QUALITÉ DE SERVICE DANS L'IOT : COUCHE DE BROUILLARD

par

Suvrojoti Paul

Mémoire présentée au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 30 octobre 2023

QUALITY OF SERVICE IN IOT: FOG LAYER

By

Suvrojoti Paul

Dissertation presented to the Department of Computer Science
with a view to obtaining the degree of Master of Science (M.Sc)

FACULTY OF SCIENCES

UNIVERSITY OF SHERBROOKE

Sherbrooke, Québec, Canada, 30 octobre 2023

Le 30 octobre 2023

Mémoire déposé par Suvrojoti Paul

Membres du jury

Prof. Bessam Abdulrazak, Directeur

Département d'informatique

Prof., Amine Trabelsi, Président-rapporteur

Département d'informatique

Prof., Yasir Malik, Évaluateur externe

Département d'informatique

Sommaire

L'Internet des objets (IdO) (Internet of Things en anglais), peut être défini comme une combinaison d’interactions entre les Humains et le monde technologique de l’Internet. De cet effet résulte une interconnexion entre les objets physiques et les appareils technologiques dans leur environnement proche. [1]. Ces dernières années le domaine de l'IdO s’est beaucoup développé, entrainant ainsi une augmentation du risque de défaillances en temps réel. Les défaillances sont souvent détectées par certains points de vulnérabilité dans le système. En se concentrant sur les causes profondes, le point de défaillance peut être détecter, ce qui conduit aux mesures à mettre en place pour surmonter les défaillances. Les systèmes IdO ont donc besoin d'avoir une architecture de Qualité de Service (QdS) adéquate. Ainsi, la QdS devient un enjeu crucial avec la démocratisation de l'IdO. La QdS est la description ou la mesure de la performance globale d'un service, tel qu'un réseau de téléphonie ou informatique, ou un service de cloud computing, en particulier la performance perçue par les utilisateurs du réseau. Dans cette étude, nous proposons les méthodes de mise en œuvre de la QdS dans les plateformes IdO. Nous mettrons en lumière les défis et les problèmes récurrents rencontrés par toutes les plateformes IdO, qui nous ont inspirés à construire un outil générique pour surmonter ces défis en imposant la QdS dans toutes les plateformes IdO avec une configuration facile à utiliser. L'objectif principal de cette étude est de permettre les fonctionnalités de QdS dans la couche Fog de l'architecture IdO. Les plateformes et systèmes existants permettant les fonctionnalités de QdS dans la couche Fog sont également mis en évidence. Enfin, nous soulignons la validation de notre modèle en le mettant en œuvre sur notre plateforme AMI-LAB.

Summary

The Internet of Things (IoT) can be defined as a combination of push and pull from the technological side and human side respectively. This push and pull effect results in more connectivity among objects and humans in the near surrounding environments [1]. With the growth in the field of IoT, in recent times, the risk of real time failures has increased as well. The failures are often detected by certain points of vulnerability in the system. Narrowing down to the root causes we get the point of failures and that leads to the required measures to overcome them. This creates the need for IoT systems to have a proper Quality of Service (QoS) architecture. Thus, QoS is becoming a crucial issue with the democratization of IoT. QoS is the description or measurement of the overall performance of a service, such as a telephony or computer network or a cloud computing service, particularly the performance seen by the users of the network.

In this study, we propose the methods of enforcement of QoS in IoT platforms. We will highlight the challenges and recurrent issues faced by all IoT platforms which in turn inspired us to build a generic tool to overcome these challenges by enforcing the QoS in all the IoT platforms with an easy to use set up. The main focus of this study is to enable QoS features in the Fog layer of the IoT architecture. Existing platforms and systems enabling QoS features in the Fog layer are also highlighted. Finally, we validate our proposed model by implementing it on our AMI-LAB platform.

Acknowledgments

First of all, I would like to thank my research director, Professor Bessam Abdulrazak, for giving me the chance to complete my master's degree within his team, for advising me throughout my work and for being patient and understanding with the lengths they have gone through.

I would also like to thank the AMI-Lab laboratory team for the warm welcome they gave me, their enthusiasm, and their availability for my many questions.

I thank IT support for all their support and assistance with the platform and its maintenance. I thank my family for their continual support and encouragement throughout my writing. Finally, I would like to thank Souhail Maraoui and Amin Rezaei for their support and assistance for the reflection and the scientific approach and for allowing me to participate in the publication of two papers and to submit another.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

IoT	Internet of Things
API	Application Programming Interface
REST	Representational State Transfer
JSON	JavaScript Object Notation
QoS	Quality Of Service
MQTT	Message Queue Telemetry Transport
HTTP	Hypertext Transfer Protocol
PAAS	Platform As a Service
CPU	Central Processing Unit
RAM	Random Access Memory
SAOP	Stacked Alternating Offer Protocol
VM	Virtual Machine
LSM	Limited Set of QoS Metrics
ASL	Absence of Self-learning Ability
AFT	Absence of Fault-Tolerance Mechanisms
HCR	Higher computation requirements
VPN	Virtual Private Network
AMI-Lab	Ambient Intelligence Lab

Introduction

The key component today in every technological research domain is data. Without data all machine learning concepts and AI's will be rendered useless. This requirement of a huge amount of data is one of the factors which lead to the growth of IoT. To ensure IoT success, quality of service (QoS) is needed, QoS assurance is used to mitigate limitations from equipments and network infrastructures to achieve proper functioning of all services.

Following we will introduce the context of IoT, the addressed problem, and the goal of this study.

A. Context of the Work

The emergence of the Internet of Things (IoT), which refers to the interconnected network of objects, is poised to match the historical achievements of notable technological advancements like the printing press, the steam engine, and electricity [2]. The concept of IoT is not limited to a fixed stream of industries, unlike most revolutionary changes. We already have IoT implementations in agriculture, medicine, military, engineering to even in our daily livelihoods [3]. The important factor in these implementations today is to have stable and trustworthy IoT platforms, which is ensured by a concept called "Quality of Service" (QoS). QoS guarantees the stability of the platforms and the services [4]. To understand the implementation of QoS in IoT systems, first we need to discuss the IoT architecture.

IoT architecture

A typical IoT architecture consists of four main layers (Figure 1) [5]. The lowest level of the architecture is the **edge layer**. After the edge level and moving up the architecture, we have the **Fog Layer.** The next one up the chain is the **network layer** and finally comes the **cloud or data center layer**. An older version of the IoT architecture only involves the edge layer followed by the network layer and the cloud layer and finally the application layer on top of the cloud layer for the proper visualization of the data [6] [7] [8]. The fog layer has been included in the newer architectures of IoT due to several challenges faced by cloud layers in delivering services with a higher efficiency.

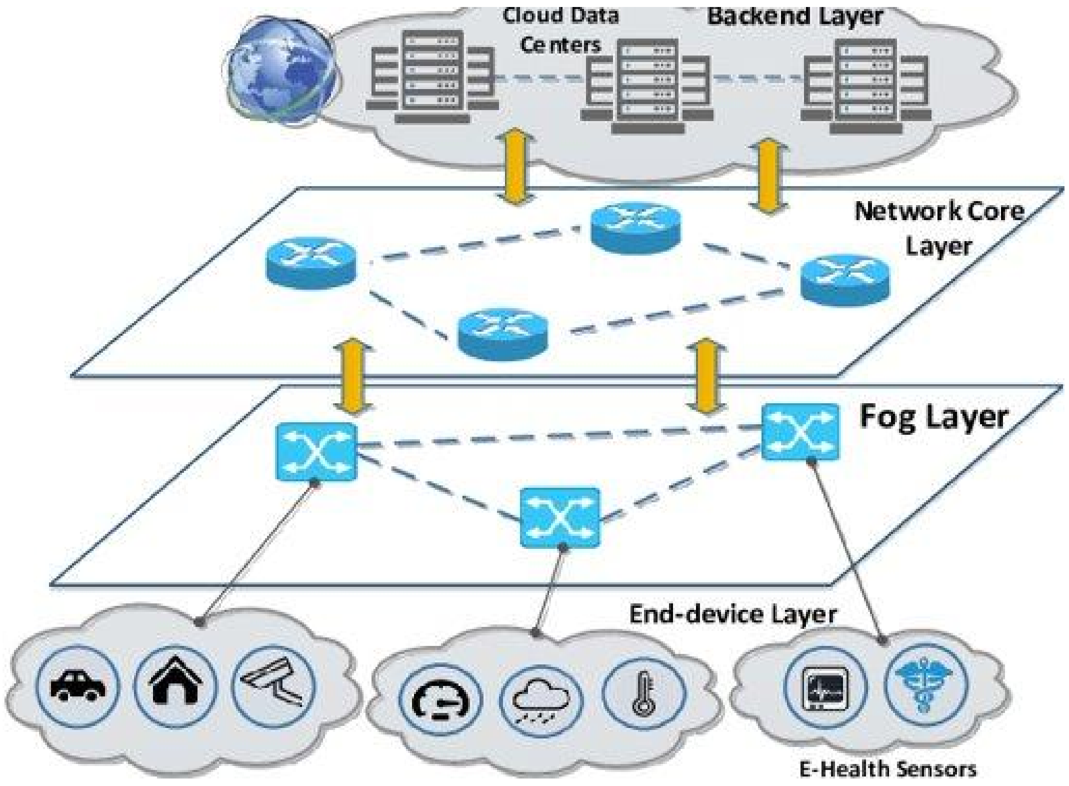


Figure 1. *Basic Architecture of IoT*[5]

i) Edge Layer

The edge computing includes a wide range of sensors, actuators, and devices [9]. These are the devices or sensors that we encounter in our day-to-day lives, devices like door sensors, smart watches, motion sensors, smart security locks, leak sensors, thermostats, etc.

