time, there should be a mechanism for deleting buffer tuples that client applications can never use as some client application’s sessions end. Figure 22 shows how client applications share a sensor stream buffer. The grey arrow at the topmost right part of the figure shows the part of the buffer that the middleware deleted after the end of *C₁*’s session. The deleted portion of the buffer represents sensors data that arrived at the middleware between 10.00AM and 10.15. Although C_1's session lasts for an hour, buffer tuples for the first fifteen minutes are deleted because C_2 joined the session at 10.15, so C_2 uses the buffer tuples till 11.15AM. Furthermore, the blue arrow shows that no tuples would be deleted at the end of C_3's session because C_3's session started after C_1, C_2 and ended before C_1 and C_2 sessions ended. Moreover, the golden arrow points the portion of the buffer that is deleted at the end of C_2' session. C_2's session ended at 11.15 AM, yet C_4's session started at 11.00 and ended at 11.25, so the buffer content from 11.00 till 11.15 (e.g., C_2' end session time) cannot be deleted. However, as C_4 is the only client application that has an active session with sensor S, and C_4 cannot use buffer content that arrived before 11.00, that portion of the buffer is deleted at the end of C_2's session. Finally, at the end of C_4's session, the buffer is dropped as no other client application needs the buffer. The buffer is created again when a client application requests executing a query that requires buffering sensor data.

In our design, buffer tuples are deleted by the buffer manager, yet the portion of the buffer to be deleted is specified by the Data Transmitter that requests Buffer Manager to delete the tuples. At the end of a client application session, the Data Transmitter asks the buffer manager to delete part of the buffer that is no longer needed by the client application and other client applications cannot use.

It's important to note that most likely there might be tuples in the buffer that client applications cannot use. This happens when client applications use pricing policy that has a frequency less than the sensor frequency. For example, the sensor sends data every second, yet client applications want to receive the data every 2 or 4 seconds as shown in Figure 21. In this case, some tuples won't satisfy the subset extraction condition for any client application that does not use every second frequency. However, in our design, we opt to buffer all tuples without checking if they are used by active queries. The rationale for this decision is that the process of checking whether the arrived sensor message (e.g., tuple) satisfies the subset extraction condition of any client application has an active session before buffering the message is time and resource consuming if the number of client applications with active sessions is large. For this reason, we opt to buffer the all the message even though no client application might use it as the message would just be buffered temporarily.

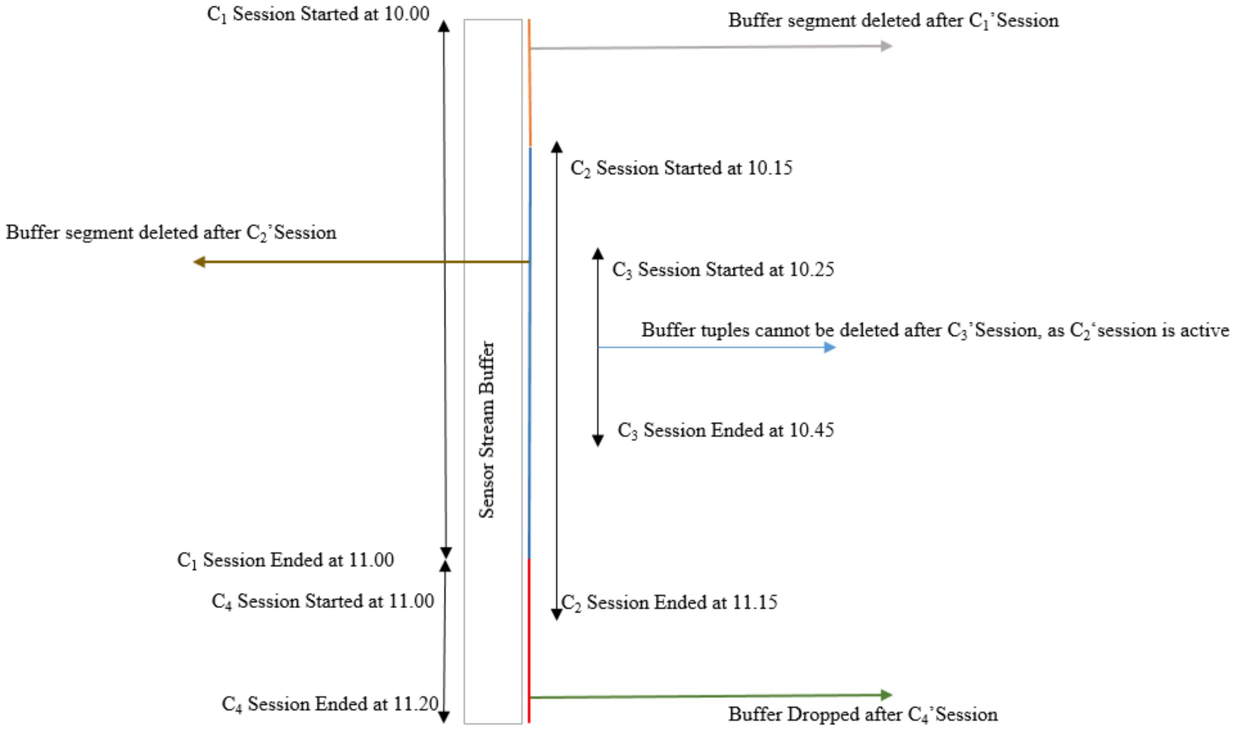

C_1 Session Started at 10.00
Buffer segment deleted after C_1'Session
C_2 Session Started at 10.15
Buffer segment deleted after C_2'Session
C_3 Session Started at 10.25
Buffer tuples cannot be deleted after C_3'Session, as C_2'session is active
C_3 Session Ended at 10.45
Sensor Stream Buffer
C_1 Session Ended at 11.00
C_4 Session Started at 11.00
C_2 Session Ended at 11.15
C_4 Session Ended at 11.20
Buffer Dropped after C_4'Session

Figure 22 Sensor Stream Buffer Management

4.5.3 Sensing Discovery Module

Sensing Discovery provides client applications with information about sensor locations, types, pricing policies, and data scheme. The Sensing Discovery Module receives client application sensing discovery queries through the service interface and then uses the Query Interpreter to parse the query. The result of Query Interpreter is used to execute the queries over sensing discovery information model (see section 3.2.1). After that, the Sensing Discovery Module passes the query result to the client application through the Service Interface.

4.5.4 Historical Session Handler

As mentioned previously, this module handles consumers' requests for sensors readings in the past. This module consists of two components. A cloud based data repository, and

Sensor Agents. Every sensor has a Sensor Agent that catches every message the sensor sends to the cloud-based publish/subscribe broker. Once the message is received, it is stored in the data repository. Now, we assume that sensors messages are heterogeneous. To illustrate, one sensor might just sense the temperature, so that its message structure looks like this "{temp:50}". Another sensor might sense two attributes (e.g., Temperature and humidity). In this case message structure looks like this "{temp:50, hum:80}". Despite message structure, all sensors messages are stored in the same repository. When messages are written to the repository, they are labeled with a sensor tag. Using this tag, the middleware relates messages to sensors. During the lookup process, the sensor tag is used to retrieve sensor messages over a defined period in the past. To apply a filter over the retrieved data, the historical Session Handler calls the analytical module which applies the filter and returns a subset of the retrieved data that passes the filter condition.

4.5.4.1 Historical Session Query Strategy

When a consumer initiates a historical session request, the Historical Session Module Handler handles it. The consumer specifies the sensor, the timeframe of the data and the filtering information as parameters through the client application or directly using the API from any custom application. The Historical module receives the consumer provided parameters and connects to the cloud data repository to retrieve the historical data. The Historical module sends the sensor tag and the timeframe parameters to the cloud data repository. Upon receiving the results from the repository, if the consumer did not specify a filter, the historical session handler which writes the results in a csv file and passes that file to the service interface to deliver it to the consumer. In case the consumer specified a filter, the Historical module looks up the sensor structure description, which the sensor owner enters at the registration process. Now, the Historical has the required knowledge to do filtering, as it knows the result set, the structure of the result set, and the filter condition. The Historical module goes over the result set row by row and evaluates the filter expression. Rows that pass the filter are sent back to the historical session handler which writes the results in a csv file and passes that file to the service interface to deliver it to the consumer.

4.5.5 Billing Module

The Billing Module is triggered at the end of a client application session, regardless if it is live or historical. The main task of this module is to calculate the client application session chargers based on the pricing policy used in the session. For live session charges, Billing Module has to know the pricing policy, session start time, session ends time and the user id. For a historical session, the Billing Module has to know the pricing policy and the number of processed sensor readings. After that, session charges are applied to the user account, and the sensor owner account. In case a client application requests a session that lasts for more than a week. The Billing module chargers the client application at the end of every week.

Chapter 5

5 Implementation

In this section, we describe the tools and technologies used to implement the Sensing as a Service middleware described in chapter 4.

5.1 Cloud Platform

The Sensing as a Service Middleware described in chapter 4 is built on top of Amazon Web Services platform (AWS) [55]. Mainly, the middleware relies on four AWS services which are AWS IoT [48], AWS EC2 [59], AWS RDS [54], and AWS DynamoDB [52]. The used AWS services are briefly described below.

- *AWS IoT:* The middleware uses AWS IoT to connect to client deployed sensors. In the Sensor Representation & Management Layer, the middleware connects to sensors through AWS IoT MQTT Broker [56] that can to connect to billions of IoT devices. The middleware uses Eclipse Paho library [60] to create MQTT clients, which are called sensor agents in the architecture.

- *AWS EC2:* Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides secure, and scalable virtual machines. Our middleware prototype is hosted on an EC2 instance.

- *AWS DynamoDB* is a fully managed NoSQL database offered by the AWS platform. The middleware uses DynamoDB as a Historical Data Repository. A copy of sensor data is sent to DynamoDB upon its arrival at the AWS IoT MQTT broker. When the middleware receives a client request for historical sensor data, the middleware queries the DynamoDB for the requested sensor data, then delivers DynamoDB query result to the requestor.

- *AWS RDS* is Amazons Relational Database service. The middleware uses an AWS RDS instance as a host for the middleware information model described in chapter3.

5.2 Middleware Prototype

The middleware prototype was developed using Java Spring Boot Framework [65]. Spring Boot is a popular Java framework for developing enterprise web applications. The prototype has a graphical user interface that consists of two major components: user management and sensor management. The user management component is used for user registration and management. The sensor management component is used by sensor owners to add their sensors to the middleware. Besides, the middleware prototype provides a RESTful API interface through which client application can access sensor data. The Restful API is documented using Swagger API documentation Framework 72]. Figure 23 depicts the mapping between the used technologies and the proposed architecture.

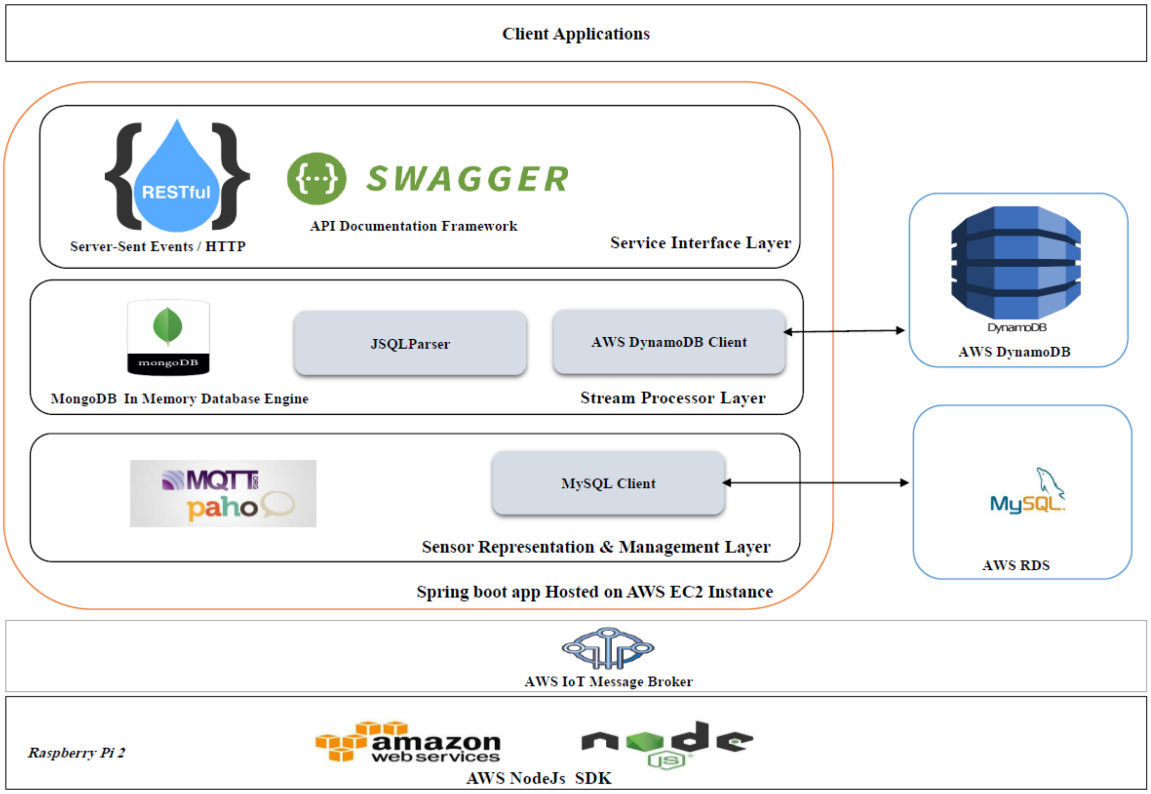


Figure 23 Mapping Middleware Implementation to the proposed Architecture.

5.3 Communication Protocols

The middleware prototype uses two different communication protocols which are MQTT protocol to communicate with sensors, and the Server-Sent Events protocol to push sensor data to client applications.

5.3.1 MQTT

MQTT stands for (Message Queue Telemetry Transport). MQTT is an Internet of Things communication protocol that is designed to be an extremely lightweight publish-subscribe messaging transport for lower powered devices [64]. The middleware prototype uses MQTT through Eclipse Paho Java Client [60].

5.3.2 Server-Sent Events

The middleware uses Server-Sent Events (SSE) to push sensor data to client applications, upon sensor data arrival, in a form of server notification to a client application. The SSE is a unidirectional communication protocol that allows the server to push data to a client. SSE is built on top of HTTP [69].

5.4 Sensor Gateway

The Sensor Gateway is a low powered computer that is used to connect sensors to the middleware. In our prototype, sensors are attached to a Raspberry Pi 2 [67] with Quad-Core 900 MHz CPU and 1GB RAM. The Raspberry Pi uses an operating system called Raspbian, a Debian-based computer operating system. Raspbian is highly optimized for the Raspberry Pi line's low-performance ARM CPUs [66]. In our implementation, sensors are attached to the Raspberry Pi through the GPIO Pins [68].

5.5 Data Dispatcher

Data Dispatcher is a Node JS application that runs on the Raspberry Pi. The Data Dispatcher reads sensor data through the Raspberry PI GPIO Pins and sends sensor data to the AWS MQTT Broker that delivers sensor data to the middleware. The Data Dispatcher uses the AWS IoT SDK to communicate with the AWS MQTT Broker.

5.6 Query Parser

The middleware Query Parser is built on top of an open source tool called JSqlParser [62]. JSqlParser translates SQL statements to a traversable hierarchy of Java classes.

5.7 In-Memory Database engine

The middleware uses MongoDB as an In-Memory repository to store sensor data for window operation. MongoDB has a component called storage manager responsible for how data is stored. As our prototype aim is to perform stream processing, we used MongoDB In-Memory storage engine to speed up stream processing tasks. MongoDB In-Memory Storage engine is available on MongoDB Enterprise edition [63].

Chapter 6

6 Experimental Design and Results

This chapter describes the performance of the Sensing as a Service middleware described in chapter 4. The goal of the evaluation is to observe the middleware's performance when it's flooded with client application requests for sensor data. Section 6.1 describes the experimental environment, section 6.2 describes the experimental parameters and scenarios, section 6.3 describes the evaluation metrics, and section 6.4 describes the experimental results of the proposed Sensing as a Service Middleware in terms of response time, memory consumption, and CPU utilization.

6.1 Experimental setup

This section describes the experimental setup.

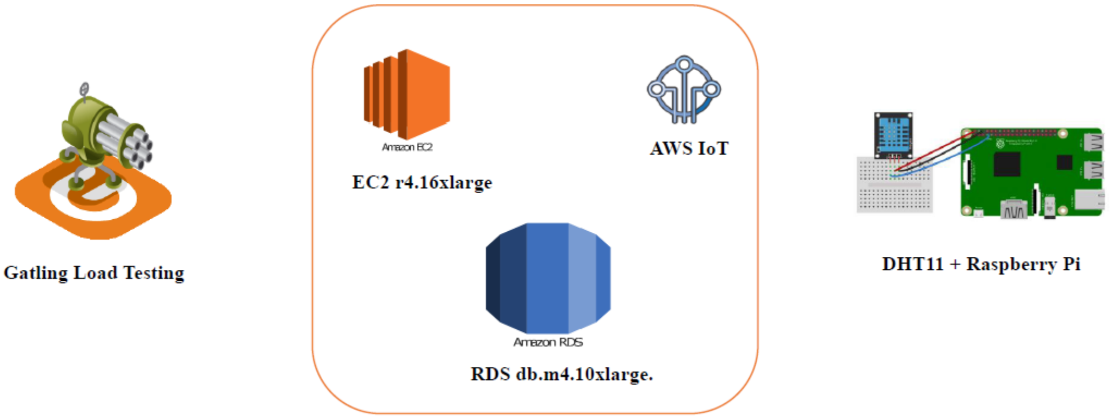


Figure 24 Experimental Setup.