In Figure 2 we can see the various key elements of an edge device layer. The “Things” layer is comprised of two parts- the sensors and the actuators.

- *Sensors*

  The **sensors** are considered as the system’s eyes and ears in the real world. They monitor environmental elements like ambient light, moisture, temperature, water, and gas leaks. The sensors although most often are considered as physical objects, they can very well be defined as anything that can be read, from files to any product-specific data. Sensors are mainly hardwired into products or devices that communicate via a short-haul communication protocol like Bluetooth Low Energy (BLE) or ZigBee.

- *Actuators*

  The **actuators** in the real world are considered as the system’s hands and feet. It affects the logical state of a product. It might include some light that can be turned off or on or some water valves that can be opened or closed. Actuations can also extend to system configurations or commands (system commands like reboot, configuration updates, etc.) that are sent remotely to a faraway place.

- *Controller*

  The next layer in the defined architecture is the **controller.** It functions as a hardware or software component that is embedded electrically with the sensors or have some logical implementation with the sensors and actuators. A controller can be as simple as converting the analog signals from a switch to digital signals which will show us the status of the bulb at any point of time. The controllers can even have some local communications in between sensors via simple serial connections or low energy wireless communications.

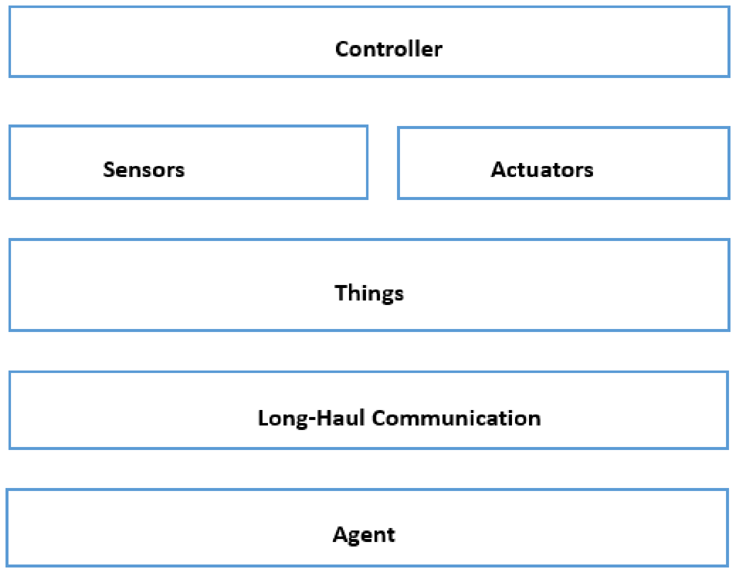


Figure 2. Architecture of edge layer

- *Agent*

An **agent** can be defined as an embedded program that has been implemented on the IoT devices or is in a device close to the IoT devices. This agent reports the status of the services and the environment. It acts as a bridge between the controller and the Fog level regulating the amount of data to be sent and the time to send the data. Even this agent acts as a mediator to all commands and requests that come from the cloud or Fog layer as it processes all the commands before the actuation.

- *Long-haul Communication*

The final component of the architecture is the **long-haul communication**. All IoT solutions require the data collected from the environment to be transmitted to the cloud, now there comes the challenge as this involves a lot of small components like security, footprint, and reliability. We have a wide variety of options for long-haul communications like cellular, Wi-Fi and wired internet along with some sub gigahertz options like LoRa and Sigfox. Most of these options use the common networking protocols like TCP (Transmission Control Protocol), UDP (User Datagram Protocol) for the transport layer and HTTP (Hypertext Transfer Protocol) and CoAP (Constrained Application Protocol) for the application layer.

ii) Fog Layer

The fog computing was first proposed by Cisco [10] to solve the applicability issue of Paas (Platform as a service). Fog computing addresses the limitations of cloud computing. Initially the data after being collected from the environment was directly transmitted to the cloud but as the cloud transmissions consumes a huge amount of network bandwidth it became very difficult to process all data and provide faster responses on a low budget. The fog layer is comprised of fog nodes, essentially functioning as industrial controllers, gateway computers, switches, and I/O devices that offer computing, storage, and connectivity services. The fog computing model expands the cloud's reach to the network's edge, where devices are located, and promotes edge intelligence. The fog computing environment is different based on the use case. The fog can be held tightly coupled with the edge using only a few industrial controllers or on the other side can be closer to the cloud having data centers acting as fog. In our study we focus on the fog layer closer to the edge consisting essentially of industrial controllers and other gateway computers with limited computational resources.

In Figure 3 we see the functioning of a typical fog layer: the sensors send the raw data to sink nodes and the sink nodes creates tasks which are pushed into the task queue of the fog layer. Then comes the service manager in the fog layer which has the logic to schedule the tasks and put them into the fog nodes to get executed [11].

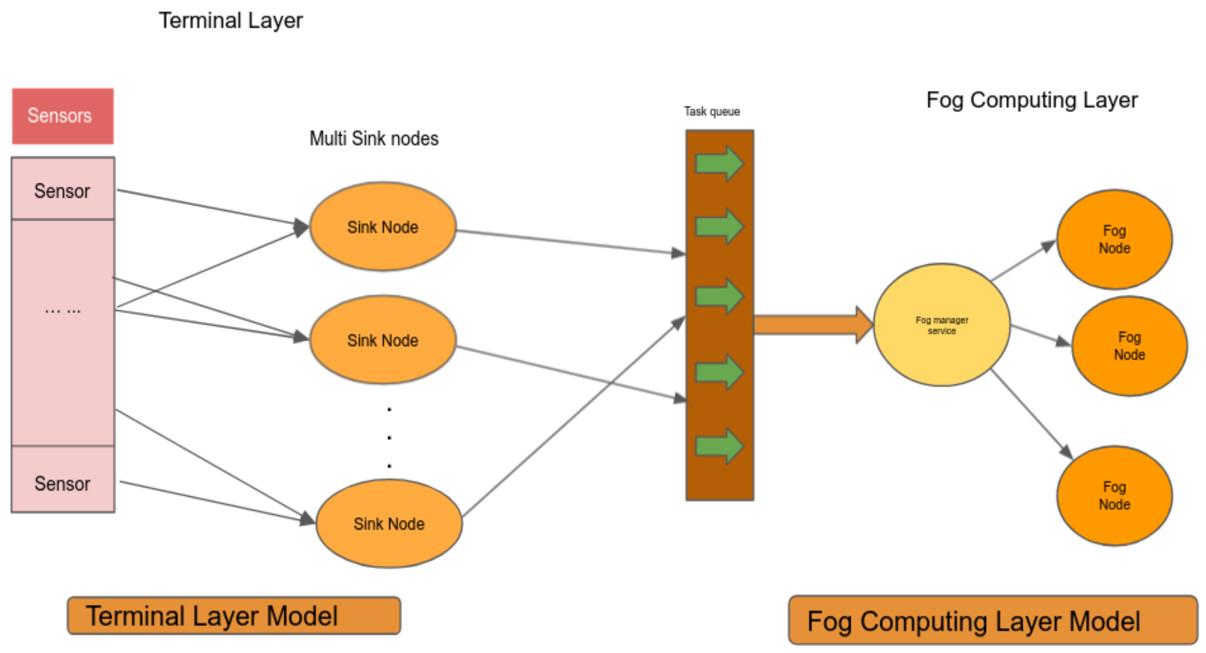


Figure 3. Architecture of fog computing model

iii) <u>Network Layer</u>

The network layer acts as the bridge between the fog layer and the application layer. The data from the fog layer is transported to the application layer via several protocols. Based on the protocols defined in the IoT sensors or devices, the communications can be both wireless and wired. The network layer has the following main functionalities- Routing and Addressing, Network Capabilities, Transport Capabilities, Error detection and Correction and data packet routing. To serve different IoT services, different networks are required with different protocols. These networks can be private, public, or hybrid models to assist with the latency, bandwidth, or security requirements. The choice of the network protocols depends on the power consumption of each node, the transmission speed needed for particular applications. In the table below (Table 1), we see the most used protocols for communication [12].

Table 1. Main IoT Protocols

Application Layer	Transport Layer	Network Layer	Adaptation Layer	Data Link Layer
MQTT. CoAP, AMQP, XMPP, DSS, DNS-SD, SSDP, mDMS.	TCP, UDP	Routing: RPL, CORPL, CARP. Addressing: Ipv4/Ipv6	6LoWPAN, 6TiSCH, 6Lo.	Zigbee, Bluetooth, LPWAN, RFID, NFC

iv) <u>Application Layer</u>

Based on the information provided by the middleware layer the application layer is required to manage all application services and processes. The application layer does not have any particular standard of implementation, but it is able to offer a variety of services in different fields of society. Like we have IoT services in industries, healthcare, military, smart grids, agriculture, and many others. The security requirements in this layer are dependent on the environment and the type of industrial applications.

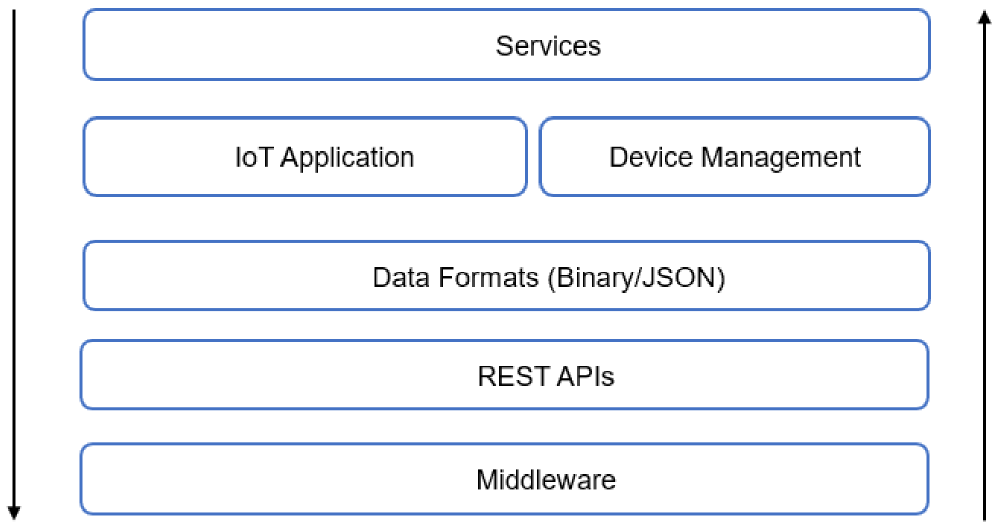


Figure 4. Application layer components of IoT

We see the components of the application layer in Figure 4. The data or information collected from the middleware is transported to the application layer (e.g., through REST APIs). The data format is usually in binary or JSON encrypted or in bytes. In the application layer, we decode the encrypted data which is to be used for several IoT applications as discussed earlier, apart from providing services to the outside world the information collected can also be used to check on devices and nodes to ensure their proper functionality. Some services are backwards compatible, like based on the user's requirement the application layer can send data back to the edge layer to perform certain actions, like turning the lights in a room on or off or changing the temperature of a room and similar tasks.

Following the discussion of the IoT architecture and IoT platforms, we can define the concept of QoS as the ability of IoT systems and its different layers to sustain a reliable flow of the needs or services corresponding to the requirements of the business applications.

With this context in mind the AMI-Lab team from the University of Sherbrooke, with its expertise, developed an IoT platform, known as AMI-Platform, which promotes better delivery of services in Smart Cities [13]. It targets helping older adults to lead an independent and purposeful life, through ambient assistive technologies. AMI-lab has developed a reliable, secured, and scalable IoT system with Plug and Play architecture. Through the deployment and the use of the AMI-Lab platform, several problems related to IoT QoS were raised.

B. Problem Statement

The IoT architecture portrays the link between each layer and the data flow pipeline in between these layers. During the process of data transfer across these layers, numerous problems (such as loss of packets, fake packet transformation, changes in packets) may be encountered (as also experienced in the previous version of AMI-Platform) which are a result of lack of QoS implementation. Several articles in the literature highlighted these problems. Following, we will be discussing the most discussed problems at each layer of the IoT architecture as well as the metrices to create a Quality of Service (QoS) based system.

- *Edge Layer:*

  The edge layer comprises devices and sensors which comes with **limited resources**. Hence, one of the biggest challenges faced today is with the **resource optimization** (for example battery, CPU and RAM usages) at the edge layer. The resource management comes hand in hand with the **cost effectiveness** of the edge system. The next set of challenges comes with the compatibility of the different types of edge sensors and devices with one another. IoT sensors are built with different types of communication protocols and these heterogeneous nature raises a challenge for the **intercommunication** in between sensors with two different types of communication protocol.

- *Fog Layer:*

  The fog layer comprises the computing nodes. The first set of challenges that we come across is the **compatibility** of the sensors and the fog nodes. The fog node being able to handle all the different types of communication protocols is a huge requirement. After which comes the **resource management** issue followed by **data security** and **transmission issues** in the network layer.

- *Network Layer:*

  The network layer mainly deals with **data security** and **integrity aspect**. Along with that it also deals with the issue of **network bandwidth** and **packet losses**. The bandwidth requirement brings the cost factor as with the increase in network bandwidth, the price of the IoT system will increase as well. The challenge in this layer is to have a proper balance of **cost** and effectiveness of the system.

- *Application layer:*

  This layer is the user perspective layer, the data after being processed is being provided in the form of specific user needs. It could be displaying the data in dashboards to raising alarms for specific situations and other types of user-specific services. The challenge here is to be able to cater to all the user needs in a **real time** and **reliable format**.

Discussion

Among all the issues discussed, the issue of QoS has been widely addressed specifically in the network layer. However, it needs to be reconsidered for the all the layers of IoT, especially on the ground level, the edge layers and fog layers. Hence, an IoT platform needs to have a well-defined QoS implementation targeting all the metrices and challenges and providing an overall reliable system at all layers. Nevertheless, QoS in fog layer, where the QoS constraints are mainly related to resources and data security, is less addressed in the literature. The solutions provided in the literature for the fog layer has restrictions in terms of scalability as the majority of the issues are related to resource constraints.

One of the solutions to handle the resource constraint QoS metric in the fog layer is to have a distributed fog system. The emergence of distributed frameworks and computation offloading architectures is on the rise to counter the singular node fog layer QoS constraints [10]. However, the limitation of resources at the fog layer limits the use of existing frameworks and architectures for cloud layers (where any resources can be scaled dynamically on demands). The cost of such scalability(due to extensive addition of resources in the fog devices) becomes significantly high when it comes to fog layer [10].

With this context, we focus in our work on two problem statement issues : 1) QoS constraints in fog layer and 2) the enabling of distributed QoS computing in fog layer.

C. Study Aim

The aim of the study is to propose a stable and optimized IoT architecture with a special focus on distributed fog system. The underlying goal is to build a QoS based fog system that enables an IoT platform to have a reliable and scalable service flow, and this functionality is then extended to a distributed framework. The idea is to provide generic and scalable tools to implement QoS features at the different layers of IoT. Our main targeted layer is the Fog layer in IoT platforms. We attempt on creating multiple components that are generic and dynamic, at the fog layer. The goal is further divided into 1) enabling QoS in a single component Fog system and 2) enabling QoS in a distributed Fog system.

1) Enabling QoS in a single-layered fog system: we propose several small QoS enabling components in the single-layered Fog system which can be used in any IoT platform at any granular level to target a specific QoS feature and support it. For example, a component specific for monitoring of running services.
2) Enabling QoS in a distributed fog system: we propose a dynamic distribution model along with an engine supporting the model to provide an easy-to-use tool for large-scale QoS implementation.

We have analyzed in our literature review the various QoS metrics that are required at each of the IoT layers. Then we have designed our approach in building stable QoS oriented IoT platforms. Based on our design we have divided our approach in the first stabilizing single layered fog system and then QoS based distributed fog layer.

D. Results

The end point of this research work is a QoS based fog system which showcases the proper implementation of QoS metrices in the fog layer of any IoT platform and the scalability aspect of this architecture. The implementation of the QoS metrices has been extended to include the distributed nature of the fog layer, which enables the system to address the QoS constraints in a much larger scale along with a proper QoS based management system. The developed distributed model is not attached to any particular IoT platform implementation and is capable of adapting itself to different IoT systems like IoT system for agriculture, IoT systems for industries or medical IoT systems, etc.