6.1.1 Cloud Deployment

We deployed the proposed middleware in an AWS EC2 instance that is designed to support applications that perform heavy in-memory processing [59]. The AWS instance type used is r4.16xlarge. This instance has a 488 GB Memory and 32 physical CPUs that provide a total of 64 cores. The instance uses the Amazon Linux operating system. The instance version of the operating system is amzn-ami-hvm-2017.03.1.20170623-x86_64-gp2 (ami-6df1e514). In addition, we deployed the middleware database in an AWS RDS (Amazon

Relational Database Service) instance [54]. The type of the RDS instance is db.m4.10xlarge. The middleware instance and the database instance are hosted in Amazon's *us-west-2a* region (e.g., Oregon, US) [57].

6.1.2 Sensor Setup

The sensor used in the experiments is the DHT11 Humiture sensor [64] that senses temperature and humidity. The sensor is attached to a Raspberry Pi [67] computer where the Data Dispatcher service is deployed. The Data Dispatcher is an application written in NodeJS using the AWS IoT developer SDK [58]. The Data Dispatcher reads sensor observations and sends these observations to the AWS IoT Publish-Subscribe broker [56]. The broker is hosted in amazon's *us-west-2a* region (e.g., Oregon, US) [57].

6.1.3 Stress Testing Tool

The Gatling Load Testing [61] is the load generation tool used in the experiments. Gatling is an open-source load and performance testing framework based on Scala. We used Gatling to generate virtual users that flood the middleware with client requests. Gatling testing scenarios are written in Scala.

6.2 Experimental Scenarios and Parameters

We used three scenarios in which virtual client applications, simulated by the load testing framework, send requests to the middleware. The requests are queries in which client applications demand live sensor data for a future period of time. In each testing scenario, we ran eight experiments, so the total number of experiments is 24. In the first scenario, the users send Raw Data queries. In the second scenario, the users send Time-based Tumbling Window queries, and the middleware uses the single buffer algorithm to store sensor data for window operations. In the third scenario, the users send Time-based Tumbling Window queries and the middleware creates a buffer for each client query in order to store sensor data for window operations. The queries used in the scenarios use the following format:

- *First Scenario:* In this scenario, clients request to receive raw sensor data for the next five minutes (e.g., 300 seconds) using a pricing policy which states that a client should receive a message every second. The query for this scenario takes the following format.

Select temp, hum from DHT11 Using 1 For 300;

- *Second and Third Scenarios:* In these scenarios, client applications request that the middleware buffer sensor data for 30 seconds then executes the aggregation functions over the buffered data. The query for those scenarios takes the following format:

  Select avg(temp), avg(hum), max(temp), max(hum) from DHT11 *Window 30 Using 1 For 300;*

For each scenario, the experimental parameters include the length of query session (e.g., for how long each query runs), the number of client applications, buffer management algorithm (for second and third scenarios). We set the experimental parameters as follows: For all testing scenarios, the query session length was set to 5 minutes, the number of sensors was one and the number of client applications took the following values: 1, 10, 50, 100, 250, 500, 750, and 1000.

It's important to note that we configured the load testing framework not to send the number of requests at once to avoid Amazon's firewall request rejection. Instead, we set the load testing framework to send the request over a period of time (i.e., 180 seconds for 1000 request). In addition, the reason we limit the number of client application requests to 1000 is that the Tomcat server [71] that runs the middleware kept crashing whenever the number of the sent requests exceeded 1500 requests which represent the maximum number of requests that can be received by the Tomcat server. Although this is configurable, we do not have the permissions to do so.

6.3 Evaluation Metrics

We used three metrics to evaluate the middleware performance in our experiments. Our evaluation metrics include Response Time, Memory Consumption, and CPU utilization.

Per client request response time: The response time represents the period of time that a client application needs to wait to receive the submitted query results. For example, in the first scenario, a client application sends a query to request raw sensor data for the next 5 minutes using a pricing policy that has a frequency of one second, so the middleware needs to push the sensor data to the client application every second in the upcoming five minutes. The response time represents how long it took the client to receive the requested number of sensor messages. We measure the response time as the difference between the request issuing time (t_{start}) and the time when the client has received all query results (t_{end}).

$$\text{responseTime} = t_{end} - t_{start}$$

Memory consumption: Memory consumption represents the amount of memory used by the middleware during an experiment. For each experiment, we monitor memory consumption values every second for the duration of the experiment. The collected values are Total memory assigned by the Java Virtual Machine (JVM) to the middleware, the free and, the used portions of the assigned memory. In addition, for the second and third scenarios, we measure the size of the sensor buffer.

CPU utilization: For each experiment, the CPU utilization represents the maximum CPU usage value for the duration of the experiment. Those values are collected using the AWS Cloud watch service [51] that monitors the AWS EC2 instance while it’s running.

6.4 Results

In this section, we present the results of the middleware performance in the three testing scenarios. Moreover, we compare the middleware performance when using the single buffer vs multiple buffers. Section 6.4.1 presents the First Testing scenario results. Section 6.4.2 presents the second testing scenario results. Section 6.4.3 presents the third testing scenario results. In section 6.5, we discuss the evaluation results.

6.4.1 First Scenario

In this section, we present the response time results for the first scenario in which client applications send Raw Data Queries to the middleware.

6.4.1.1 Response Time

Table 1 presents the response time for the first scenario’s experiments. The table shows that the number of client applications significantly affects the response time. For example, in the 1000 client application experiment, the delay reaches 108 seconds which means that there was a client application that had to wait 6 minutes and 48 seconds to receive all query results. However, the minimum response time has been slightly affected by the number of client applications. Moreover, the table shows that the standarad deviation increases as the number of client application increases. The increase in standard deviation indicates the wide range of response time values which means that not all client applications experienced significant latency. The rationale for this behavior is attributed to the Data Transmitter design. Our design aims to reduce the network traffic by minimizing the number of connections between the middleware and the AWS publish-subscribe broker, so the Data Transmitter connects to the middleware on behalf of the client applications and whenever the Data Transmitter receivesa sensor data tuple, the Data Transmitter goes over a list of client application (Query Registry) to deliver the sensor tuple to each client on the list. It's obvious that client applications at the end of the list experienced a significant delay in response time. However, the results show that our Data Transmitter design provided consistent performance up until 500 client applications.

Table 1 First Scenario Response Time.

Number Of Client Applications	Min	Max	Mean	Std Deviation
1	301198 ms	301198 ms	301198 ms	0 ms
10	299706 ms	301015 ms	300317 ms	441 ms
50	299735 ms	301291 ms	300731 ms	541 ms
100	299697 ms	301054 ms	300132 ms	295 ms
250	299537 ms	310821 ms	302857 ms	2983 ms
500	298415 ms	306126 ms	301944 ms	2352 ms
750	300528 ms	342542 ms	346194 ms	10292 ms
1000	303655 ms	408517 ms	346194 ms	20922 ms

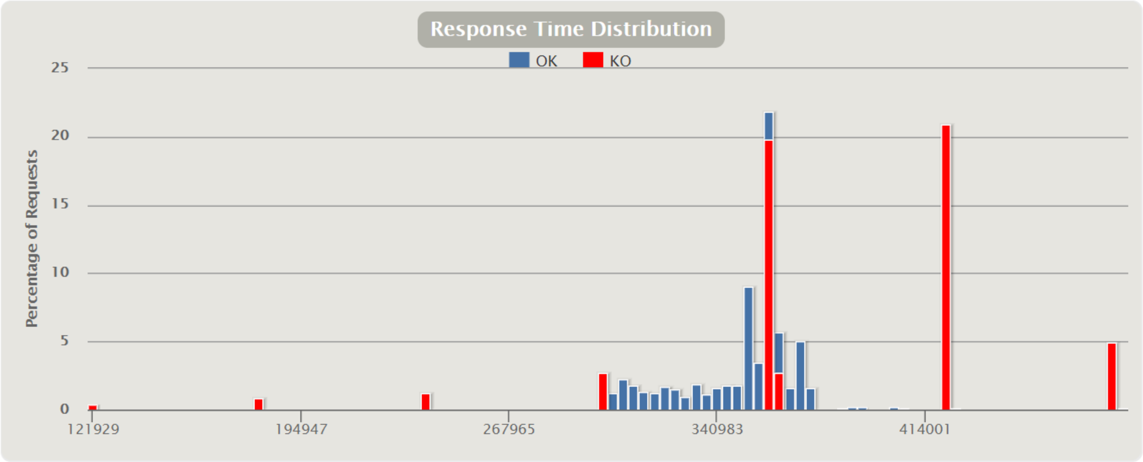


Figure 25 First Scenario Response Time Distribution For the duration of 1000 client application expriement.

Figure 25 depicts the response time distribution for the 1000 client application experiment. The figure shows that response time was 359238 ms for 21.83 % of the requests. Moreover, the figure shows that around 55 % of the requests failed to receive complete query results by the time the Load Testing Framework closed the connections with the middleware. The load testing framework was configured to wait at most 420 seconds to receive all messages. When the request time passes 420 seconds (e.g., 7 Minutes), Gatling considered the request as a failed request and closed the connection with the middleware.

6.4.1.2 Memory Consumption

In this section, we present the memory consumption results for the first scenario experiments. Table 2 presents the values of the minimum, maximum, median, average and the standard deviation of the used memory for each experiment. The values are collected every second throughout an experiment. The table shows that the middleware memory consumption rate increases as the number of client application requests increase.

Table 2 Memory Usage for the first scenario experiments

Number of Client Applications	Min	Max	Median	Average	Std Deviation
1	194 MB	863 MB	519 MB	522.65 MB	191.20 MB
10	122 MB	1089 MB	384 MB	433 MB	273 MB
50	79 MB	1202 MB	726 MB	704 MB	274 MB
100	77 MB	1215 MB	679 MB	645 MB	289 MB
250	103 MB	2969 MB	789 MB	983 MB	671 MB
500	79 MB	10401 MB	3524 MB	4001 MB	2993 MB
750	78 MB	10410 MB	2014 MB	3111 MB	2693 MB
1000	89 MB	10523 MB	3496 MB	4149 MB	2954 MB

In addition, Figure 26 shows the cumulative distribution function for the total allocated memory, free memory and the used memory for the duration of the 1000 client application request experiment. The figure shows that for 50% of the experiment time, the used memory was around 5000 MB.

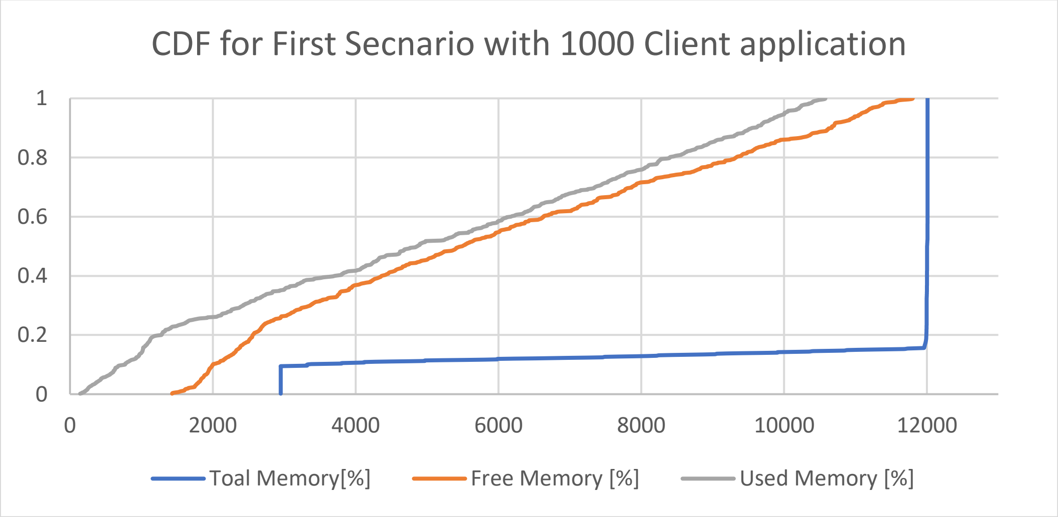


Figure 26 Cumulative Distribution Function for memory consumption during 1000 client application request for the first scenario.

6.4.1.3 CPU Utilization

In this section, we present the CPU usage for the first scenario experiments. Figure 27 depicts the maximum CPU usage for each experiment. The figure shows that the CPU usage sharply increased as the number of client application goes over 250.

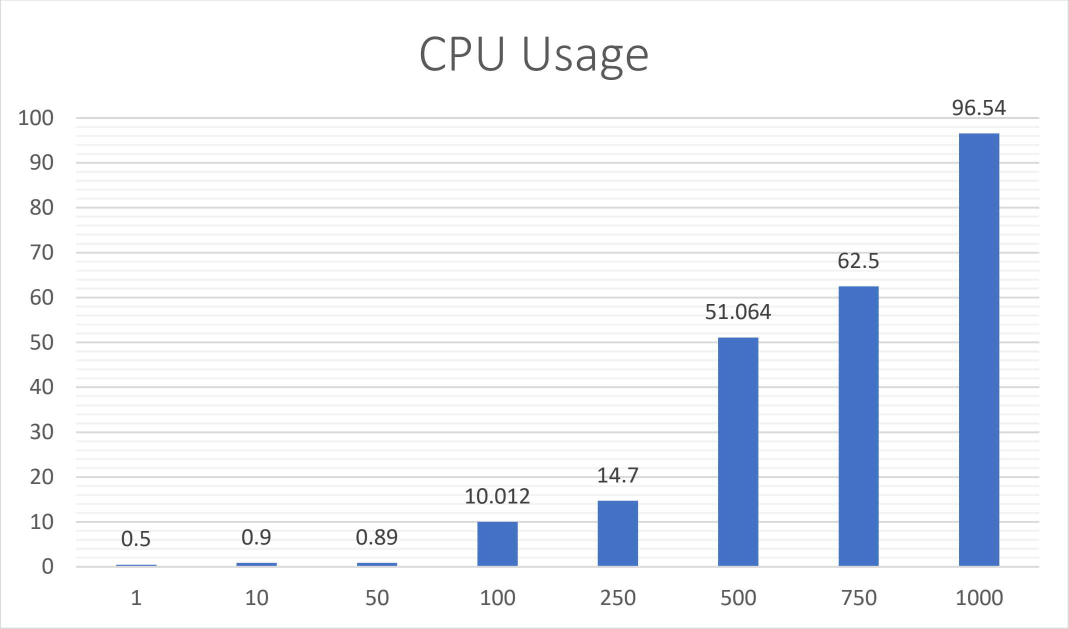


Figure 27 First Scenario Experiments CPU usage.

6.4.2 Second Testing Scenario

In this section, we present the response time results for the second testing scenario in which client applications submit Time-Based Tumbling Window Queries and the middleware uses a single buffer to store sensor data for window operations.

6.4.2.1 Response Time

Table 3 presents the values of minimum, maximum, mean, and the standard deviation of response times for each experiment. The values are represented in milliseconds.

Table 3 shows that the response time in the second scenario’s experiments has not been affected by the number of client application as the response time will be at most increased by 3 seconds. This is because of the reduced message delivery overhead.

Table 3 Second Scenario Response Time.

Number Of Client Applications	Min	Max	Mean	Std Deviation
1	301628 ms	301628 ms	301628 ms	0 ms
10	300215 ms	301599 ms	300676 ms	499 ms
50	300154 ms	301579 ms	300642 ms	252 ms
100	300240 ms	301655 ms	300637 ms	276 ms
250	300034 ms	302047 ms	300562 ms	347 ms
500	300011 ms	302817 ms	300636 ms	395 ms
750	300047 ms	302716 ms	300597 ms	356 ms
1000	300405 ms	303465 ms	300634 ms	423 ms

To illustrate, when executing Time-based Tumbling Window queries using the single buffer algorithm, the Data Transmitter does not deliver query results to client applications as this task is assigned to the query workers. Whenever the Data Transmitter receives a sensor data tuple, the Data Transmitter stores these tuples in the sensor buffer. The query workers concurrently read from the buffer using the buffer subset extraction algorithm and deliver the window result to client applications. The response time distribution for the 1000 client application requests is graphically depicted in Figure 28.

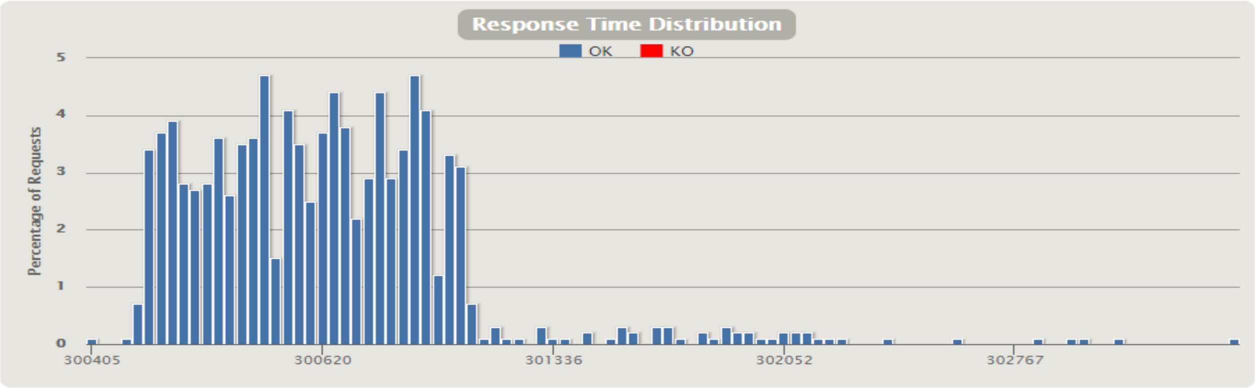


Figure 28 Response Time Distribution for the second Scenario with 1000 client Applications.