The proposed QoS based fog system with its QoS metrices has been implemented in the AMI-Lab platform, which has been used as a testbed for our validation.

E. Presentation of the Document

The document is organized as follows:

In Chapter 1, we discuss the literature review of QoS in IoT. We examine the concept of QoS in depth and then move on to the representation of the problems associated with QoS, the QoS metrices that are key aspects to all IoT platforms. After that, the existing QoS approaches in IoT are discussed.

In Chapter 2, we present our design approach for QoS in fog layer. This chapter begins with detailed discussions on the internal components of the fog layer, followed by a section of lessons learned from a real-world AMI-Lab deployment. Continuing on that, we highlight the major point of failures in the fog layer providing us the pathway to discuss about our approach in overcoming the point of failures. Next, we portray our approach which includes the sub-topics of autonomic computing and the concepts of agents. Lastly, we discuss the support for QoS in a distributed fog layer environment by presenting our distributed model.

In Chapter 3, we present the technical evaluation of our approaches. We start by discussing the set of principles used for QoS evaluations in distributed models. Following that, we describe tests and results for QoS agents in fog nodes and also for services provided by distributed model.

Finally, we have the conclusion and a section for future works.

Chapter 1 Literature Review

The work presented in this document is related to the implementation of QoS concept in IoT platforms in a generic and dynamic way, with a focus on the Fog layer.

Therefore, this chapter of literature review addresses the two aspects: 1) the QoS definitions and metrics that are needed to be targeted at each level of an IoT platform; and 2) the different approaches proposed to address the issues regarding QoS in IoT.

The QoS metrices were developed as a result of failures in real time IoT platforms which were caused by vulnerable points in the system. The vulnerable points are used to specify the exact point of failures. This point of failures gave rise to requirements for the IoT systems and these requirements either individually or collectively are defined as QoS metrices (Figure 5).

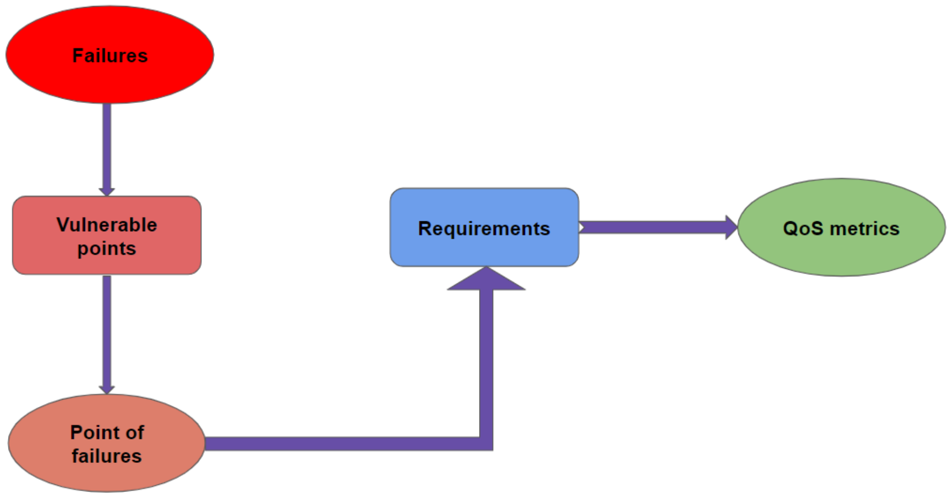


Figure 5 Origin of QoS metrics for IoT platforms

1.1 Quality of Service (QoS)

Quality is the expected product/service being realized. However, quality is a function of how the customer views the product or service that he or she receives [15]. In IoT platforms or services we use this same concept to meet the specific requirements of the customers and the service providers. QoS in IoT can be divided into mainly three components based on the layers of architecture as shown in Fig.1. QoS is needed to manage and help maintain the system functionalities and to have proper resource regulation to provide the optimized services. It paints a clear picture of how the services will function, their performance and the usability of the services to the consumers. The major challenge faced by us today is in maintaining the QoS in the Networking layer of IoT. However, to have an efficient system we need to have a QoS in each layer complementing the other layer QoS. In this thesis we will discuss the various QoS metrics in each layer of the IoT platforms. In addition to the QoS metrices we will discuss another **Multidimensional QoS** [16] where QoS of an application can be thought of as a space of n dimensions. Every kind of QoS parameter is one dimension which makes up the space. Most contemporary QoS prediction methods make use of the QoS characteristics for one specific dimension, e.g., time or location and does not take into account the other structural relationships among the various QoS data.

Next, we will discuss the QoS metrics in IoT layers and provide detailed descriptions and measurement criteria.

1.2 QoS Metrics in IoT

The following metrics are considered vital for the measurement of performance of smart IoT platforms: As mentioned above for an effective IoT system, it is important to have QoS in each layer to support the QoS in the other layers, so we will discuss the metrics in each layer of an IoT system with proper descriptions and measurement criteria.

1.2.1 Network Layer

The network layer is one of the most important layers in IoT in which researches have brought forward a number of QoS metrics that is defined in the table below with measuring criteria for each metric.

Table 2. QoS metrics in the network layer

Metrics	**Description**	**Measurement Criteria**
Jitter	Jitter metric is a quantifier of the changeability over time of the packet latency across a network and can be a measurement for the quality of a communication (like a voice or video call): a zero jitter shows a communication without any variation in latency.[17]	It is measured by the average of the deviation from the network mean delay. This is often caused by network congestion, and sometimes route changes.
Bandwidth	The maximum amount of data transmitted over an Internet connection in a given amount of time [4][6].	bits per second (bps)
Throughput	Throughput or network throughput is the rate of successful message delivery over a communication channel [18] [19].	bits per second (bit/s or bps) or data packets per time slot.
Packet Loss	The loss in data while transmission [4], [6].	Amount of data lost while data transmission over a network can be caused by various reasons like - congestion, traffic, etc.
Packet Delay	It is a complex metric as it includes all the delays at different stages of packet transmission [19].	Total delay includes delay caused on the hardware level, delay on the computation or software level and the delay in network transmission level.

Table 2. QoS metrics in the network layer

Metrics	**Description**	**Measurement Criteria**
Efficiency	The channel efficiency is used to represent the utilization ratio of the channel and the normalized throughput in a given duration.[20]	percentage (%)
Network Connection Time	The time taken by devices (things) to connect to the network.[6], [21]	seconds(s)
Monetary cost	Cost of the communication network is an important aspect which needs to be taken into consideration while planning for an IoT project [6].	Money
Availability	The network active time denotes the availability of all IoT devices.[21]	Alive time of the network(s)
Security and Privacy	The security and privacy of data are probably the most important QoS metric that needs to be addressed for any IoT-related development [22][21].	How much less prone the network is to the outside environment.
Interoperability	"The ability of two or more networks, systems, devices, applications or components to exchange information between them and to use the information so exchanged" [23], [24]	Flexibility of the network with other IoT platforms or services.
Service Level Agreement (SLA)	SLA is an agreement between a customer and the service provider which defines the basic performance levels or how reliable the platform will be up to a certain level. This agreement has to be maintained by the service providers at all costs. [25]	SLAs guarantees help desk problems, resolution time, or guarantees on service outages.

Table 2. QoS metrics in the network layer

Metrics	Description	Measurement Criteria
Monitoring	A proper and regular monitoring of networks and devices must be maintained [21].	Frequent updates and proper regulation of the network.
Reliability	Network reliability is an important aspect in IoT which should always be guaranteed [26].	Provides proper network connection for a long period of time.

1.2.2 Fog Layer

The Fog layer challenges led to the development of QoS metrices that needs to be addressed in any IoT platforms or systems to be developed. These metrics are described below.

Table 3. QoS metrics in fog layer

Metrics	Description	Measurement Criteria
Latency	The delay in time taken for the data to reach the destination across the fog network [5].	Measured in milliseconds (ms)
Bandwidth	In a fixed period of time the amount of data or information that can be transmitted [5][10].	Measured in bits per second (bps).
Scalability	The property of systems to be economically deployable at any range of sizes and configurations [19], [27].	Measured by the effectiveness of the system in variable size of deployments. The effectiveness can be measured by the systems in general quality of service and its throughput.

Table 3. QoS metrics in fog layer

Metrics	**Description**	**Measurement Criteria**
Offline capability	The ability of the system to perform when it is disconnected from the Internet network [29].	Measured by the uptime of services which are initially provided by the system. The amount of data loss is also a measure of this metric.
Security	Security of data and information is a vital metric in the fog layer similar to the network layer [19].	Measured by the ease of a data breach in the systems.
Reliability	The probability of a system to perform accurately during a given period of time	The system is able to perform without any change or modifications.
Availability	The system's ability to be accessible at any time whenever required and is not in a failed state or undergoing maintenance.	The uptime of the system in the course of a deployment.
Stability	The system's ability to have a consistent performance over time period.	Measured by system's overall usability and consistency.

1.2.3 Edge Layer

The edge layer deals mainly with sensors and devices. The challenges thus faced are different from the other layers, it mainly involves hardware-related issues which brings forth the following QoS metrics as described in the table below.

Table 4. QoS metrics in the edge layer

Metrics	**Description**	**Measurement Criteria**
Weight	Weight of sensors and devices are an important factor	Weight = kgs (general unit) Less the weight the better the solution.

Table 4 QoS metrics in the edge layer

Metrics	Description	Measurement Criteria
Interoperability	It is an open standard for all the devices and sensors which enables the following properties in between the sensors to networks and other applications: interconnection, discovery, access, integration, and usage. [24]	How efficient the sensors are to communicate with other types of sensors.
Flexibility	Devices should be flexible to adjust to needs at times.	Device flexibility is measured by its performance in different environments.
Availability	Devices should have energy and resources to always incorporate data [21].	Most sensors run on batteries and other sources of electrical energy; this metric is the measure of how long the devices can work without intervention.
Reliability	Devices should be able to self-configure in a changing environment to send correct data [26].	Reliability of sensors in different circumstances should be tested and checked before use.
Overall Accuracy	Sensor's accuracy can be defined as the maximum uncertainty between the actual value measured and the standard value set at output parameters [4].	The data coming from sensors should be accurate as it might have severe repercussions.
Long-Term Stability	Stability can be defined as the consistency in output produced by sensors over time-period [30].	Consistency in output provided by the sensors.
Response Time	Time taken by devices to respond to a call or a service requirement [19].	Sensors with low response time are best suited for any application.
Range	Range of sensors should be high enough to maintain proper connectivity[31]	Range of communication of sensors and devices should be high enough.

1.2.4 Application Layer

The application layer is the layer that provides the user with the desired results after the computation of all the information gathered from different IoT devices. The challenges faced in this layer mostly focuses on the computation methods and the performance of the algorithms used.

Table 5. QoS metrics in the application layer

Metrics	Description	Measurement Criteria
Scalability	Computing applications defines scalability as producing maximum throughput in minimum response time[6], [27]	Measure as a property of a system to handle a growing amount of work by adding resources to the system.
Dynamic Availability	It is a quality parameter which tells whether the system is accessible or not when required for use, under normal operating conditions [32].	All services should run seamlessly to provide output whenever required.
Reliability	The system probability of producing correct outputs for a particular period of time. [31]	Measured by the consistency of system performance.
Response Time	Response Time is the time interval between request submitted and the service responded [19].	Lower the response time of the services the better the computing solution.
Capacity	Capacity is the measure of the maximum amount of computing resources provided by the computing service provider that can be processed and analyzed by the computing software [32].	Capacity of resources.
Security and Privacy	Ensuring security in computing involves the confidentiality and the integrity of the data being guaranteed at the computing node [33].	Maintaining a high level of security at the computing level is needed to guarantee personal privacy.

Table 5. QoS metrics in the application

Metrics	**Description**	**Measurement Criteria**
Customer Support Facility	Customer Support is the services provided by the vendors in case of any fault or discrepancies in the system [4].	Time taken to detect a failure and respond to bring the service back online.
User Feedback & Reviews	An individual's experience and review play an important role in the selection of computing services[34]	Feedback forms and surveys.
Age of Information	In today's world, we need real time data in almost every field: medical, military, vehicular and others where data is supposed to be very time sensitive. It is for this reason that the age or the timestamp of the data is so crucial [35].	Timestamp of each data is very crucial to maintain its accuracy.
Reputation	It is the customer or user-level satisfaction on using the particular services.	User level of satisfaction.

1.3 QoS Approaches in IoT

An IoT platform needs to have a well-defined QoS implementation targeting all the metrices and challenges and providing an overall reliable system at all layers. In this field, there are only a few articles in the literature targeting approaches taken in the field of IoT to establish a well-defined IoT platform with the satisfaction of QoS metrices. These architectures have their own advantages and disadvantages at managing a QoS enabled IoT system.

1.3.1 QoS Broker and Feedback approach

The broker and feedback approach is a very simplistic way of sharing the information amongst each layer. There is a QoS broker in each layer which enables smooth transaction of data in between them [34]. The brokers have a direct link to the QoS management facility which oversees the overall communications.

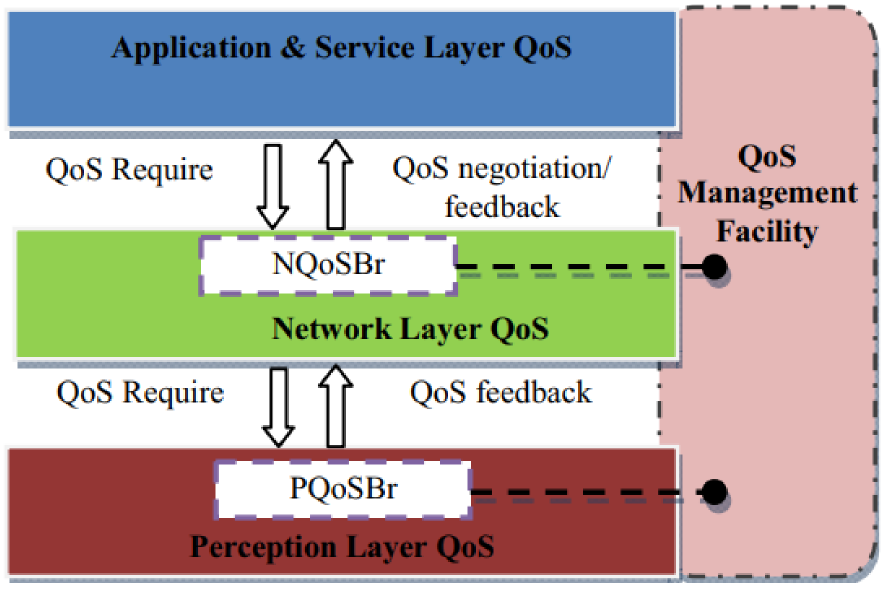


Figure 6. QoS Architecture using a broker and feedback approach.[29]

This architecture has three main components- QoS of each layer, QoS brokers and QoS Management Facility. **QoS of each layer** denotes their respective QoS metrics that need to be resolved to have a high performance. **QoS brokers** in the Network layer and Perception layer are responsible for resolving the QoS requirements from the upper layers. **QoS management** was designed as a common cross-layer facility for supporting QoS operations in all three layers.

Table 6. Application types and applying QoS Level. [29]

Application type	I control	II query	III real-time monitoring	IV non real-time monitoring
QoS level	Guaranteed Service	Guaranteed Service /Differentiated Service	differentiated service	Best Effort

Control Mechanism

The upper layers call the brokers in the lower layer with four following arguments-

L – Least time delay accepted.

P – Parameters to send to lower layers.

R – Service priority

T – Application Type (this type denotes whether the service is real time or not whether the QoS needs to be guaranteed or not) (Table 6).

The application layer queries initiates a communication with the network layer with a set of arguments and the network layer redirects the requirements to the perception layer. So, the lower layer broker on receiving the arguments translates them to QoS requests which are then fulfilled by that current layer. If the QoS requirement cannot be satisfied, then the broker is used to negotiate with the upper layer. Finally, the QoS feedback is sent to the upper layer. In this way, the QoS is guaranteed in each layer.

Discussion

This approach focuses on the simple feedback technique where in each layer of the IoT gets feedback from the other layer on the validity and the scope of the QoS metrices and the QoS management system is then used to manage the negotiation process. So, in the end it is able to implement the QoS metrices in all the layers.

As a drawback of this architecture, we can see four different types of QoS levels which regulates the services functionality. The QoS negotiations are done in between the different layers so it targets the metrices that are dependent on the layers for example, bandwidth requirements, latency factors, packet loss ratios, etc. However, as we have seen in the above section (chapter 1.2) each layer has its own separate sets of QoS requirements which need to be satisfied irrespective of the requirements of the other layers of IoT. This model does not incorporate this aspect. There is no mention of any of the self-learning model which can help the system to not compute the same types of QoS scenarios in different IoT systems. So, scalability is not taken into consideration.