6.4.2.2 Memory Consumption

In this section, we present the memory consumption results for the second scenario. Table 4 presents the values of the minimum, maximum, median, average and the standard deviation of the middleware memory consumption during each experiment. The values are collected every second throughout each experiment.

Table 4 Memory Usage for the second scenario experiments

Number of Client Applications	Min	Max	Median	Average	Std Deviation
1	128 MB	1049 MB	489 MB	645 MB	349 MB
10	85 MB	1052 MB	627 MB	632 MB	236 MB
50	79 MB	1202 MB	726 MB	704 MB	274 MB
100	77 MB	1215 MB	679 MB	645 MB	289 MB
250	103 MB	2969 MB	789 MB	983 MB	671 MB
500	79 MB	10401 MB	3524 MB	4001 MB	2993 MB
750	78 MB	10410 MB	2014 MB	3111 MB	2693 MB
1000	89 MB	10523 MB	3496 MB	4149 MB	2954 MB

It’s obvious that the middleware uses more memory as the number of client applications increases. However, every time the garbage collection is triggered, it sharply decreases the amount of used memory. Figure 29 graphically depicts the memory usage for the duration of the 1000 client applications experiment. The figure shows how the garbage collection actively decreases the amount of used memory. However, as the server has 488 GB RAM, the Java Virtual Machine can allocate more RAM to the application. Figure 30 shows the cumulative distribution function for the total allocated memory, free memory and the used memory for the duration of the 1000 client application request experiment. The figure shows that for 50% of the duration of the experiment, the used memory was around 4825 MB.

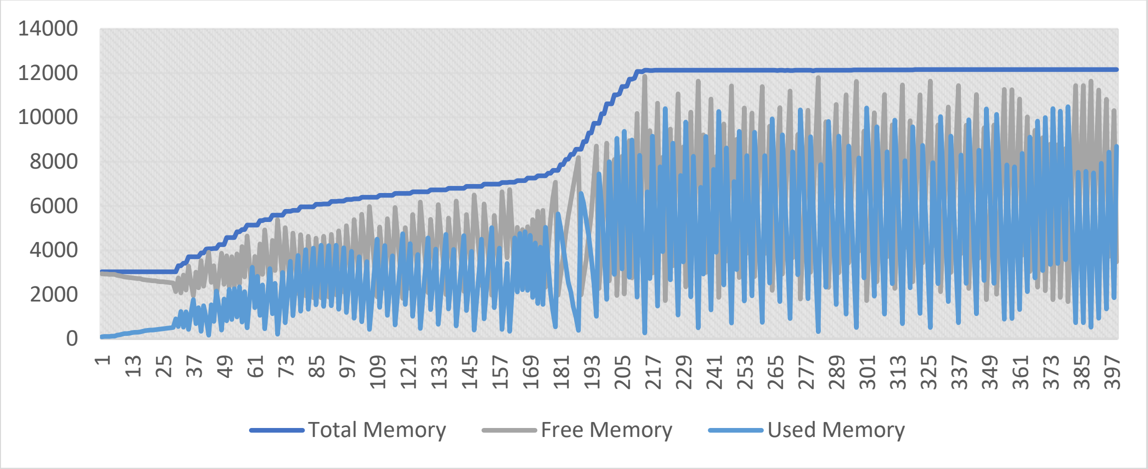


Figure 29 Memory Consumption.

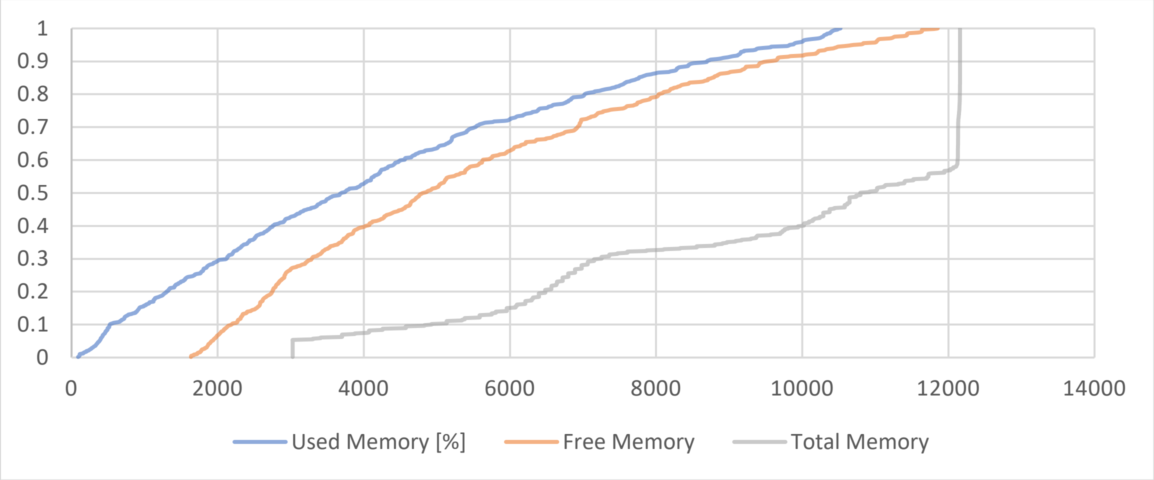


Figure 30 CDF for the duration of the 1000 client application request in the Second Scenario.

In addition, Table 5 shows that the amount of memory used for the sensor data buffer remained the same for all experiments because the middleware used a single buffer to store sensor data.

Table 5 Buffer size information during 1000 client application experiment.

Sensor Data Buffer	
Minimum	0
Maximum	41.28 KB
Average	28.83 KB
Median	35.068KB
Standard Deviation	13.94 KB

6.4.2.3 CPU Utilization

In this section, we present the CPU utilization for the second scenario experiments. Figure 3 depicts the maximum CPU usage for each experiment. The figure shows that the CPU usage increases as the number of client applications increase. However, the maximum value for the middleware CPU usage was just 35.77% during the 1000 client application experiment. In comparison to other scenarios, the reduced processing assigned to the Data Transmitter resulted in less CPU usage.

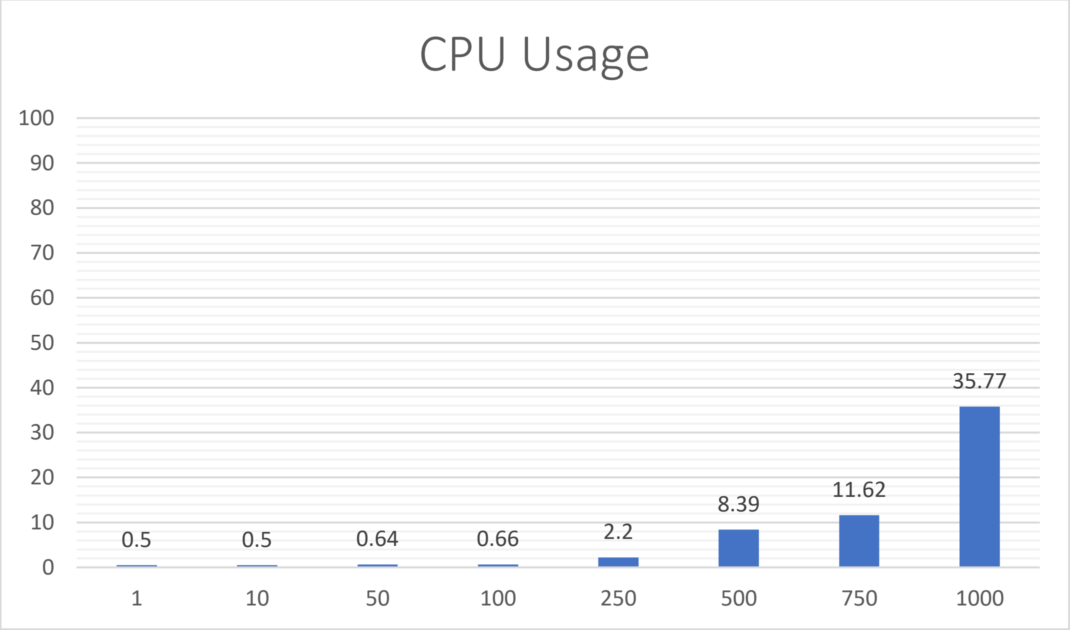


Figure 31 CUP usage in second scenario experiments.

6.4.3 Third Testing Scenario

In this section, we present the response time results for the third testing scenario in which client applications submit Time-Based Tumbling Window Queries and the middleware creates a buffer for each client application request to store sensor data for window operations.

6.4.3.1 Response Time

Table 6 presents the values of minimum, maximum, mean, and the standard deviation of response times for each experiment. The values are represented in milliseconds. Table 6 shows that the response time is affected by the number of client applications. The table shows the there was a delay of 97 seconds in the 1000 client application requests experiment. Unlike the second testing in which the Data Transmitter stores the sensor data in a single buffer, in the third testing scenario, every client application has a buffer. Whenever the Data Transmitter receives a sensor data tuple, the Data Transmitter needs to go over the Query Registry and store the received tuple in each client application buffer. Consequently, client applications at the end of the list experience an increased delay as the number of client application goes over 250.

Table 6 Third Scenario Response Time.

Number Of Client Applications	Min	Max	Mean	Std Deviation
1	301602 ms	301602 ms	301602 ms	0 ms
10	300733 ms	301599 ms	300833 ms	256 ms
50	300087 ms	301562 ms	300659 ms	325 ms
100	300774 ms	335173 ms	310771 ms	12752 ms
250	298566 ms	302083 ms	300736 ms	423 ms
500	300256 ms	311070 ms	305777 ms	2905 ms
750	300157 ms	313238 ms	307641 ms	3244 ms
1000	308039 ms	397429 ms	348350 ms	21383 ms

The response time distribution for the 1000 client application experiment is graphically depicted in Figure 32. The figure shows that 3.76 % of client application waited 361226 ms to receive the results of their submitted queries. Overall, the figure shows how the number of client applications affected the response time as the minimum response time in the experiment was 308486 ms (8 seconds).

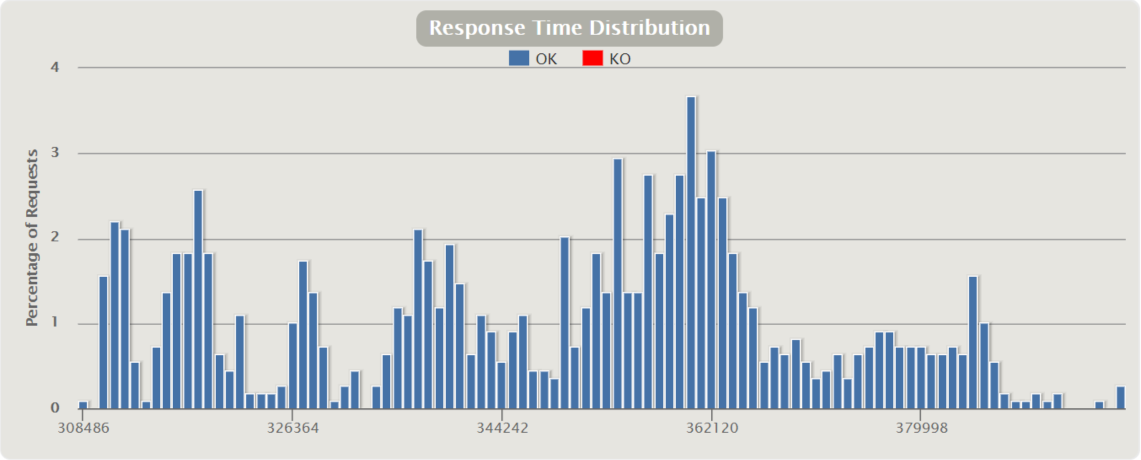


Figure 32 Response Time distribution for the Third Scenario with 1000 client applications.

6.4.3.2 Memory Consumption

In this section, we present the memory consumption results for the third scenario. Table 7 presents the values of the minimum, maximum, median, average and the standard deviation of the middleware memory consumption during each experiment. The values are collected on every second throughout each experiment. The table shows that the middleware memory consumption rate increases as the number of client application requests increase.

Figure 33 shows the cumulative distribution function for the total allocated memory, free memory and the used memory for the duration of the 1000 client application request experiment. In addition, the amount of memory used for the sensor data buffer increases with the number of client applications. Table 8 shows the size of 1000 client application buffer for the 1000 client application experiment. A comparison of Table 5 and Table 8 shows how the single buffer approach significantly decreases the size of sensor data buffer.

Table 7 Memory Usage for the third scenario experiments.

Number of Client Applications	Min	Max	Median	Average	Std Deviation
1	90 MB	828 MB	506 MB	503.71 MB	192.53 MB
10	104 MB	1038 MB	605 MB	593.06MB	230.12 MB
50	75 MB	1996 MB	759 MB	811.60 MB	506.10 MB
100	89 MB	9233 MB	2106 MB	2409.51 MB	1577.58 MB
250	79 MB	5887 MB	1893 MB	2060 MB	1460 MB
500	78 MB	10385 MB	4370.5 MB	4612.19 MB	3110.69 MB
750	89 MB	10512 MB	4545 MB	4730.48 MB	3165.57 MB
1000	79 MB	10487 MB	5317 MB	5195 MB	2907 MB

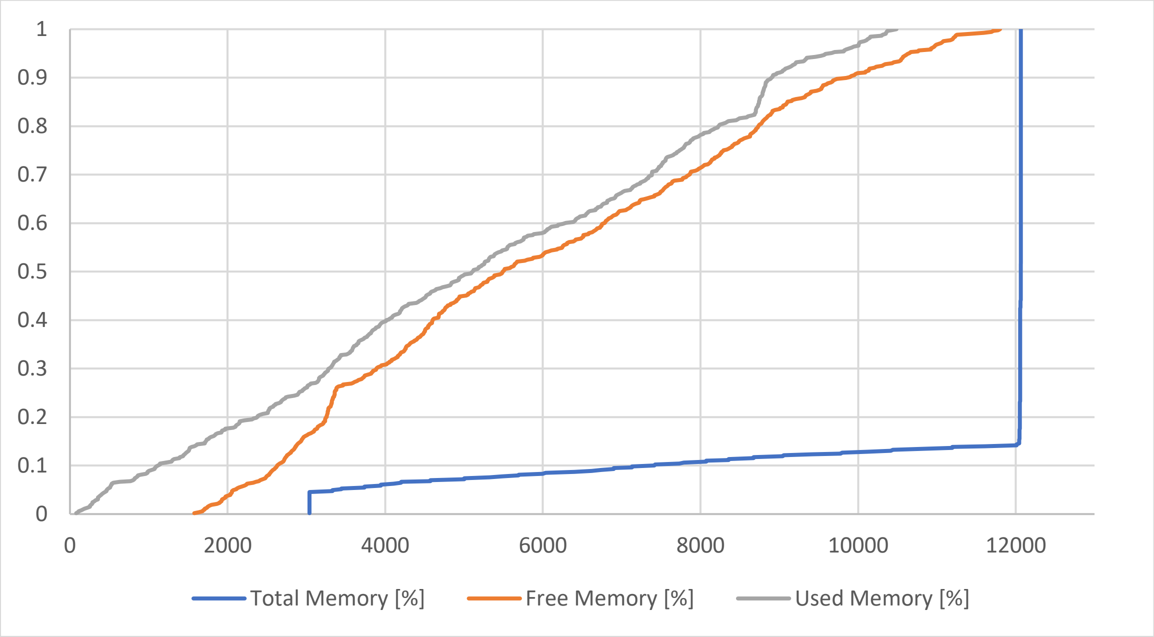


Figure 33 Cumulative Distribution Function for memory consumption during 1000 client application request for the third scenario.

Table 8 Sensor Buffer Size throughout the 1000 client application experiment in the third scenario.

Sensor Data Buffer	
Minimum	0
Maximum	6063.88 KB
Average	2885.56 KB
Median	3476.14 KB
Standard Deviation	1675.57 KB

6.4.3.3 CPU utilization

In this section, we present the CPU usage for the first scenario experiments. Figure 34 depicts the maximum CPU usage for each experiment. The figure shows that the CPU usage increases as the number of client applications increase. The figure shows that the CPU usage is sharply increased as the number of client application goes over 250.

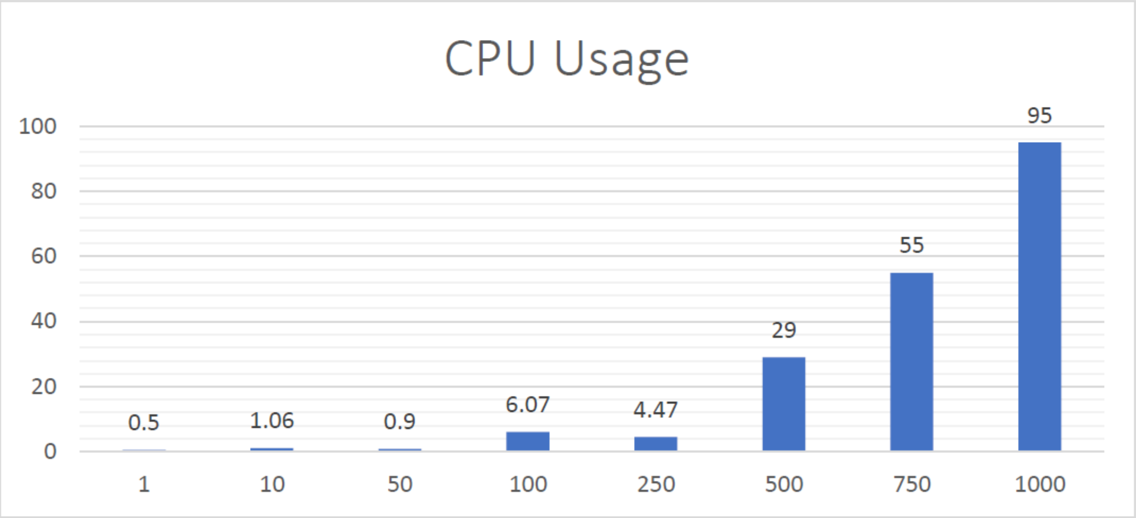


Figure 34 Third Scenario Experiments CPU usage

6.5 Experimental Discussion

We observe that the second testing scenario experiments provided the best results in terms of response time, memory consumption, and CPU usage as the number of client applications increases. In this section, we discuss the reasons for different middleware behavior in the three scenarios.