1.3.2 IoTQoSystem with QoS Client and Server approach.

The IoTQoSystem approach is based on the negotiation process between the client and the server. The basic requirement of such negotiation models are high success rate and fast negotiation outcomes, dynamic support for changes in parameters. This architecture is incorporating a similar negotiation model with three elements- the QoS parameters, the negotiation protocol, and the strategy [36]. The QoS parameters include the technical and non-technical attributes, for example, response time, availability, throughput is included as technical parameters while price penalty and reputation are included in the non-technical section. In the negotiation protocol, we have a set of rules that is defined to facilitate the negotiation process between the client and the server. If after a series of offers and counteroffers between the client and server a favorable outcome is reached, then it's a success else a failure. The strategy includes the creation of the decision rules, the rules which are set to accept offers or decline the offers are made in the strategy phase. Thus, this approach incorporates these implementations to provide a QoS based system.

The two main components of this architecture are the IoTQoSystem Client and the IoTQoSystem Service (Figure 7).

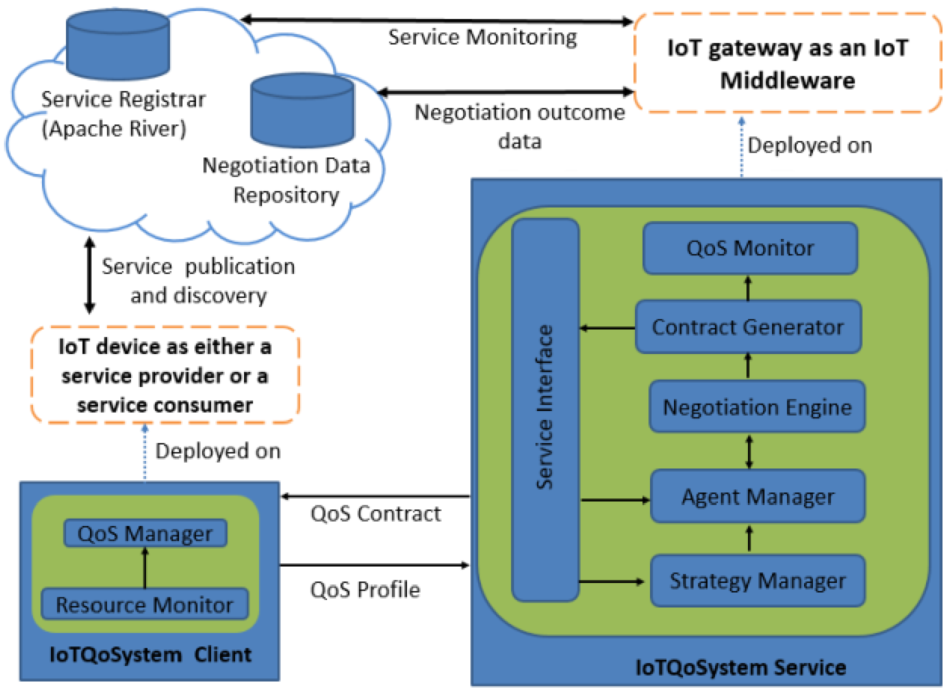


Figure 7. QoS Architecture using client and server approach.

A) IoTQoSystem client

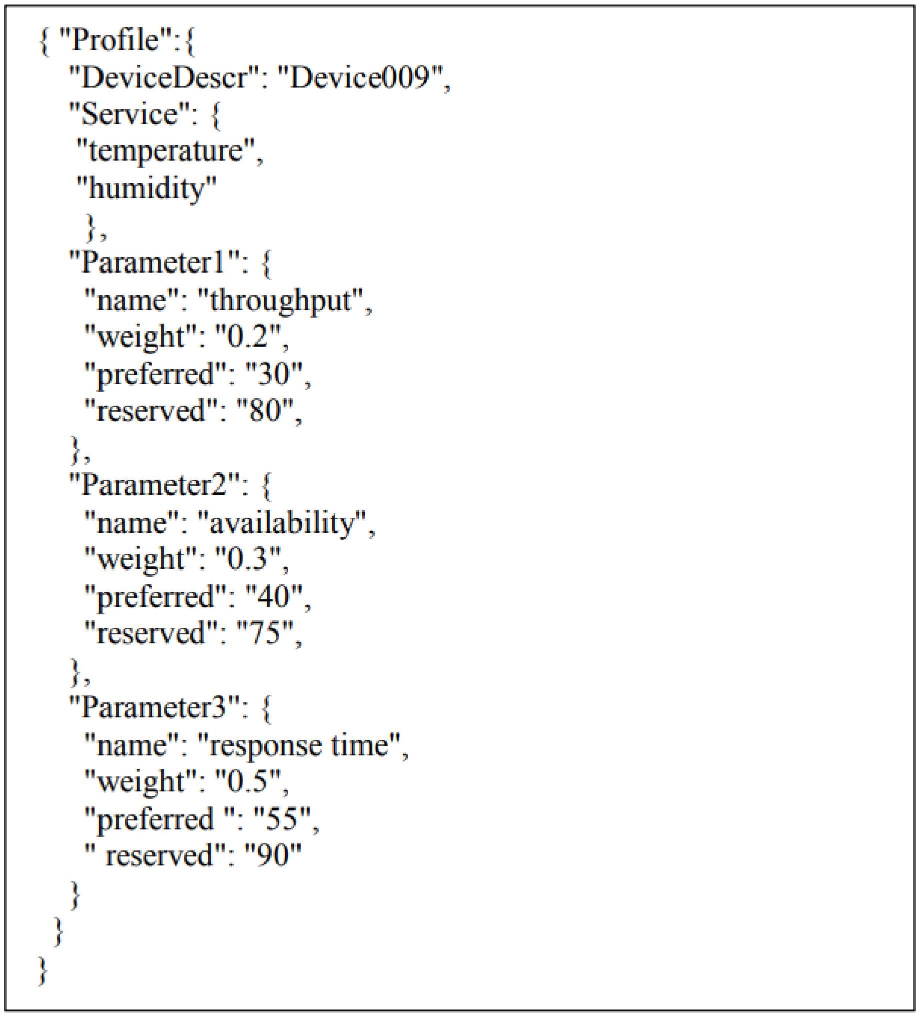
This client is essentially responsible for providing the most appropriate QoS profile to initiate a negotiation process. The client on the service provider monitors the device resources. It has two components of its own:

a) Resource monitor:

Every device or sensor in the real world is dependent on some external resource such as battery or internet connectivity. So, the resource monitor is used for monitoring of such external resources which are critical for the service provider point of view.

b) QoS Manager:

The QoS Manager oversees providing the Service component the accurate QoS profile (Figure 8). The service consumer selects one profile, and the service provider updates that profile. The QoS Manager then sends them both to the service side.



```
{ "Profile":{
   "DeviceDescr": "Device009",
   "Service": {
    "temperature",
    "humidity"
     },
   "Parameter1": {
     "name": "throughput",
     "weight": "0.2",
     "preferred": "30",
     "reserved": "80",
   },
   "Parameter2": {
     "name": "availability",
     "weight": "0.3",
     "preferred": "40",
     "reserved": "75",
   },
   "Parameter3": {
     "name": "response time",
     "weight": "0.5",
     "preferred ": "55",
     " reserved": "90"
   }
  }
}
```

Figure 8. QoS Profile.

B) IoTQoSystem Service

IoTQoSystem Service manages the negotiation process. It generates the QoS agreement and monitors the QoS as the service starts running. It is also responsible for the coordination of the negotiation agents and provides an interface through which IoT devices can submit their QoS profile for negotiation. The subcomponents are as follows:

a) Service Interface:

In this module the QoS profiles from the client component are received which are then authenticated and validated. The validation is done to see if there is some discrepancy in the two profiles and if so then the client side is informed to take necessary actions.

b) Agent Manager:

Based on the received QoS profiles the agent manager is responsible for generating negotiating agents. The agents are used to define IoT device preferences to drive the negotiations. The work of [19] is adapted to develop the agent manager for the IoTQoSystem service.

c) Strategy Manager:

The strategy manager is used to bind each agent with the correct strategy initial parameters and implementing the decision models.

d) Negotiation Engine:

The negotiation engine is in charge of providing a negotiation solution using the Stacked Alternating Offer Protocol (SAOP) [37]. It simulates the negotiating agent's behavior based on their specified MDS[38] parameters. After a negotiation session, all the data, such as the total number of offers, the strategy parameters involved and the completion time of negotiation, are stored in the negotiation knowledge database. This data will assist the strategy manager in making informed decisions when defining the initial optimum MDS parameters for subsequent negotiations. If a negotiation process fails, it requests the agent to remodel their negotiation strategy; otherwise, it sends the negotiation outcome to the contract generator.

e) Contract Generator:

This generator comes in place when a potential negotiation solution is identified, then it translates the resulting solution into a binding agreement that gives the contract between the service consumer and service providers.

f) QoS Monitor:

The QoS monitor oversees constant monitoring of all negotiation services. In case of any failures a renegotiation service is called.

Discussion

For the IoTQoSystem approach, the advantage is that it has a good defined semantics for the QoS profile. We can define a huge range of metrices with the preferred value along with the weight of that metric and the reserved weight. This allows for good negotiation process along with the enhanced scalability of the architecture compared to the QoS Broker.

The major drawback of this approach is again the lack of any self-learning component and thus every time the systems encounter an issue, they require the user's intervention to validate. Certain metrics for a specified system design will have the same values which the system itself can learn from past negotiations and update itself without having to go through the negotiation process. Apart from the self-learning aspect, we see that this method would require a lot of computation power to implement. As we know, the fog layer of any IoT platform has very limited resources so in order to implement such complex modules it would bring forward a significant amount of overhead to the system overall. This approach does not provide any management system for the deployed services after negotiations, no fault tolerance mechanisms to proactively restart negotiations in case of some run time changes to the system.

1.3.3 QoS estimate approach based on multidimensional QoS.

QoS of an IoT platform can be thought of as a space of n dimensions. Every kind of QoS parameter is one dimension which makes up the space. Most contemporary QoS prediction methods exploit the QoS characteristics for one specific dimension, e.g., time or location and do not exploit the structural relationships among the multidimensional QoS data. This approach enables the implementation of QoS in a multi-dimensional aspect [30]. The dimensions and the targeted QoS parameters are described in the table below.

Table 7. multi-dimensions and parameters.

Dimension	**QoS Parameters**		
Sensing Dimension	Accuracy	Availability	Stability
Transmission Dimension	Transmission Time	Storage Capacity	Reliability
Application Dimension	Functionality	Normative	Robustness

There is a division in the type of QoS parameters, the positive attributes, and the negative attribute. The parameters such as accuracy and availability are considered to be positive attributes as the bigger the QoS attributes the better QoS it has. On the other hand, we have transmission time or storage capacities which have better QoS if its QoS attributes are smaller. The flow architecture of this approach includes the following steps acquiring QoS parameters, normalizing the parameters, mapping the QoS in the three-dimensional space, calculating the Euclidian distance and estimating the QoS of the IoT application (Figure 9).

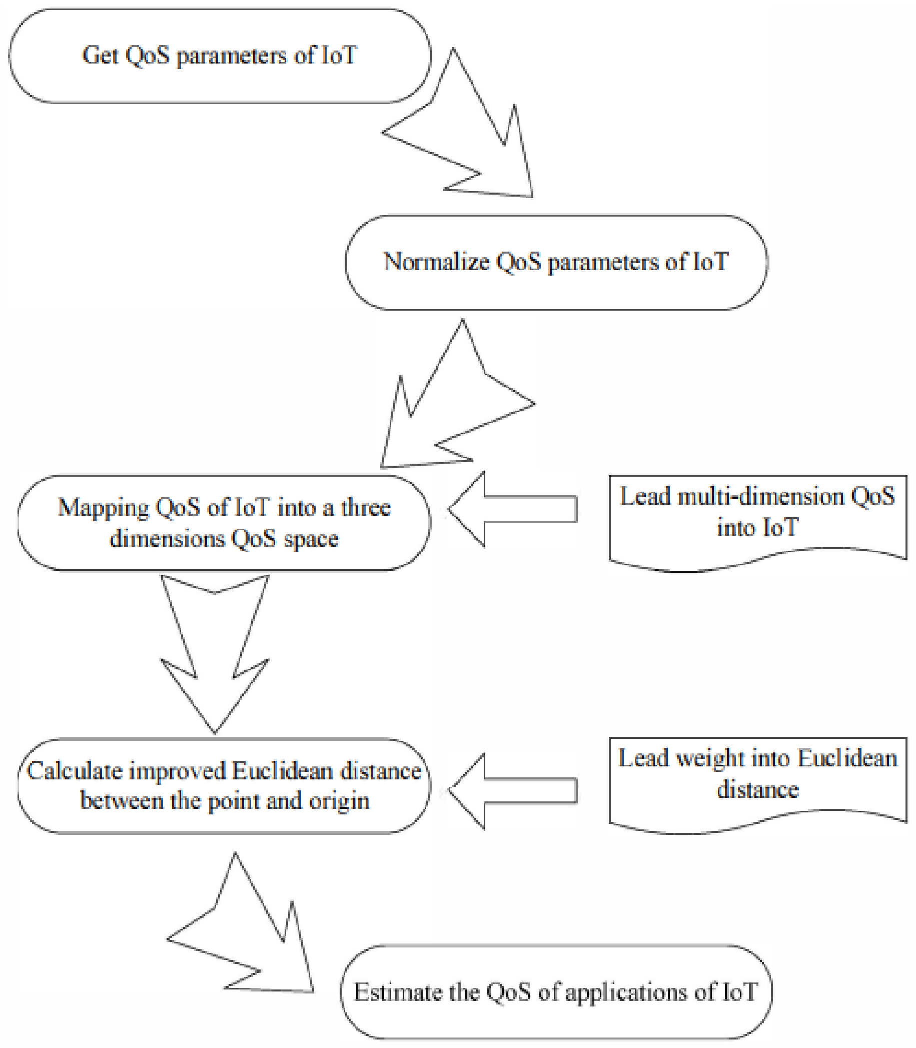

Get QoS parameters of IoT
Normalize QoS parameters of IoT
Mapping QoS of IoT into a three dimensions QoS space
Lead multi-dimension QoS into IoT
Calculate improved Euclidean distance between the point and origin
Lead weight into Euclidean distance
Estimate the QoS of applications of IoT

Figure 9. Flow architecture of QoS estimate approach on multidimensional QoS

The normalizing of the QoS parameters is done to map QoS attributes of every dimension into a defined range of 0 and 1, this enables them to have a similar growth direction. After the parameters are mapped into the different dimensional layers based on Table 6, the Euclidian distance is calculated with an improvement on the initial formula. The Euclidian distance is capable of calculating distance between points in a multidimensional space but does not consider the weight of each dimension, so an additional weightage is added to the formula. Let, T (t1, t2, t3) be a point in space of three dimensional QoS and their weightage could be represented as k1, k2, k3 where in the summation of k1, k2 and k3 is 1. So, the improved formula is- $\boldsymbol{L} = \sqrt{\boldsymbol{k_1} * \boldsymbol{t_1^2} + \boldsymbol{k_2} * \boldsymbol{t_2^2} + \boldsymbol{k_3} * \boldsymbol{t_3^2}}$. With this estimation approach each of the attributes would have equal weight in the selection of services and applications which needs to be met under specified constraints.

Discussion

The QoS estimation approach discussed here is quite advantageous and yielded highly accurate outputs with the selected range of attributes used in the defined model. However, the major drawback of this approach in the current field of IoT is that the dimensions are more than the mentioned three and the number of targeted parameters are more than what has been taken into consideration. The lack of dynamicity to include more attributes makes it specific for a particular application and cannot be used for scalable projects. The approach can be modified with proper testing on multi attribute systems to generalize a formula which can then be used as a general QoS estimation model.

1.3.4 QoS aware scheduling of services in IoT

The three layered IoT architecture is addressed in this approach [39]. In the application layer the QoS optimization is done by modeling the problem into a Markov decision process. The network layer optimization is done by minimizing the connection cost and other network layer metrices. In the edge layer, the objective is to obtain the resource usages by the edge nodes and to optimize the cost of the resource usage.