First, the second scenario experiments provided the lowest response time as the number of client applications increased. On the other hand, the response time for the first and third scenarios sharply increases when the number of client applications passes 250 client applications. This behavior is mostly attributed to the Data Transmitter design. In the second testing scenario, the Data Transmitter is assigned a simple task in which the Data Transmitter stores sensor messages in a single sensor buffer. On the contrary, more processing is assigned to the Data Transmitter in the first and third testing scenarios. In the first testing scenario, the Data Transmitter delivers each sensor message to all client applications using the Server-Sent Events protocol (SSE) [73] which increases response time for client applications at the end of the Query Registry as the Data Transmitter has to push the message to several SSE channels. In the third testing scenario, the Data Transmitter stores each sensor message in every client application buffer, yet the query result delivery is assigned to query workers. As the number of client applications increases, the Data Transmitter needs more time to store a sensor message in all client application buffers which results in a sharp increase in response time. To illustrate, a query worker is a thread that executes the aggregation functions at the end of each window (e.g., every 30 seconds), delivers the window results to the client application over the SSE and then sleeps for the window size (30 seconds). When the Query Worker wakes in order to execute the query over a client application sensor buffer, the Query Worker checks if the buffer has the required number of tuples to execute the query. If the buffer does not have the required number of tuples, the query worker waits until the number of tuples reaches the required number tuples to execute the query. In our experiments, the buffer should have 30 tuples for every window execution. This is because client applications used a window of size 30 seconds and a pricing policy which allows client applications to receive sensor data every second. A delay in delivering sensor tuples to client application buffers results in longer waits for the buffer size to reach the required number of window tuples which increases the response time for client applications.

Second, the three testing scenarios show a similar behavior for memory consumption. The used memory is increased with the number of client applications. Table 2, Table 4, and Table 7 show that middleware memory consumption was almost the same in the first and second scenarios while the third scenario experiments consumed slightly more memory.

Overall, the middleware, on average, used around 4500 MB during the 1000 client application experiment in the three scenarios. It’s important to note that memory allocation and management is controlled by the Java Virtual Machine (JVM) and the frequency of memory freeing depends on the size of available RAM. To illustrate, the EC2 instance that hosts the middleware has 488 GB RAM. Table 2, Table 4, and Table 7 show that the average of memory usage for 500 client application experiments in the three testing scenarios was around 4000 MB. We ran the experiment of 500 client applications on a computer that has 16 GB RAM, and the used memory never passed 1500 MB as shown in Figure 35.

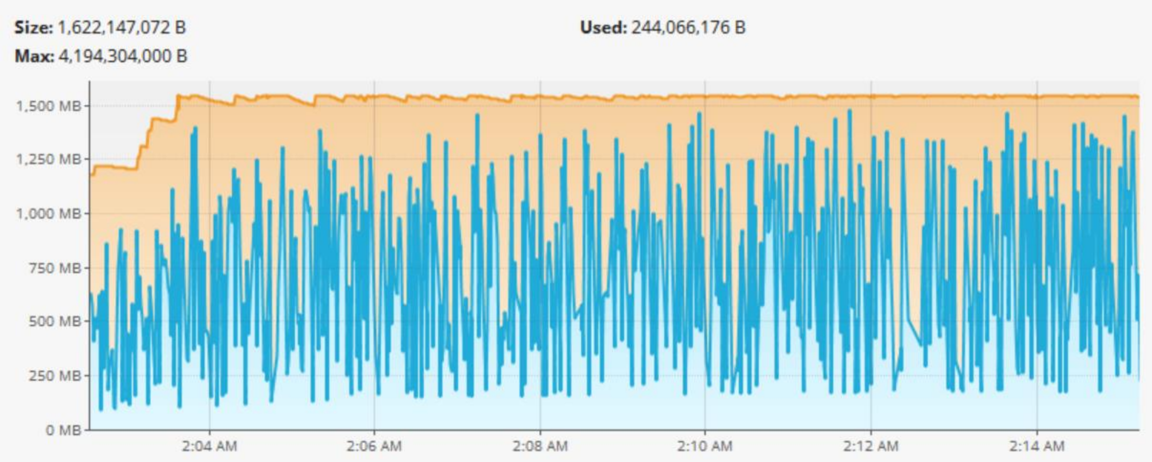


Figure 35 Memory Usage for 500 client application experiment on a machine that has 16 GB RAM.

In addition, the second testing scenario shows that buffer management significantly decreases the memory used for sensor data buffer. Figure 36 shows the cumulative distribution function graph for the in-memory storage size used by the second and the third testing scenarios in the 1000 client application experiments. The figure shows that for the duration of the two experiments, 50% of the storage size measurements were around 33 KB for the second testing scenario (e.g., single buffer scenario) whereas 50% of the storage size measurements was around 3500 KB for the third testing scenario (e.g., buffer per client scenario).

Finally, Figure 27, Figure 31, and Figure 34 show that the second testing scenario CPU usage was significantly lower than the first and the third testing scenarios in all experiments which might be a result of less processing tasks assigned to the Data Transmitter in the second testing scenario.

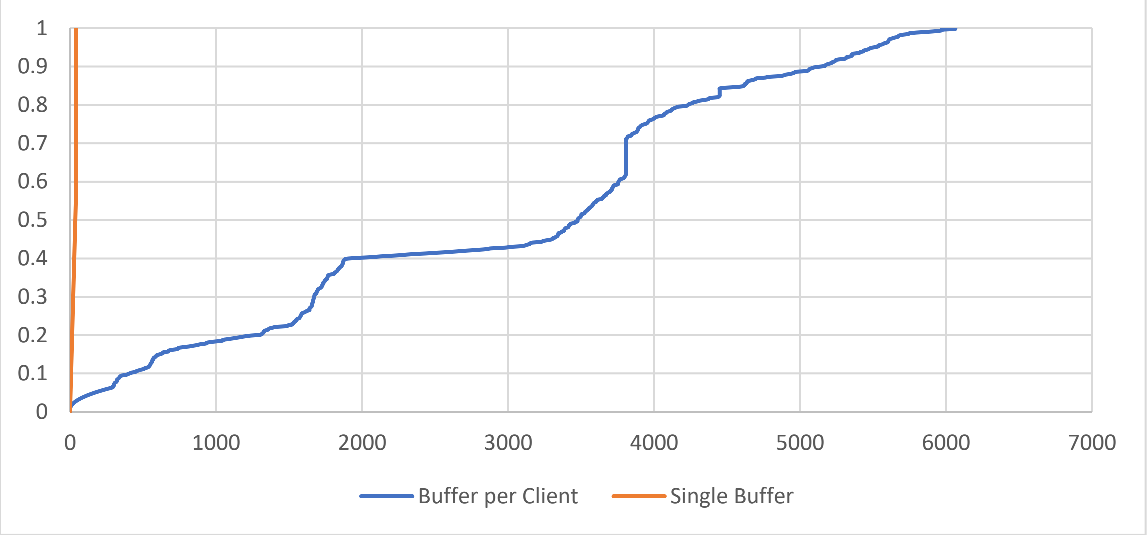


Figure 36 cumulative distribution function graph for the in-memory storage size in the second and third testing scenarios.

Chapter 7

7 Conclusion and Future Work

7.1 Conclusion

Recent advancement in information technology has increased the number of devices that are connected to the Internet which has resulted in the emerging phenomenon known as the Internet of Things (IoT). However, the problem with IoT devices is that they are heterogeneous and owned by different organizations and individuals which makes IoT data sharing a real challenge. This thesis addresses the architectural design and implementation of sensing as a service middleware for the Internet of Things.

In this thesis, we proposed and implemented a cloud-based sensing as a service middleware that enables sensor data sharing for IoT applications. Our middleware decouples IoT applications from the underlying IoT infrastructure. The middleware provides an abstraction layer which enables developers to access sensors, owned by other entities, over the Internet using a SQL-like query language that supports data filtering and aggregation operation over sensor data streams using continuous query semantic. Client applications are charged for the amount of sensor data they consume using a pay-as-you-go pricing model specified by sensor owners. In addition, we proposed multitenancy algorithms to reduce network traffic and cloud resource consumption. More specifically, we proposed buffer management techniques to reduce the amount of RAM used for sensor stream processing operations using algorithms proposed in Chapter 4. Furthermore, we proposed an algorithm to minimize the number of connections between the middleware and the publish-subscribe broker using the Data Transmitter algorithm in Chapter 4.

7.2 Future Work

In this thesis, we built a proof-of-concept Sensing as a Service middleware for the Internet of Things. Although our implementation performed well in a real-world deployment, there are several challenges that need to be addressed to improve the middleware performance and support more features.

First, the experiments show that the Data Transmitter design requires more development to provide better response time. The experiments show that response time is increased is increased when the number of client applications, requesting the same sensor data, is increased. Our Data Transmitter design minimizes the number of connections between the middleware and the publish-subscribe Broker by not opening more than a single connection for each sensor despite the number of client applications requesting the sensor data. The experiments show that this design becomes unfeasible when the number of client applications goes over 250 clients. With that being said, it might be useful if the middleware creates multiple Data Transmitters per sensor and each Data Transmitter opens a single connection with the Publish-Subscribe broker and serves a maximum of 250 client applications.

Second, subset extraction algorithm proposed in chapter 4 needs more development to deal with sensor messages delay. The algorithm assumes that sensor messages arrive in the exact frequency that sensor owner specifies in sensor registration. If a sensor message experienced a delay, the algorithm would not be able to recognize it. As a result, client application window subset might have tuples less than its number of required tuples. For instance, the algorithm might retrieve four tuples while the window subset is supposed to be five tuples. Although we did not encounter this problem during the experiments, it’s likely to happen if sensor messages experienced an unexpected delay.

Third, despite the middleware prototype ability to connect to billions of sensors through the cloud-based Publish-Subscribe broker, the middleware prototype cannot serve a high number of client applications, and the middleware is susceptible to a single point of failure in case the middleware crashed because of a high number of client application requests. There are many solutions to this problem. For example, it’s possible to run to middleware in several servers and use load balancing techniques to keep the middleware instances running. Another solution might be to develop a distributed version of the middleware in which the tasks assigned to the middleware are assigned to a set of nodes that work together in a collaborative fashion. Medusa [46], Borealis [2], and Stream Cloud [22] are good examples of such distributed design as they provide fallback mechanisms that help the middleware to cope with spikes in workload.

Fourth, the sensing discovery module requires more development to support sensing discovery queries described in chapter 3. Currently, our implementation does not provide full support to search the information model proposed in chapter 3. As a proof of concept, we limit our implementation to list out sensor deployed in a given city. Moreover, sensor search techniques proposed in CASSARAM [33] might be considered to improve sensor search operations when the number of sensors is large. Furthermore, the sensing discovery module should be deployed on a separate server to reduce the workload as Sensing discovery operations and stream processing operation are completely separate operations.

Fifth, the stream processor, and query parser components require more development to support query operations such as Join and Merge. Currently, a client application submits a select statement to query a single sensor. As the Join and Merge operations are supported by the in-memory stream buffer repository (e.g., MongoDB), the middleware can support those operations with more development on the stream processor and query parser components.

Sixth, the middleware needs to adopt fault tolerance techniques to deal with situations when sensor connectivity is lost. This can be done by either looking for another sensor in the same area to carry out the sensing operations. Another technique might be to use machine learning techniques to predict sensor readings based on the historical sensor data.

Finally, cloud platforms have powerful stream analytics services that can be used by the stream processor component to support complex stream analytics operations. Services such as AWS Kinesis [53], and IBM Watson [74] have great potential to extend the middleware stream processing operations if they have APIs through which the stream processor can direct the sensor stream to the analytics service and instructs the analytics service to carry out the client application request. Then, when the analytical service finishes the work, the result can be delivered to the client application via the middleware stream processor component. We tried to use AWS kinesis through the stream processor component. However, we found that AWS kinesis does not have a programmable interface, so stream analytics operations have to be configured manually, and thus the service cannot be used to carry out client applications stream analytics tasks. Nevertheless, when Amazon builds an API for Kinesis, the service can significantly improve the middleware analytics capabilities.

8 References

1. Abadi, Daniel J., et al. "Aurora: a new model and architecture for data stream management." The VLDB Journal—The International Journal on Very Large Data Bases 12.2 (2003): 120-139.
2. Abadi, Daniel J., et al. "The Design of the Borealis Stream Processing Engine." Cidr. Vol. 5. No. 2005. 2005.
3. Abdelwahab, Sherif, et al. "Cloud of things for sensing as a service: sensing resource discovery and virtualization." Global Communications Conference (GLOBECOM), 2015 IEEE. IEEE, 2015.
4. Abdelwahab, Sherif, et al. "Enabling smart cloud services through remote sensing: An internet of everything enabler." IEEE Internet of Things Journal 1.3 (2014): 276-288.
5. Akyildiz, Ian F., et al. "A survey on sensor networks." IEEE Communications magazine 40.8 (2002): 102-114.
6. Arasu, Arvind, et al. "Stream: The stanford data stream management system." Data Stream Management. Springer Berlin Heidelberg, 2016. 317-336.
7. Arasu, Arvind, et al. "STREAM: the stanford stream data manager (demonstration description)." Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003.
8. Arasu, Arvind, Shivnath Babu, and Jennifer Widom. "The CQL continuous query language: semantic foundations and query execution." The VLDB Journal—The International Journal on Very Large Data Bases 15.2 (2006): 121-142.
9. Balazinska, Magdalena, Hari Balakrishnan, and Michael Stonebraker. "Load management and high availability in the Medusa distributed stream processing system." Proceedings of the 2004 ACM SIGMOD international conference on Management of data. ACM, 2004.
10. Bonnet, Philippe, Johannes Gehrke, and Praveen Seshadri. "Towards sensor database systems." Mobile Data Management. Springer Berlin/Heidelberg, 2001.

11. Bradley, Joseph, Joel Barbier, and Doug Handler. "Embracing the internet of everything to capture your share of $14.4 trillion." White Paper, Cisco (2013).
12. Carney, Don, et al. "Operator scheduling in a data stream manager." Proceedings of the 29th international conference on Very large data bases-Volume 29. VLDB Endowment, 2003.
13. Chandrasekaran, Sirish, and Michael J. Franklin. "Streaming queries over streaming data." Proceedings of the 28th international conference on Very Large Data Bases. VLDB Endowment, 2002.
14. Chandrasekaran, Sirish, et al. "TelegraphCQ: continuous dataflow processing." Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003.
15. Compton, Michael, et al. "The SSN ontology of the W3C semantic sensor network incubator group." Web semantics: science, services and agents on the World Wide Web 17 (2012): 25-32.
16. Corcho, Oscar, and Raúl García-Castro. "Five challenges for the semantic sensor web." Semantic Web 1.1, 2 (2010): 121-125.
17. Cranor, Charles D., Theodore Johnson, and Oliver Spatscheck. "Stream Processing Techniques for Network Management." Data Stream Management. Springer Berlin Heidelberg, 2016. 431-449.
18. Cranor, Chuck, et al. "Gigascope: High performance network monitoring with an SQL interface." Proceedings of the 2002 ACM SIGMOD international conference on Management of data. ACM, 2002.
19. da Rocha, Atslands R., et al. "A semantic middleware for autonomic wireless sensor networks." Proceedings of the 2009 workshop on middleware for ubiquitous and pervasive systems. ACM, 2009.
20. Eisenhauer, Markus, Peter Rosengren, and Pablo Antolin. "A development platform for integrating wireless devices and sensors into ambient intelligence systems." Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2009. SECON Workshops' 09. 6th Annual IEEE Communications Society Conference on. IEEE, 2009.

21. Giffinger, Rudolf, et al. "Smart cities. Ranking of European medium-sized cities, Final Report, Centre of Regional Science, Vienna UT." (2007): 303-320.

22. Gulisano, Vincenzo, et al. "Streamcloud: An elastic and scalable data streaming system." IEEE Transactions on Parallel and Distributed Systems 23.12 (2012): 2351-2365.

23. Huang, Vincent, and Muhammad Kashif Javed. "Semantic sensor information description and processing." Sensor Technologies and Applications, 2008. SENSORCOMM'08. Second International Conference on. IEEE, 2008.

24. Issarny, Valerie, Mauro Caporuscio, and Nikolaos Georgantas. "A perspective on the future of middleware-based software engineering." 2007 Future of Software Engineering. IEEE Computer Society, 2007.

25. Johnson, Theodore, et al. "A heartbeat mechanism and its application in gigascope." Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, 2005.

26. Kim, Mihui, et al. "Developing an On-Demand Cloud-Based Sensing-as-a-Service System for Internet of Things." Journal of Computer Networks and Communications 2016 (2016).

27. Lee, Kwang-Won, Jung-Hwan Park, and Ryum-Duck Oh. "Design of active semantic middleware system to support incomplete sensor information based on ubiquitous sensor network." Application of Information and Communication Technologies (AICT), 2010 4th International Conference on. IEEE, 2010.

28. Le-Phuoc, Danh, et al. "The linked sensor middleware–connecting the real world and the semantic web." Proceedings of the Semantic Web Challenge 152 (2011): 22-23.

29. Madden, Samuel, et al. "Continuously adaptive continuous queries over streams." Proceedings of the 2002 ACM SIGMOD international conference on Management of data. ACM, 2002.

30. P. Guillemin and P. Friess. (2009). Internet of things strategic research roadmap. Technical report, The Cluster of European Research Projects.

31. Patroumpas, Kostas, and Timos Sellis. "Window specification over data streams." Current Trends in Database Technology–EDBT 2006 (2006): 445-464.

32. Perera, Charith, et al. "Sensing as a service model for smart cities supported by internet of things." Transactions on Emerging Telecommunications Technologies 25.1 (2014): 81-93.
33. Perera, Charith, et al. "Sensor search techniques for sensing as a service architecture for the internet of things." IEEE Sensors Journal 14.2 (2014): 406-420.
34. Perera, Charith, Chi Harold Liu, and Srimal Jayawardena. "The emerging internet of things marketplace from an industrial perspective: A survey." IEEE Transactions on Emerging Topics in Computing 3.4 (2015): 585-598.
35. Shah, Mehul A., et al. "Flux: An adaptive partitioning operator for continuous query systems." Data Engineering, 2003. Proceedings. 19th International Conference on. IEEE, 2003.
36. Shah, Mehul A., et al. "Java support for data-intensive systems: Experiences building the Telegraph dataflow system." ACM Sigmod Record 30.4 (2001): 103-114.
37. Sheng, Xiang, et al. "Sensing as a service: Challenges, solutions and future directions." IEEE Sensors journal 13.10 (2013): 3733-3741.
38. Soldatos, John, et al. "Openiot: Open source internet-of-things in the cloud." Interoperability and open-source solutions for the internet of things. Springer, Cham, 2015. 13-25.
39. Stonebraker, Michael, Uğur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing." ACM SIGMOD Record 34.4 (2005): 42-47.
40. Su, Kehua, Jie Li, and Hongbo Fu. "Smart city and the applications." Electronics, Communications and Control (ICECC), 2011 International Conference on. IEEE, 2011.
41. Sundmaeker, Harald, et al. "Vision and challenges for realising the Internet of Things." Cluster of European Research Projects on the Internet of Things, European Commision 3.3 (2010): 34-36.
42. Tan, Lu, and Neng Wang. "Future internet: The internet of things." Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on. Vol. 5. IEEE, 2010.

43. Wang, Meisong, et al. "City Data Fusion: Sensor Data Fusion in the Internet of Things." International Journal of Distributed Systems and Technologies (IJDST) 7.1 (2016): 15-36.
44. Zafeiropoulos, A., et al. "Data Management in the Semantic Web, ser. Distributed, Cluster and Grid Computing-Yi Pan (Georgia State University), Series Edito, H. Jin, Ed." (2011).
45. Zafeiropoulos, Anastasios, et al. "A semantic-based architecture for sensor data fusion." Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM'08. The Second International Conference on. IEEE, 2008.
46. Zhong, Jianlong, and Bingsheng He. "Medusa: A parallel graph processing system on graphics processors." ACM SIGMOD Record 43.2 (2014): 35-40.
47. "5 Things To Know About The IBM Internet Of Things Foundation." Ibm.com. N.p., 2017. Web. 23 July 2017.
48. "How AWS Iot Works - AWS Iot." Docs.aws.amazon.com. N.p., 2017. Web. 23 July 2017.
49. "IBM Bluemix Docs." Console.bluemix.net. N.p., 2017. Web. 23 July 2017.
50. "Microsoft Azure Iot Suite Overview." Docs.microsoft.com. N.p., 2017. Web. 23 July 2017.
51. Amazon Web Services, Inc. (2017). Amazon CloudWatch - Cloud & Network Monitoring Services. [online] Available at: https://aws.amazon.com/cloudwatch/ [Accessed 20 Aug. 2017].
52. Amazon Web Services, Inc. (2017). Amazon DynamoDB Getting Started – Amazon Web Services. [online] Available at: https://aws.amazon.com/dynamodb/getting-started/ [Accessed 20 Aug. 2017].
53. Amazon Web Services, Inc. (2017). Amazon Kinesis Streams – Amazon Web Services (AWS). [online] Available at: https://aws.amazon.com/kinesis/streams/ [Accessed 20 Aug. 2017].
54. Amazon Web Services, Inc. (2017). Amazon RDS for MySQL – Amazon Web Services (AWS). [online] Available at: https://aws.amazon.com/rds/mysql/ [Accessed 20 Aug. 2017]

55. Amazon Web Services, Inc. (2017). What is AWS? - Amazon Web Services. [online] Available at: https://aws.amazon.com/what-is-aws/ [Accessed 20 Aug. 2017].
56. Docs.aws.amazon.com. (2017). Message Broker for AWS IoT - AWS IoT. [online] Available at: http://docs.aws.amazon.com/iot/latest/developerguide/iot-message-broker.html [Accessed 20 Aug. 2017].
57. Docs.aws.amazon.com. (2017). Regions and Availability Zones - Amazon Elastic Compute Cloud. [online] Available at: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html [Accessed 20 Aug. 2017].
58. Docs.aws.amazon.com. (2017). Using the AWS IoT Device SDK for JavaScript - AWS IoT. [online] Available at: http://docs.aws.amazon.com/iot/latest/developerguide/iot-device-sdk-node.html [Accessed 20 Aug. 2017].
59. EC2, (2017). [online] Available at: https://www.amazonaws.cn/en/ec2/details/ [Accessed 20 Aug. 2017].
60. Eclipse Paho, (2017). [online] Available at: https://eclipse.org/paho/clients/java/ [Accessed 20 Aug. 2017].
61. Gatling Load and Performance testing. (2017). Gatling Load and Performance testing - Open-source load and performance testing. [online] Available at: http://gatling.io/ [Accessed 20 Aug. 2017].
62. GitHub. (2017). JSQLParser/JSqlParser. [online] Available at: https://github.com/JSQLParser/JSqlParser/wiki [Accessed 20 Aug. 2017].
63. MongoDB, (2017). [online] Available at: https://docs.mongodb.com/manual/core/storage-engine [Accessed 20 Aug. 2017].
64. Mqtt.org. (2017). MQTT. [online] Available at: http://mqtt.org/ [Accessed 23 Jul. 2017].
65. Projects.spring.io. (2017). Spring Boot. [online] Available at: https://projects.spring.io/spring-boot/ [Accessed 20 Aug. 2017].

66. Raspberry Pi. (2017). Download Raspbian for Raspberry Pi. [online] Available at: https://www.raspberrypi.org/downloads/raspbian/ [Accessed 20 Aug. 2017]

67. Raspberry Pi. (2017). Raspberry Pi 2 Model B - Raspberry Pi. [online] Available at: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/ [Accessed 20 Aug. 2017].

68. Raspberrypi.org. (2017). GPIO: Raspberry Pi Models A and B - Raspberry Pi Documentation. [online] Available at: https://www.raspberrypi.org/documentation/usage/gpio/ [Accessed 20 Aug. 2017].

69. Streamdata.io. (2017). Server-Sent Events explained with usecases. [online] Available at: https://streamdata.io/blog/server-sent-events/ [Accessed 20 Aug. 2017]

70. Sunfounder.com. (2017). Humiture Sensor Module. [online] Available at: https://www.sunfounder.com/humiture-sensor-module.html [Accessed 20 Aug. 2017].

71. Tomcat.apache.org. (2017). Apache Tomcat® - Apache Tomcat 8 Software Downloads. [online] Available at: https://tomcat.apache.org/download-80.cgi [Accessed 20 Aug. 2017].

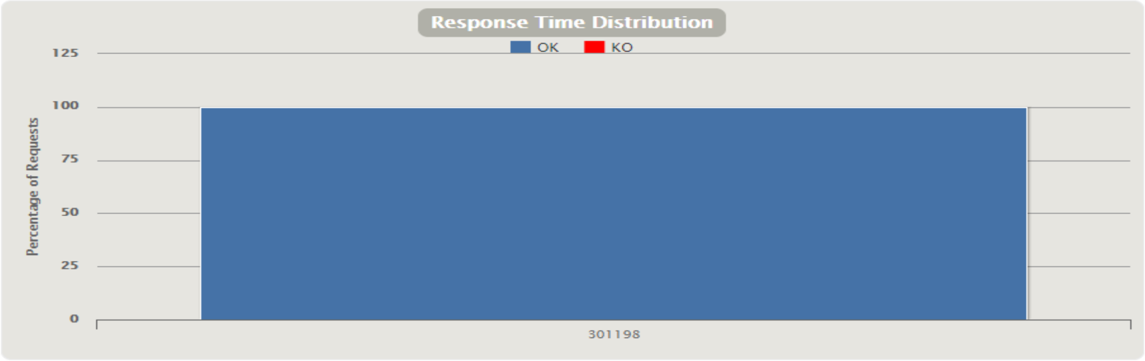
72. Tools, S., Editor, S., Codegen, S., UI, S., Inspector, S., Tools, C., Integrations, O., Discussion, R., Forum, O. and Champions, S. (2017). World's Most Popular API Framework | Swagger. [online] Swagger. Available at: https://swagger.io/ [Accessed 28 Aug. 2017].

73. W3.org. (2017). Server-Sent Events. [online] Available at: https://www.w3.org/TR/eventsource/#server-sent-events-intro [Accessed 20 Aug. 2017].

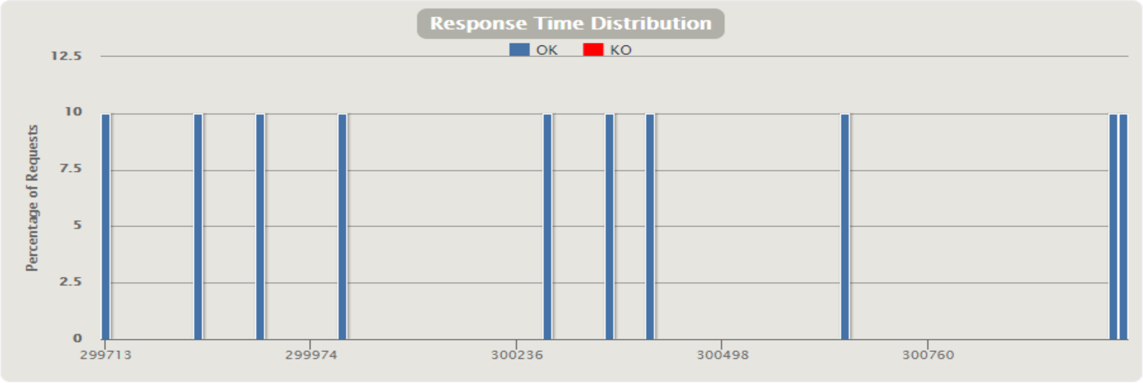
74. Watson Analytics. (2017). Watson Analytics. [online] Available at: https://www.ibm.com/watson-analytics [Accessed 20 Aug. 2017].

Appendices

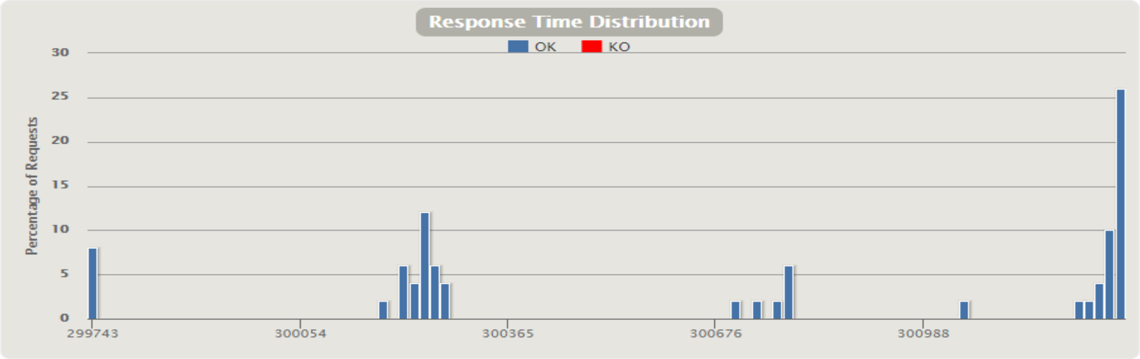
Appendix A: Response Time Distribution for First Scenario experiments



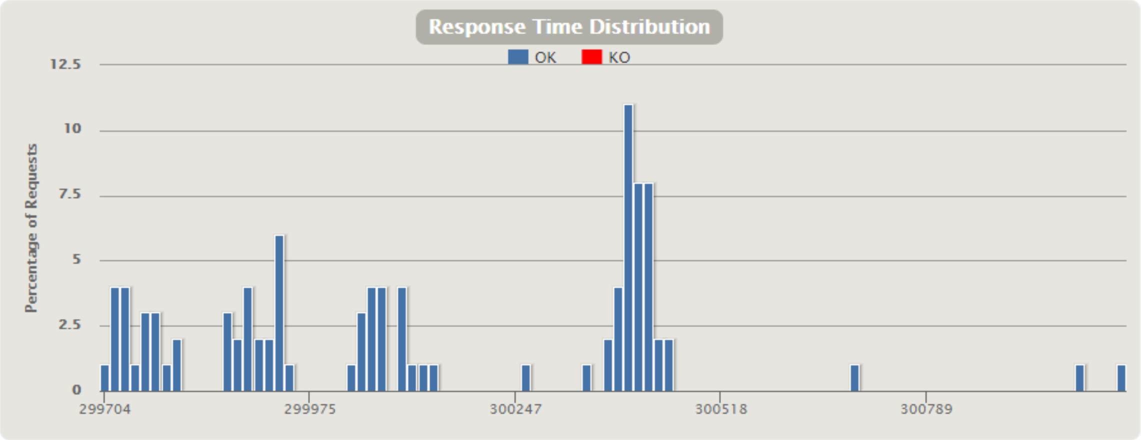
Appendix A Figure 1: 1 client application.



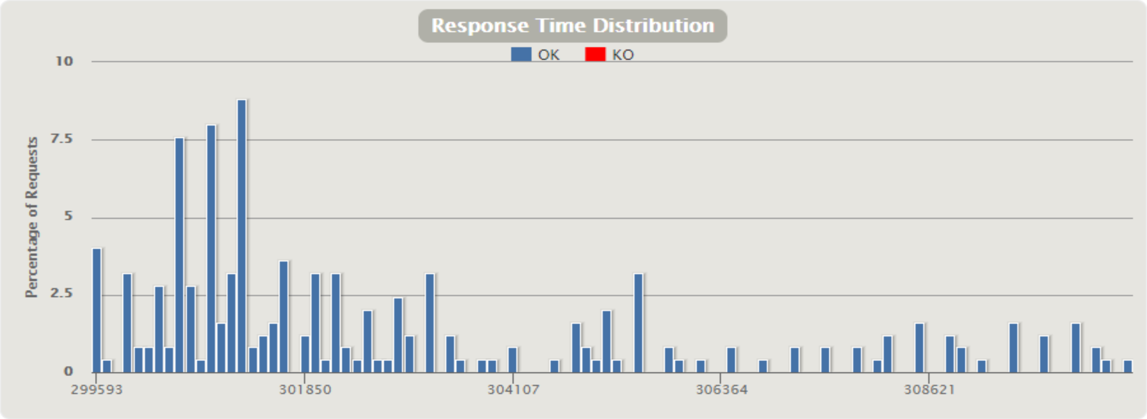
Appendix A Figure 2 10 client applications.



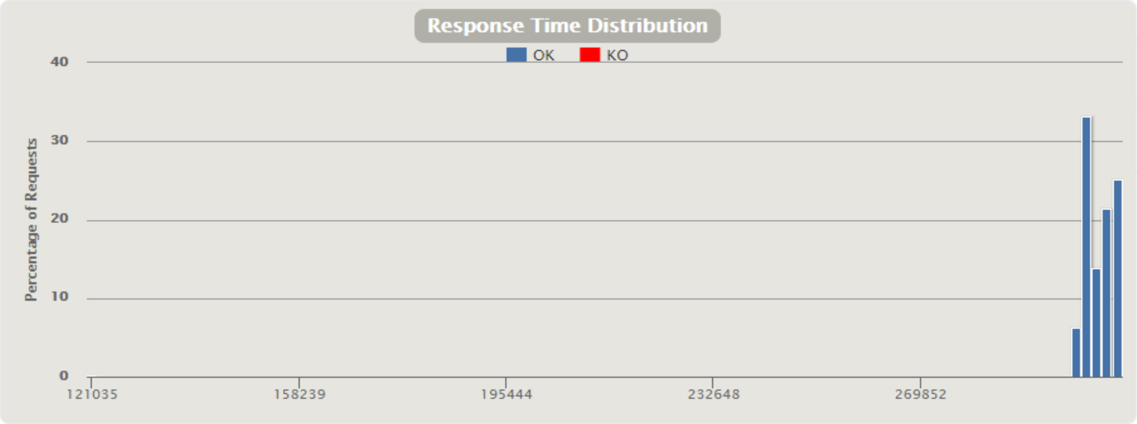
Appendix A Figure 3: 50 client applications



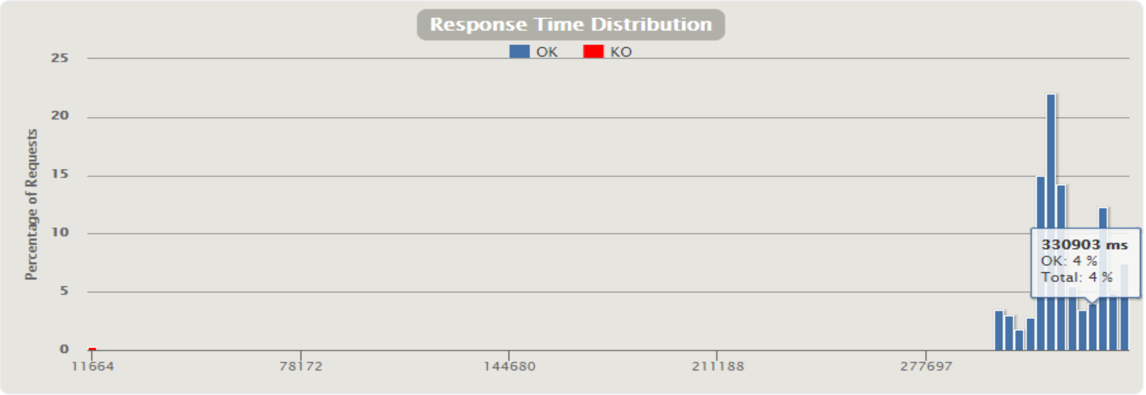
Appendix A Figure 4: 100 client applications



Appendix A Figure 5: 250 client applications

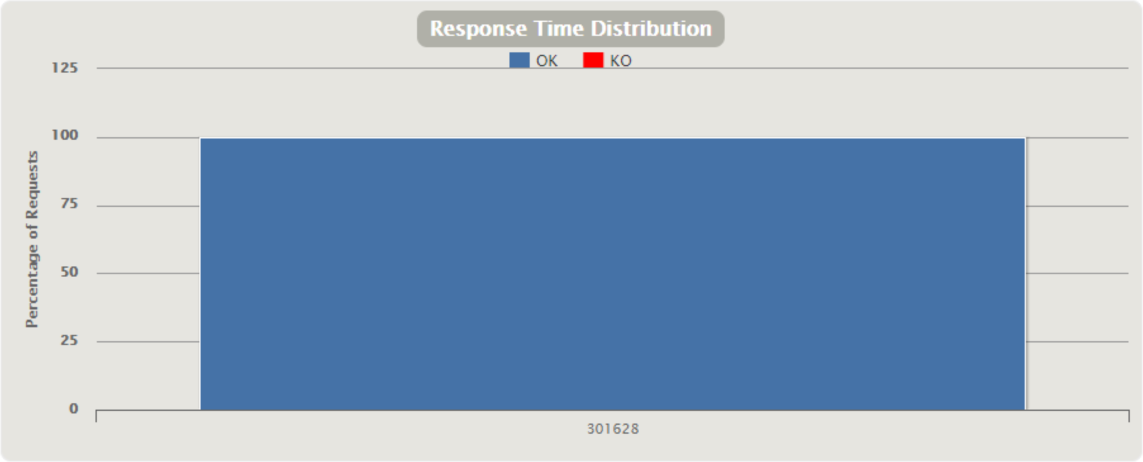


Appendix A Figure 6: 500 client applications

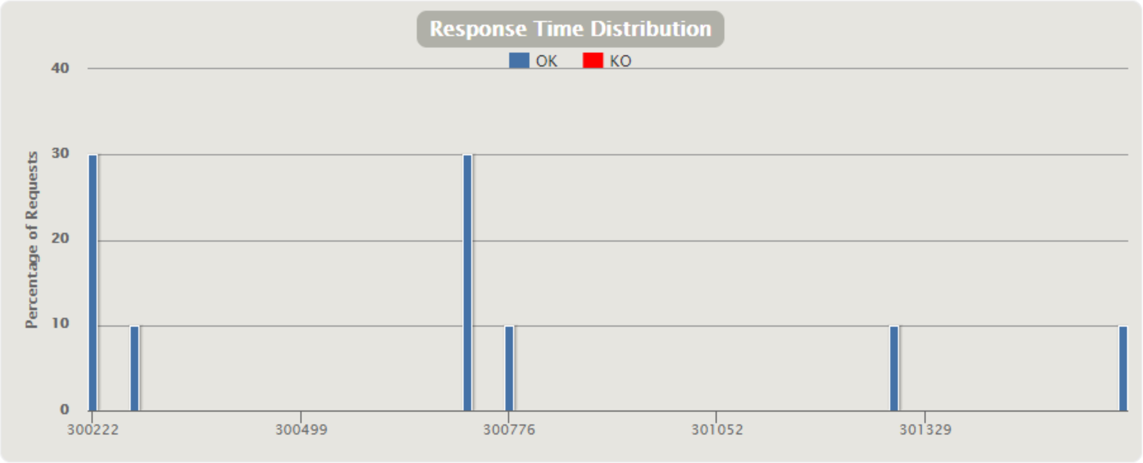


Appendix A Figure 7: 750 client applications

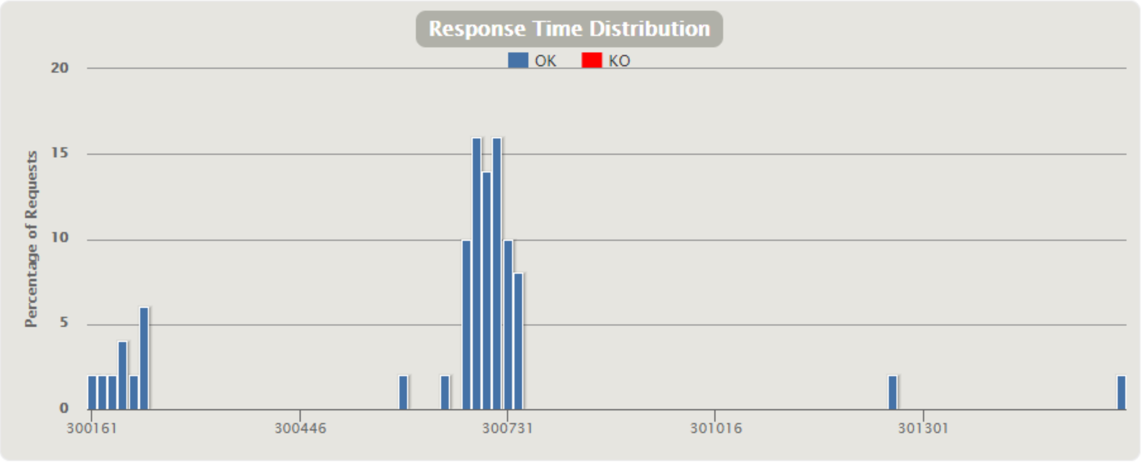
Appendix B: Response Time Distribution for Second Scenarios experiments



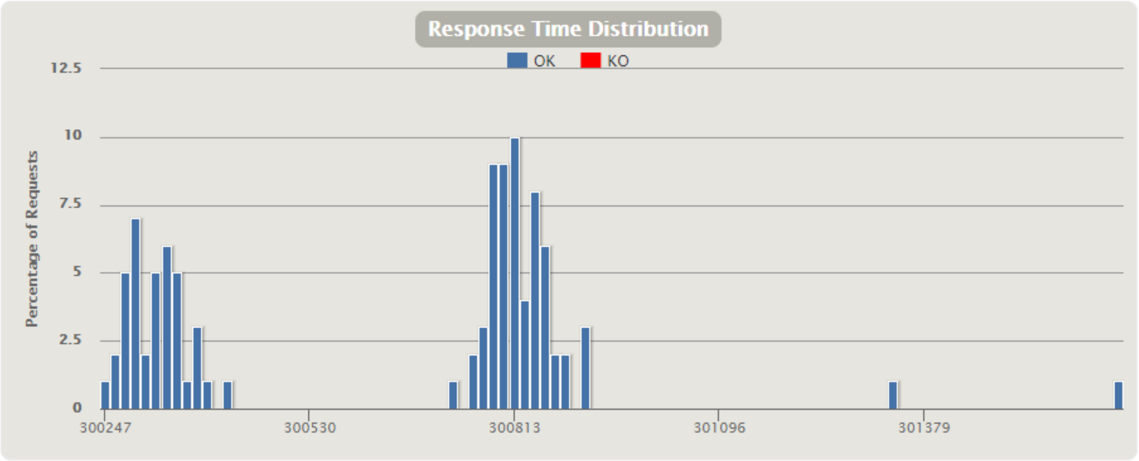
Appendix B Figure 1: 1 client application



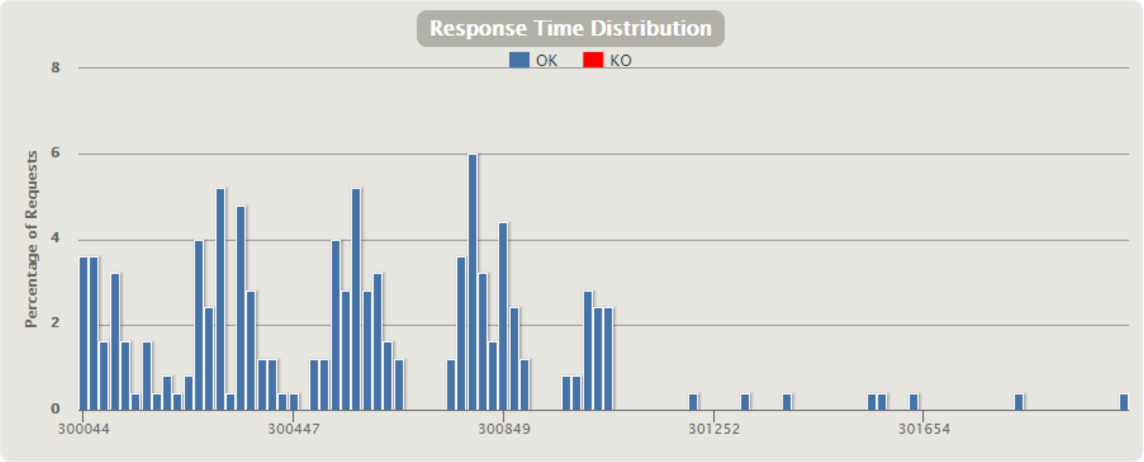
Appendix B Figure 2: 10 client applications



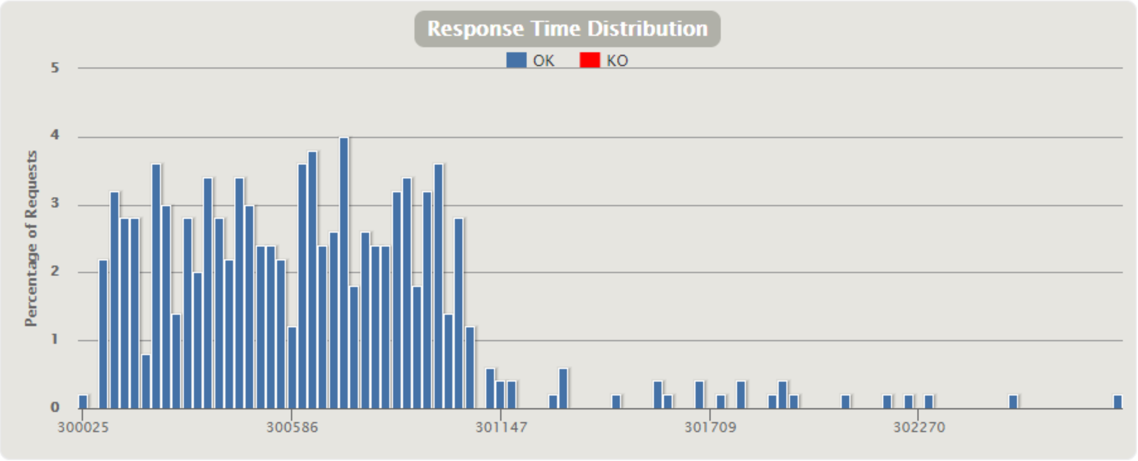
Appendix B Figure 3: 50 client applications



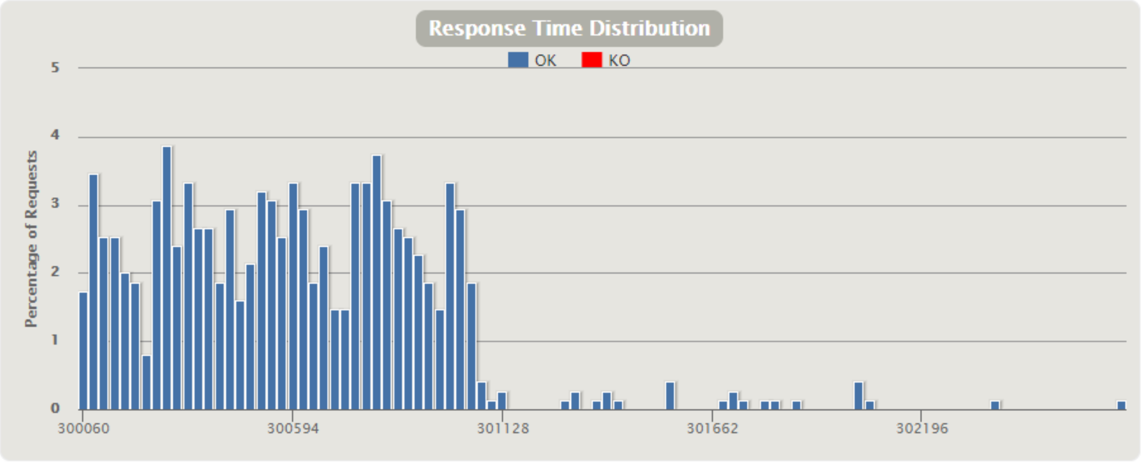
Appendix B Figure 4: 100 client applications



Appendix B Figure 5: 250 client applications

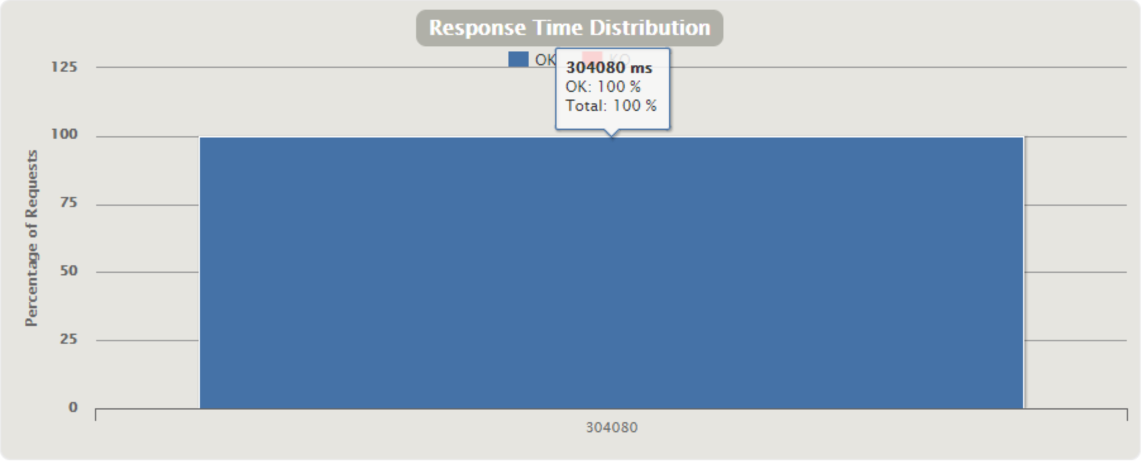


Appendix B Figure 6: 500 client applications

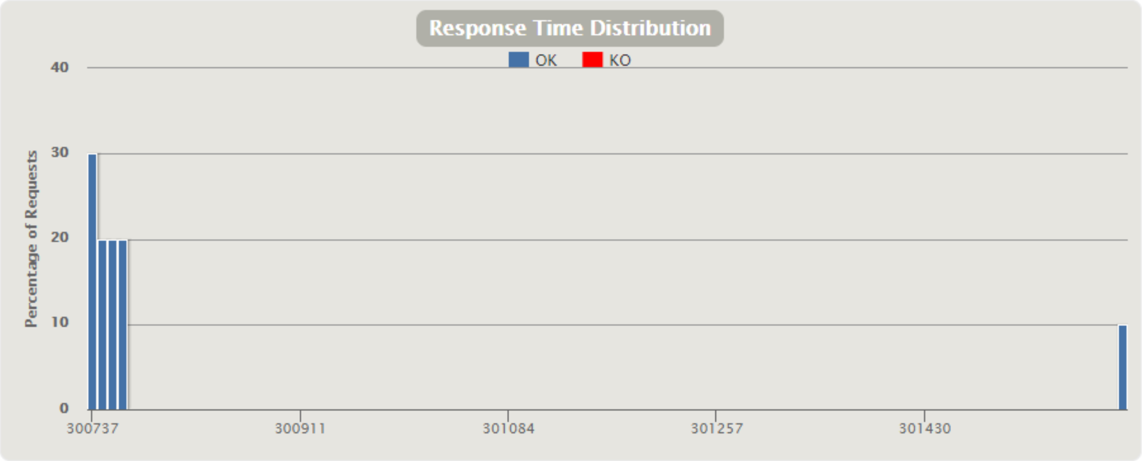


Appendix B Figure 7: 750 client applications

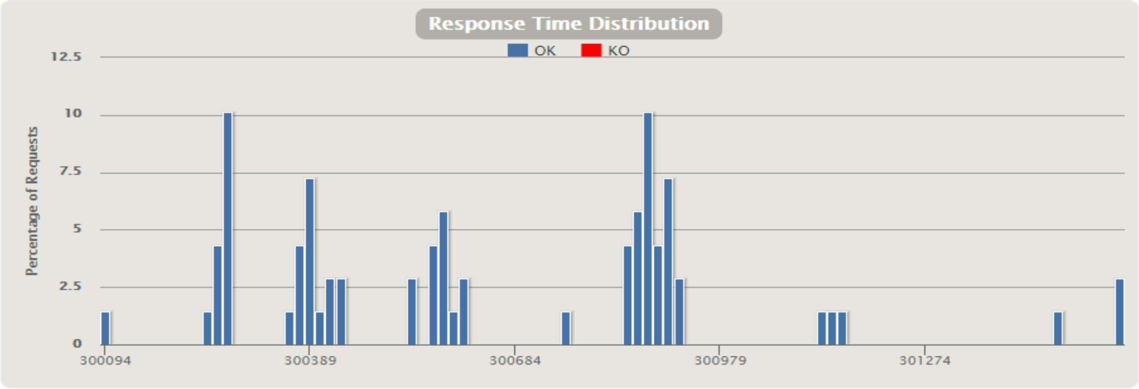
Appendix C: Response Time Distribution for third Scenarios experiments



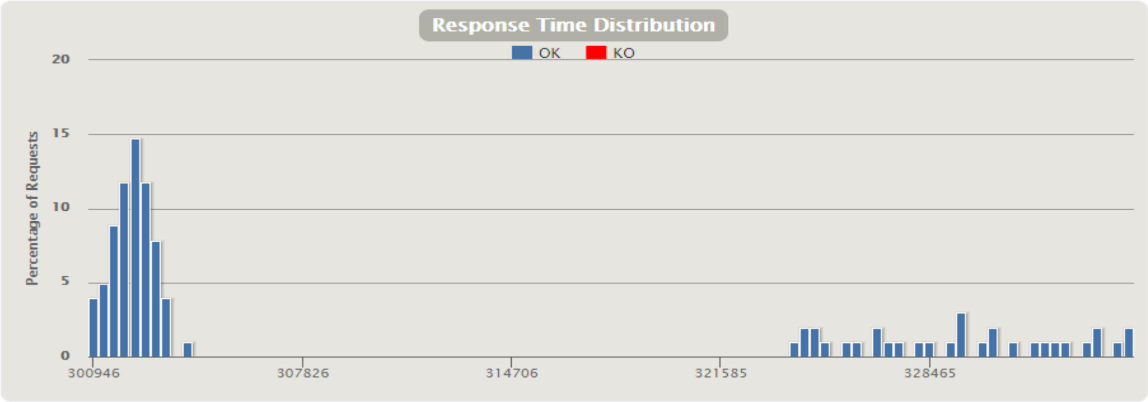
Appendix C Figure 1: 1 client application



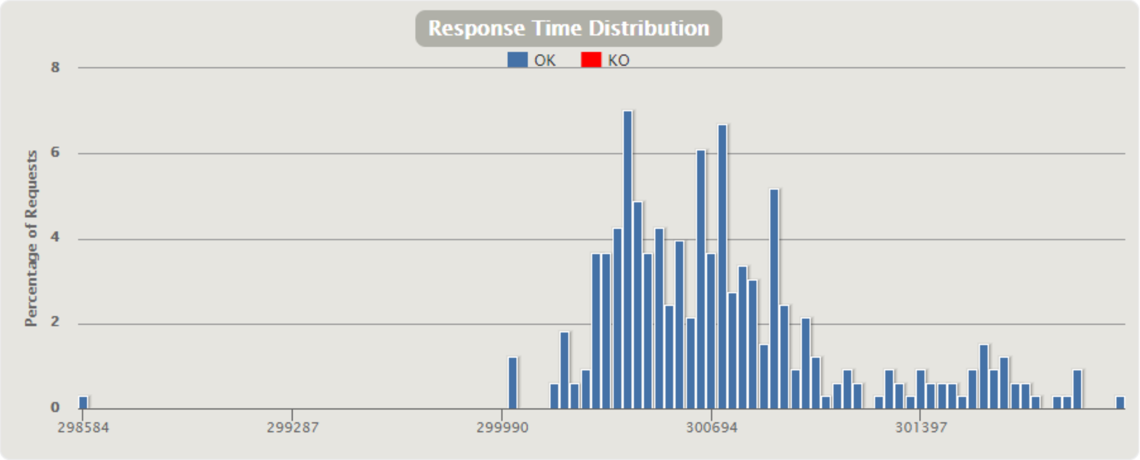
Appendix C Figure 2: 10 client applications



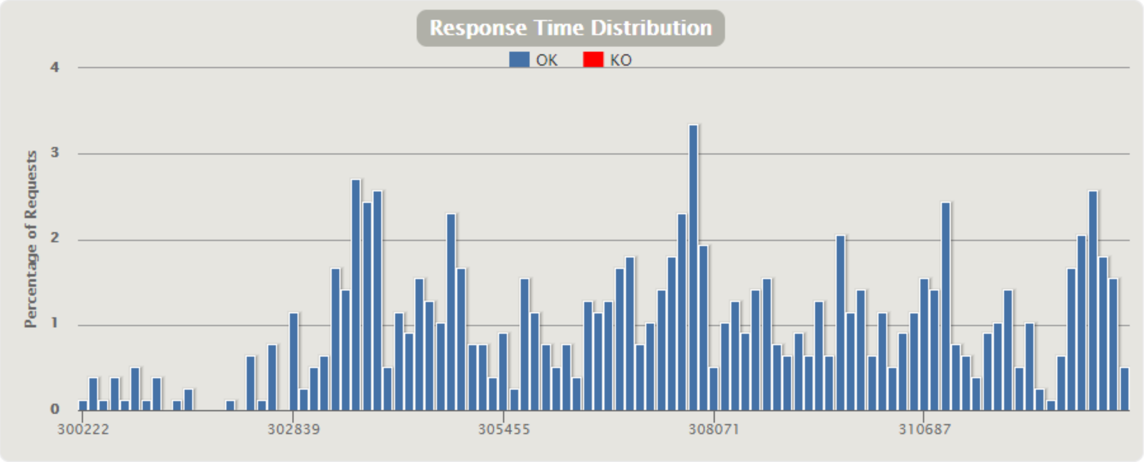
Appendix C Figure 3: 50 client applications



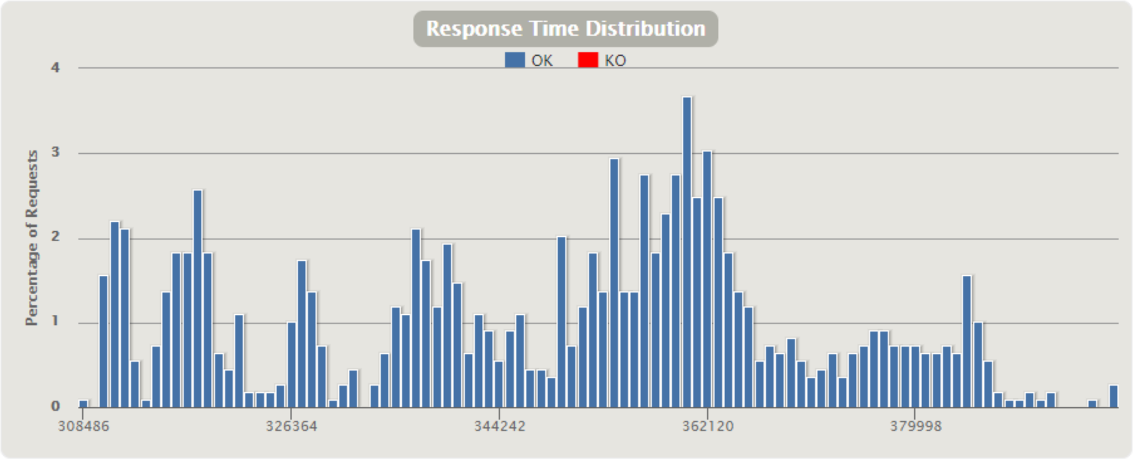
Appendix C Figure 4: 100 client applications



Appendix C Figure 5: 250 client applications

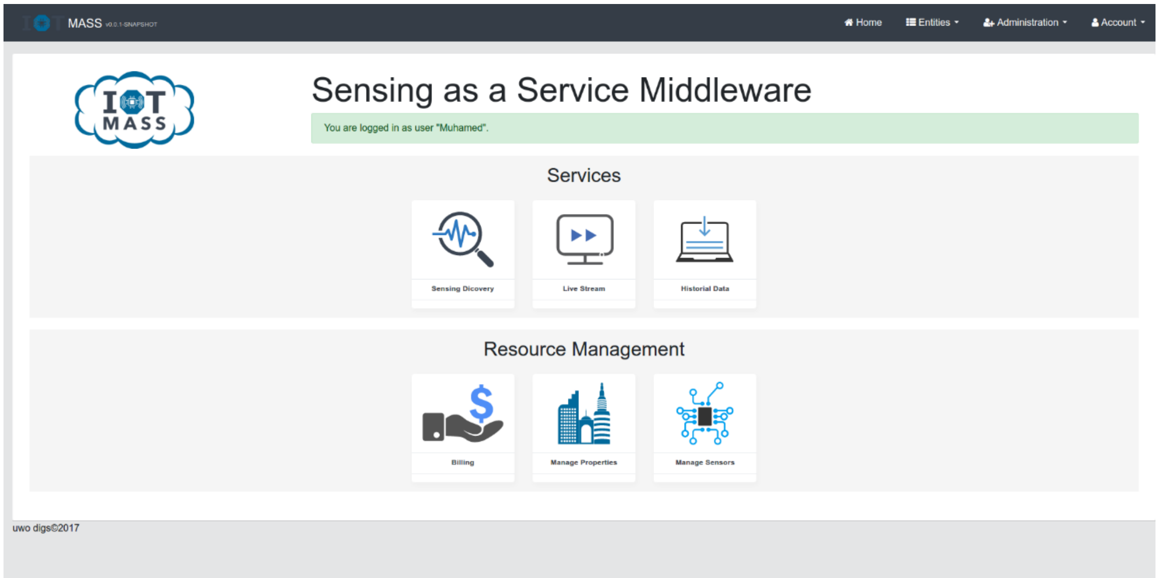


Appendix C Figure 6: 500 client applications



Appendix C Figure 7: 750 client applications

Appendix D: Middleware Prototype



Sensing as a Service Middleware

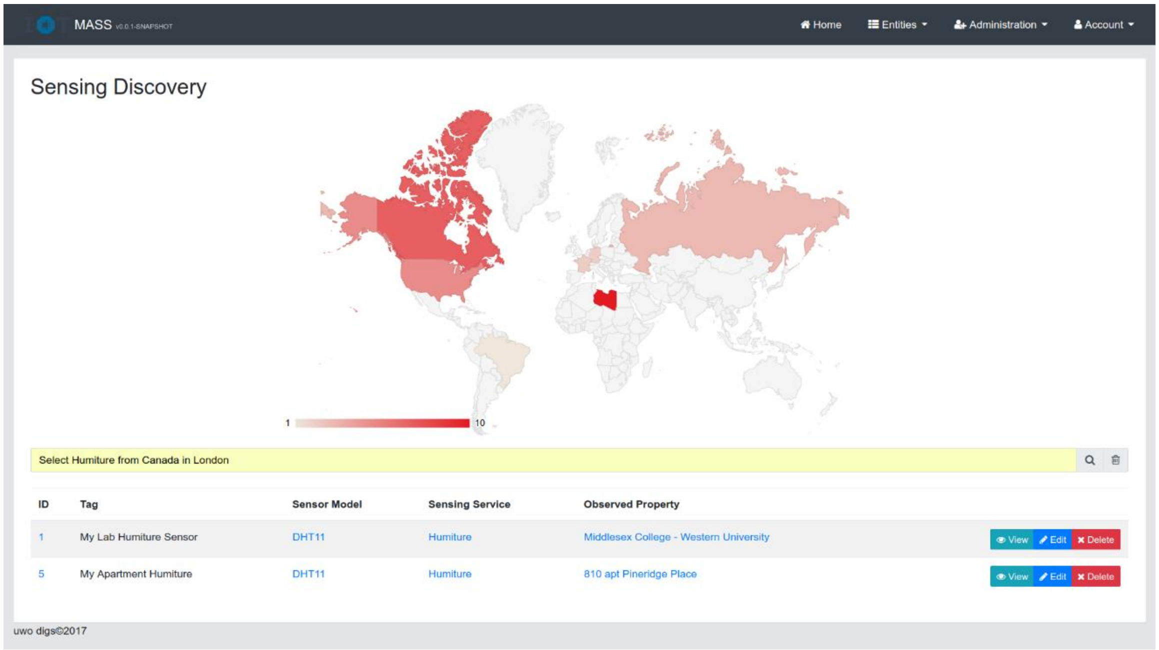
You are logged in as user "Muhamed".

Services

Resource Management

uwo digs©2017

Appendix D Figure 1 : Middleware Home Page.



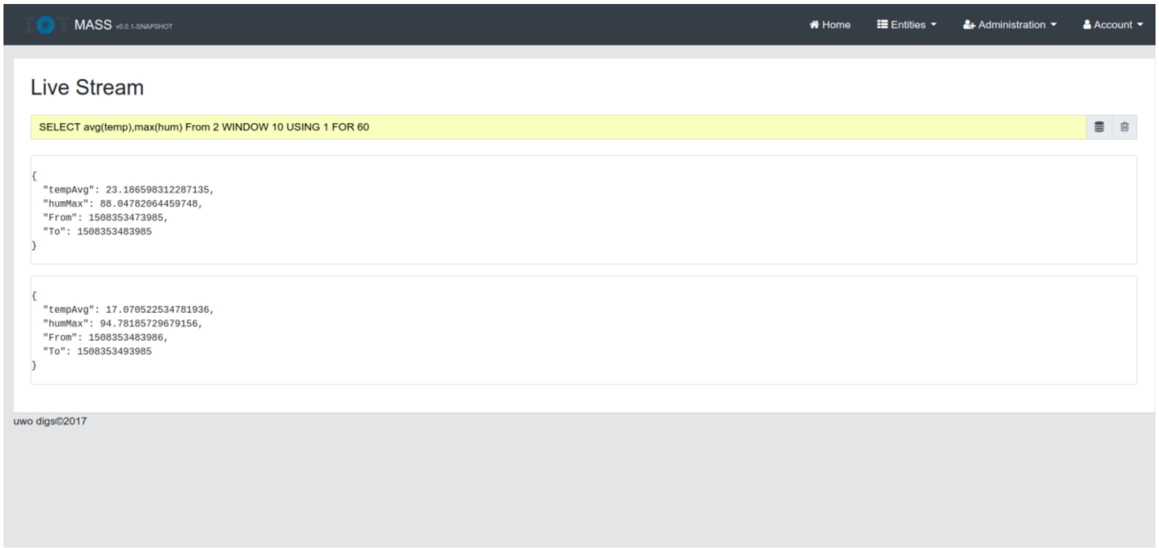
Sensing Discovery

Select Humiture from Canada in London

ID	Tag	Sensor Model	Sensing Service	Observed Property	
1	My Lab Humiture Sensor	DHT11	Humiture	Middlesex College - Western University	View Edit Delete
5	My Apartment Humiture	DHT11	Humiture	810 apt Pineridge Place	View Edit Delete

uwo digs©2017

Appendix D Figure 2: Sensing Discovery Module Interface.



Live Stream

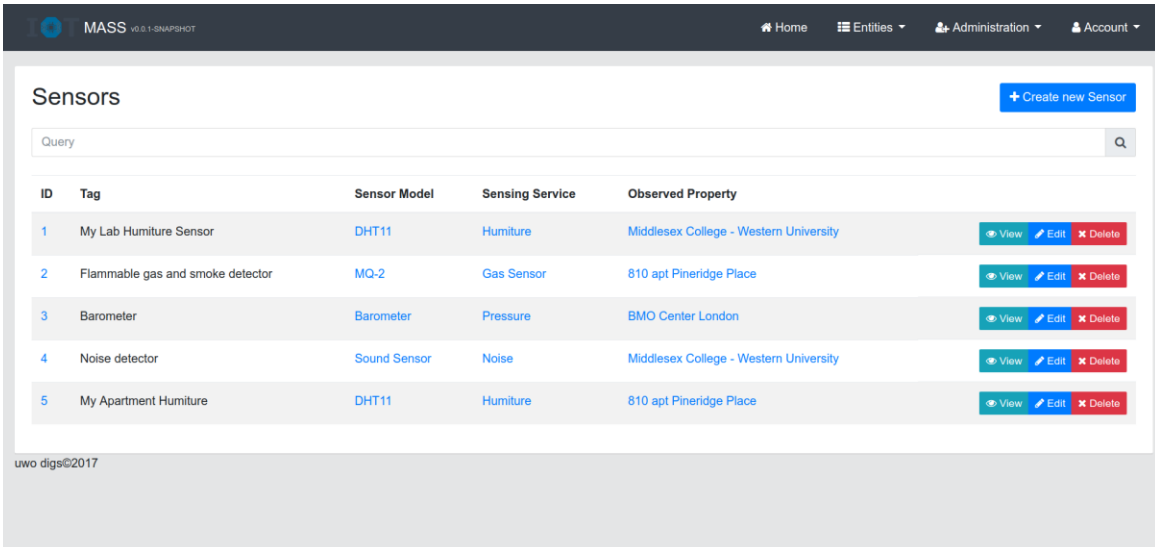
SELECT avg(temp),max(hum) From 2 WINDOW 10 USING 1 FOR 60

```
{
  "tempAvg": 23.186598312287135,
  "humMax": 88.04782064459748,
  "From": 1508353473985,
  "To": 1508353483985
}
```

```
{
  "tempAvg": 17.070522534781936,
  "humMax": 94.78185729679156,
  "From": 1508353483986,
  "To": 1508353493985
}
```

uwo digs©2017

Appendix D Figure 3: Live Stream Session.

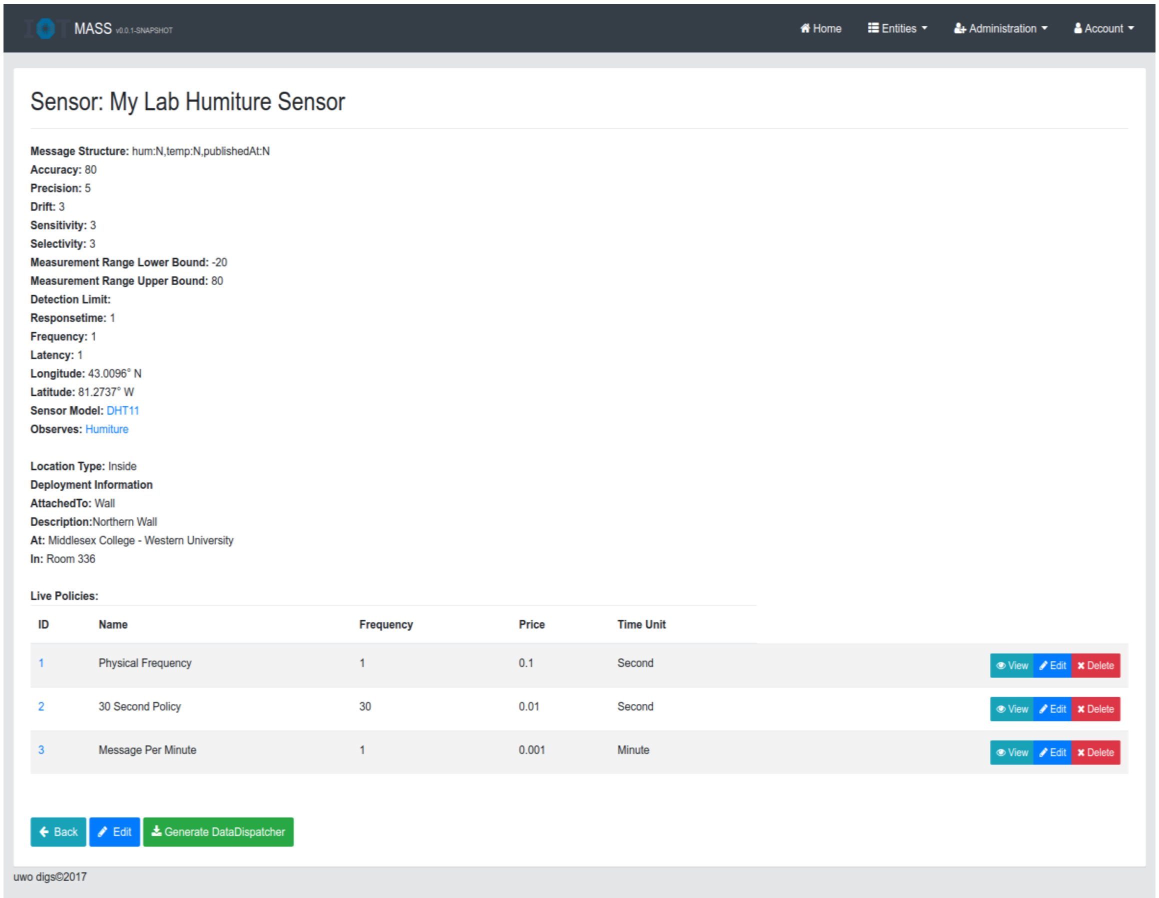


Sensors

ID	Tag	Sensor Model	Sensing Service	Observed Property	
1	My Lab Humiture Sensor	DHT11	Humiture	Middlesex College - Western University	View Edit Delete
2	Flammable gas and smoke detector	MQ-2	Gas Sensor	810 apt Pineridge Place	View Edit Delete
3	Barometer	Barometer	Pressure	BMO Center London	View Edit Delete
4	Noise detector	Sound Sensor	Noise	Middlesex College - Western University	View Edit Delete
5	My Apartment Humiture	DHT11	Humiture	810 apt Pineridge Place	View Edit Delete

uwo digs©2017

Appendix D Figure 4: Sensor Management Module.



Sensor: My Lab Humiture Sensor

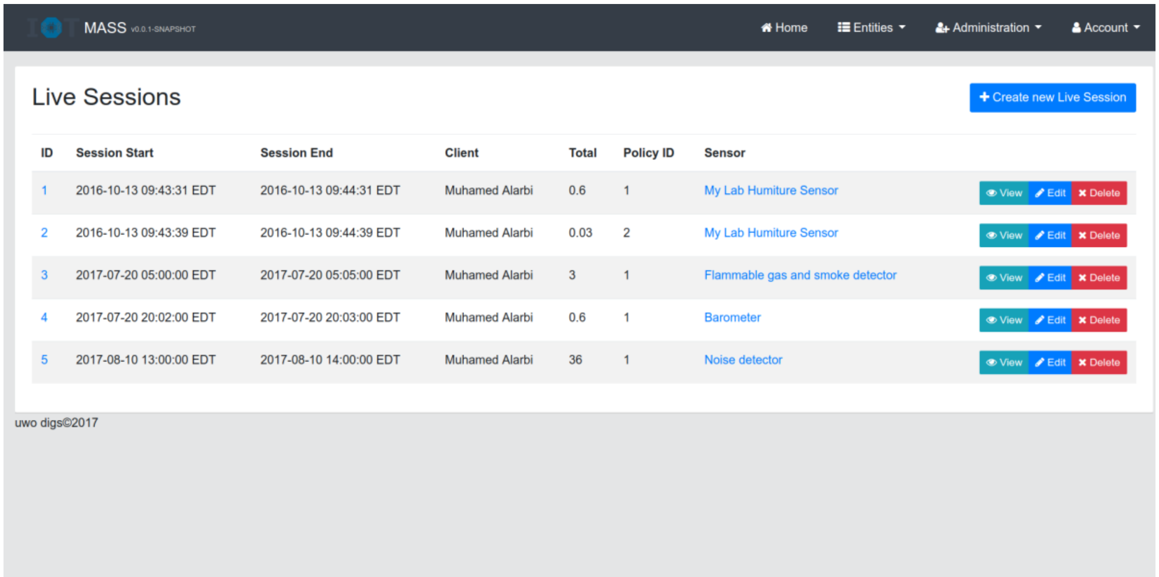
Message Structure: hum:N,temp:N,publishedAt:N
Accuracy: 80
Precision: 5
Drift: 3
Sensitivity: 3
Selectivity: 3
Measurement Range Lower Bound: -20
Measurement Range Upper Bound: 80
Detection Limit:
Responsetime: 1
Frequency: 1
Latency: 1
Longitude: 43.0096° N
Latitude: 81.2737° W
Sensor Model: DHT11
Observes: Humiture

Location Type: Inside
Deployment Information
AttachedTo: Wall
Description: Northern Wall
At: Middlesex College - Western University
In: Room 336

Live Policies:

ID	Name	Frequency	Price	Time Unit	
1	Physical Frequency	1	0.1	Second	View Edit Delete
2	30 Second Policy	30	0.01	Second	View Edit Delete
3	Message Per Minute	1	0.001	Minute	View Edit Delete

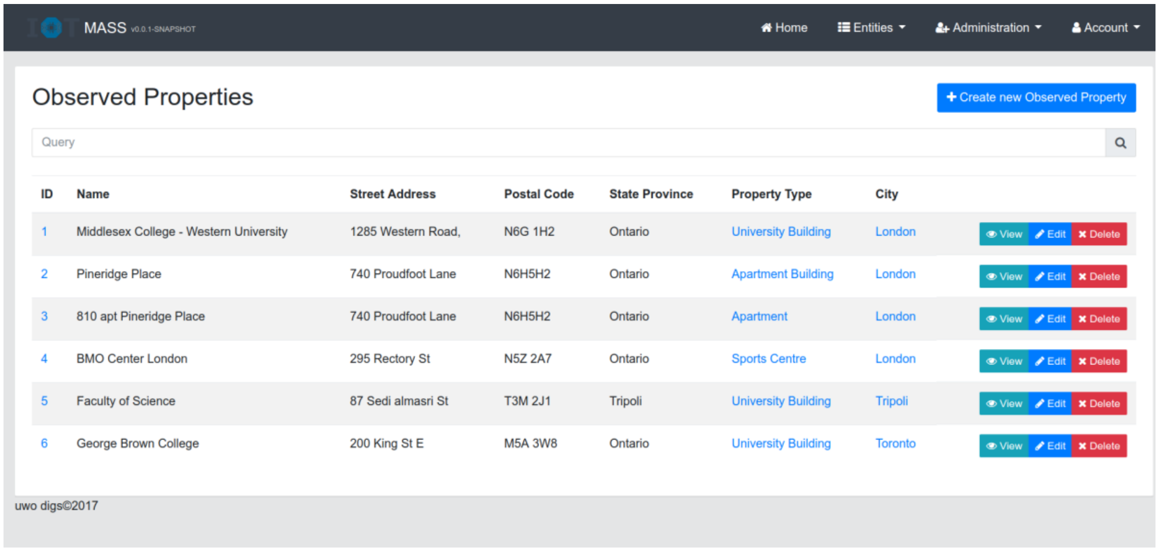
Appendix D Figure 5: Sensor Details.



Live Sessions

ID	Session Start	Session End	Client	Total	Policy ID	Sensor	
1	2016-10-13 09:43:31 EDT	2016-10-13 09:44:31 EDT	Muhamed Alarbi	0.6	1	My Lab Humiture Sensor	View Edit Delete
2	2016-10-13 09:43:39 EDT	2016-10-13 09:44:39 EDT	Muhamed Alarbi	0.03	2	My Lab Humiture Sensor	View Edit Delete
3	2017-07-20 05:00:00 EDT	2017-07-20 05:05:00 EDT	Muhamed Alarbi	3	1	Flammable gas and smoke detector	View Edit Delete
4	2017-07-20 20:02:00 EDT	2017-07-20 20:03:00 EDT	Muhamed Alarbi	0.6	1	Barometer	View Edit Delete
5	2017-08-10 13:00:00 EDT	2017-08-10 14:00:00 EDT	Muhamed Alarbi	36	1	Noise detector	View Edit Delete

Appendix D Figure 6: Billing Module.



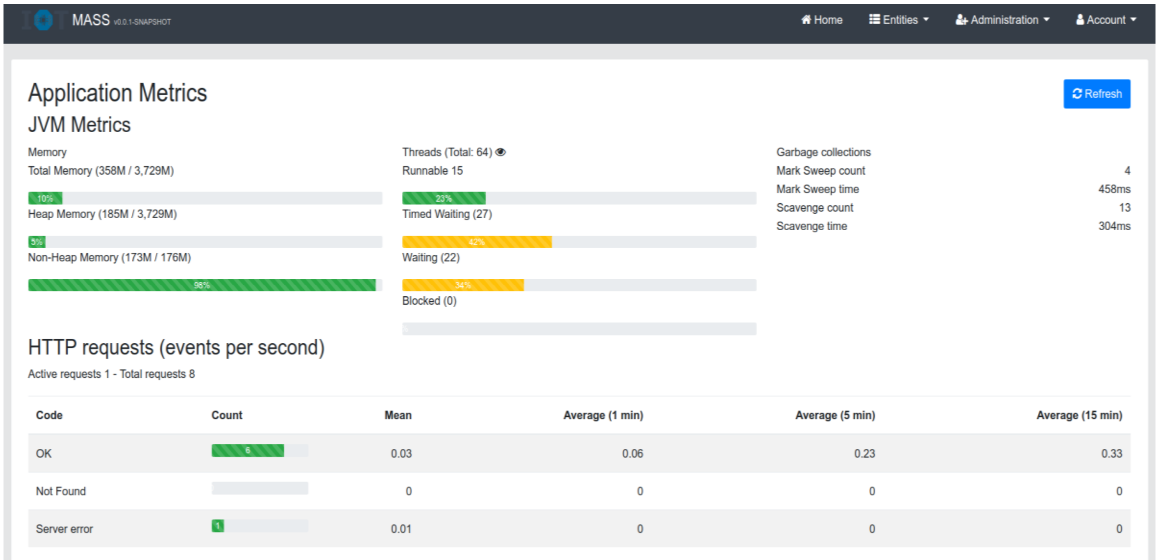
MASS v0.0.1-SNAPSHOT

Observed Properties

ID	Name	Street Address	Postal Code	State Province	Property Type	City	
1	Middlesex College - Western University	1285 Western Road,	N6G 1H2	Ontario	University Building	London	View Edit Delete
2	Pineridge Place	740 Proudfoot Lane	N6H5H2	Ontario	Apartment Building	London	View Edit Delete
3	810 apt Pineridge Place	740 Proudfoot Lane	N6H5H2	Ontario	Apartment	London	View Edit Delete
4	BMO Center London	295 Rectory St	N5Z 2A7	Ontario	Sports Centre	London	View Edit Delete
5	Faculty of Science	87 Sedi almasri St	T3M 2J1	Tripoli	University Building	Tripoli	View Edit Delete
6	George Brown College	200 King St E	M5A 3W8	Ontario	University Building	Toronto	View Edit Delete

uwo digs©2017

Appendix D Figure 7: Observed Properties Page.



MASS v0.0.1-SNAPSHOT

Application Metrics

JVM Metrics

Memory
Total Memory (358M / 3,729M) 10%
Heap Memory (185M / 3,729M) 5%
Non-Heap Memory (173M / 176M) 98%

Threads (Total: 64)
Runnable 15 23%
Timed Waiting (27) 42%
Waiting (22) 34%
Blocked (0)

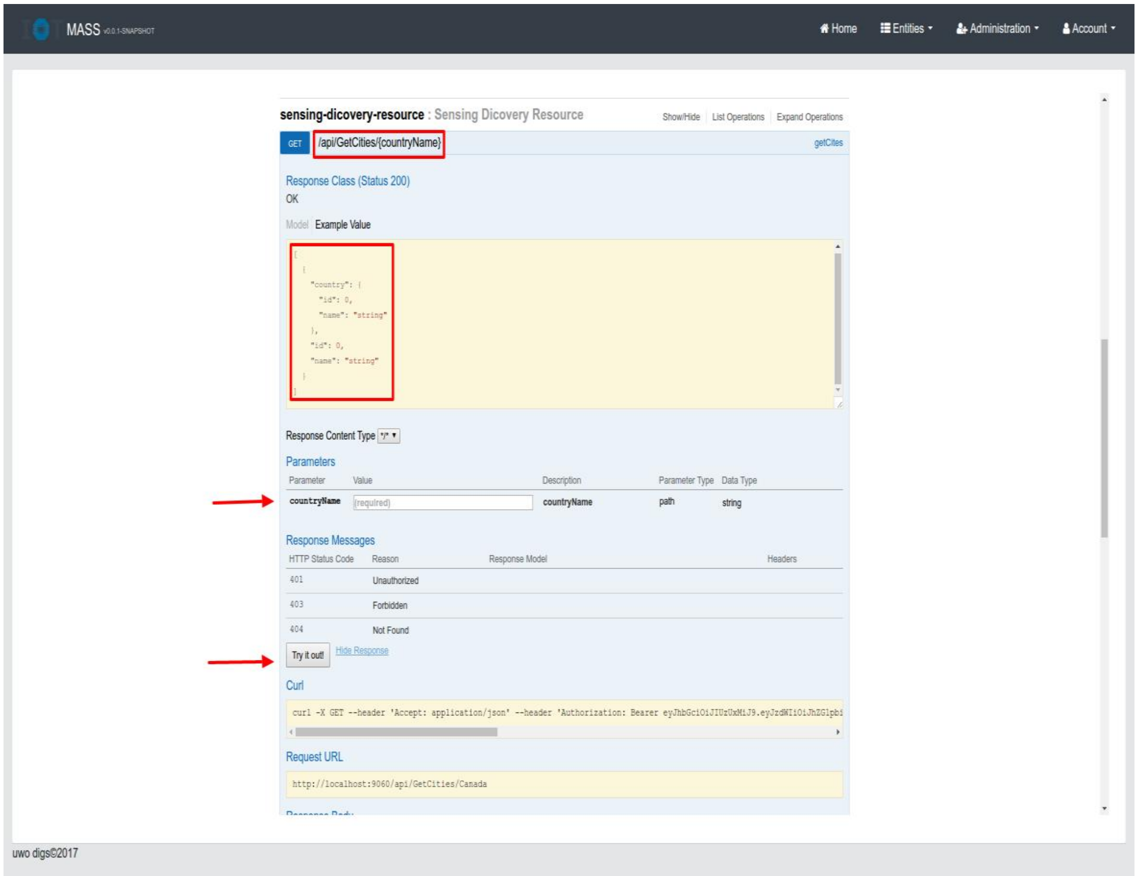
Garbage collections	
Mark Sweep count	4
Mark Sweep time	458ms
Scavenge count	13
Scavenge time	304ms

HTTP requests (events per second)

Active requests 1 - Total requests 8

Code	Count	Mean	Average (1 min)	Average (5 min)	Average (15 min)
OK	6	0.03	0.06	0.23	0.33
Not Found		0	0	0	0
Server error	1	0.01	0	0	0

Appendix D Figure 8: Middleware Health Dashboard.



sensing-dicovery-resource : Sensing Dicovery Resource

GET /api/GetCities/{countryName} getCites

Response Class (Status 200)

OK

Model Example Value

```
[
  {
    "country": {
      "id": 0,
      "name": "string"
    },
    "id": 0,
    "name": "string"
  }
]
```

Response Content Type */*

Parameters

Parameter	Value	Description	Parameter Type	Data Type
countryName	(required)	countryName	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Curl

```
curl -X GET --header 'Accept: application/json' --header 'Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZGlpbi
```

Request URL

```
http://localhost:9060/api/GetCities/Canada
```

Appendix D Figure 9: API Documentation.

Curriculum Vitae

Name:	Muhamed Alarbi
Post-secondary Education and Degrees:	University of Tripoli Tripoli, Libya 2007-2011 B.Sc. The University of Western Ontario London, Ontario, Canada 2015-2017 M.Sc.
Honors and Awards:	University of Tripoli Graduate Scholarship 2014-2017
Related Work Experience	Teaching Assistant University of Tripoli 2013-2014