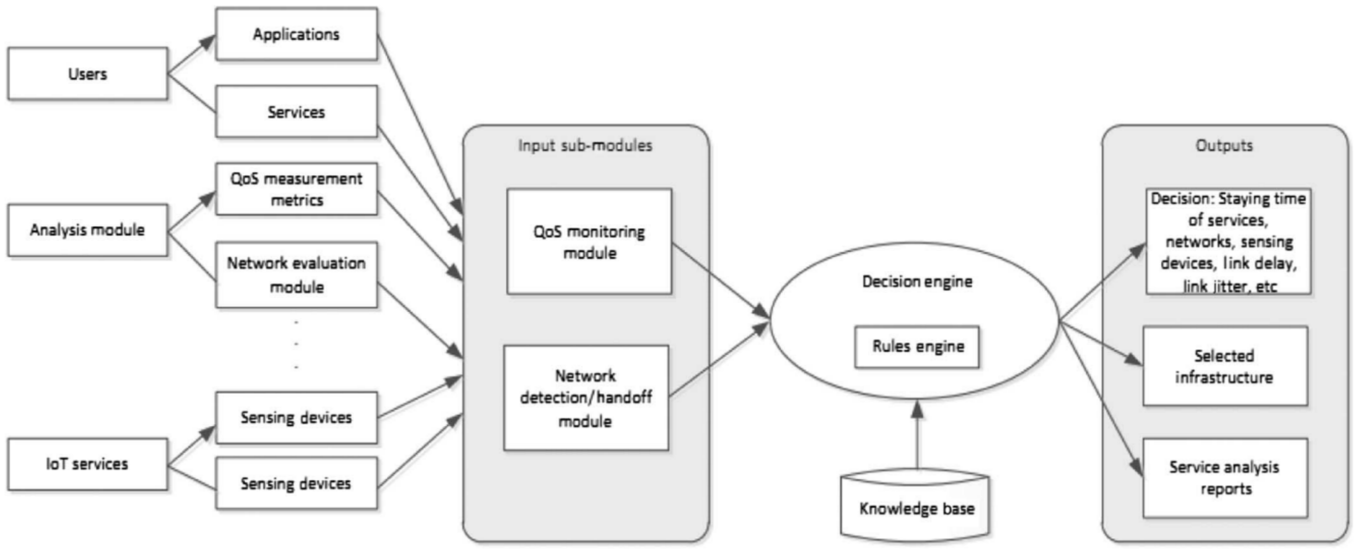


Figure 10. *Decision-making architecture for QoS in IoT*

The automatic monitoring of QoS metrics is a necessity and the QoS monitoring module (Figure 10) ensures that aspect. The QoS monitoring module is tasked to perform the following tasks-segregating the QoS requirements from the incoming applications into specific internal services of a macro service, generating the QoS attribute descriptions in all the three layers, and compiling the descriptions into subtasks that are to be executed.

The other QoS component is the decision-making engine. The decision-making in the application layer includes proper QoS scheduling of resources to incoming services. In the network layer, the decision-making includes non-technical metrics such as users, delay and the technical metrics which includes reliability and scalability. At the sensing or edge layer the decision-making involves accuracy of data, energy, and resource consumption of networks, it also includes the number of users and area coverage as QoS metrics in this layer.

Discussion

The scheduling approach is very well designed and implemented. It has generalized concepts which can be used at all the IoT layers with different parameters. There is an implementation of the knowledge base which has all the specific requirements and rules to help make proper decisions dynamically to produce the suitable outcomes. The knowledge base can be updated dynamically with newer deployments, so it is not restricted to a particular project implementation. The drawback of this method is the fixed domain, the QoS metrices targeted in this method is the resource utilization by the running processes and the QoS monitoring of network attributes. Also, the possibility of adding a self-learning component as a module can help further optimize the decision-making for the scalability of the system.

1.3.5 QoS based task distribution in edge computing networks

In this approach the researchers have tackled the issue of resource constraints in the edge layer along with the limitations of network bandwidth. The idea is to develop an algorithm which schedules and creates a task generation module to distribute the tasks to the appropriate edge devices [40].

The first step of the process is to get the tasks from varied IoT devices as inputs and the first module (problem formulation) consists of the available resources which acts as constraints in the task distribution step. The constraints that this model took into consideration are – each task can be assigned to at most one edge nodes and it should be successfully executed in them, the execution of the tasks would require storage in the designated node, so the storage capacities are checked with respect to the task storage requirements before they are being assigned to the following nodes, the virtual machines on each node should be sufficiently secured to be able to receive tasks for execution, each edge node has a defined VM which caps the number of tasks it can be assigned to, each task is accompanied with a time duration in which the tasks must complete its execution, and the last constraint is the bandwidth is shared by multiple edge nodes, this particular constraint is further divided into two parts. First sub constraint is the difference in incoming and outgoing link bandwidth and secondly is the bandwidth capacity of all the links in the edge network.

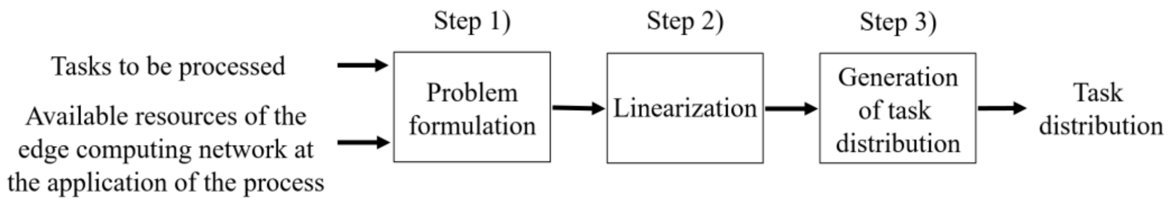

Step 1)
Step 2)
Step 3)
Tasks to be processed
Available resources of the edge computing network at the application of the process
Problem formulation
Linearization
Generation of task distribution
Task distribution

Figure 11 The task distribution process

Based on these constraint parameters the linearization module is developed as it was noticed the last two constraints is nonlinear in nature. Here, there is an assumption that all tasks are to be successfully executed exactly at their targeted destinations. This module is used to produce a mixed-integer linear problem which is then resolved by the third section, the generation of the task distribution module, to have a final output as the tasks to be distributed.

Discussion

This approach helps to increase the number of tasks that the edge computations can handle given a fixed number of resources. The key advantage of this approach was reflected in its performance compared to local execution of tasks or random distribution of tasks. The drawback, on the other hand, is due to the assumption that no task has priorities over other tasks. In real-time IoT systems, there are situations which requires certain tasks to be carried out before any other tasks. Also, the assumption regarding the VMs with one particular operating system downgrades the scalability factor of this approach.

1.3.6 QoS management for distributed IoT systems

This paper proposes a distributed QoS management framework for IoT that operates under resource constraints [28]. The authors argue that existing centralized QoS management solutions are not well suited for IoT environments due to the large number of devices and the limited resources available.

To optimize resource usage, the authors introduce a new algorithm based on dynamic programming approach. This algorithm is a decision-theoretic model that determines the optimal allocation of resources to meet QoS requirements while minimizing resource consumption. The algorithm works as follows: The QoS manager receives a QoS request from an IoT device, which includes the QoS requirements and the available resources. The QoS manager analyzes the QoS request and determines the optimal resource allocation based on the current state of the system, including the available resources and the performance of other IoT devices. The QoS manager sends the resource allocation plan to the QoS broker, which communicates the plan to the IoT device. The IoT device performs the requested operation using the allocated resources. The QoS monitor continuously measures the performance of the IoT devices and updates the QoS manager with the current performance metrics. The QoS manager uses the updated performance metrics to adjust the resource allocation plan and optimize resource usage based on the current state of the system.

This approach is based on a decision-theoretic model that considers the QoS requirements, the available resources, and the performance of other IoT devices. The model uses a utility function to measure the satisfaction of QoS requirements and a cost function to measure the resource consumption. The dynamic approach optimizes the resource allocation by finding the trade-off between utility and cost. The authors evaluate the algorithm using simulation and show that the proposed algorithm can effectively manage QoS in a distributed manner while effectively utilizing available resources.

Discussion

In this approach we see a promising solution to QoS management in IoT environments, which can help address the challenges posed by resource constrained IoT devices. The proposed framework and algorithm can provide a more efficient and scalable solution to QoS management in IoT environments compared to traditional centralized solutions. The limitation of the proposed framework is the overhead of the QoS monitor and the need for a reliable and efficient communication channel between the broker and IoT devices. The future research directions could be integrating machine learning techniques to improve the decision-making process of the dynamic problem architecture.

1.4 Conclusion

We discussed in this section of literature review on QoS metrices and the methods of enforcement of the QoS defined metrices. A special attention was given to QoS metrices that need to be satisfied in the fog layer of an IoT platform. We also investigated the current implementations and architectures that try to achieve an ideal QoS established IoT system. Following are the key drawbacks that we come across in the review:

a) LsM: Limited set of QoS Metrices

The architectures discussed in our literature review is found to be targeting a specific set of QoS metrices. As the concept of IoT is still comparatively new in the today's scenario, it is to be noted that most of the QoS related research has been done on the cloud and network layers. However, in order to build a stable IoT system it is no longer the only requirement, the QoS metrices must be satisfied in all layers starting especially from the edge layer and fog layers.

b) ASL: Absence of Self-Learning Ability

Most of the research lack the idea of introducing the self-learning component in their architectures. As we struggle to build a QoS established IoT system, we come across several real-time issues due to varied points of failure which requires a specific but redundant fix. So, having a self-learning model in place helps abstract the regular need of catering to similar types of issues or failures. This can reduce the user intervention at each point of failure significantly.

c) AFT: Absence of Fault-Tolerance Mechanisms

Another drawback that we get to see is the lack of system ability to cope with faults in real time. Any IoT system is prone to face unaccountable scenarios and in order for a reliable delivery of services the system should have default mechanisms to fall back to known stable versions irrespective of any unseen circumstances.

d) HCR: Higher Computation Requirements

Few of the QoS architectures have a very detailed and higher resource requirements for their implementation. The fog layer being short-handed in the resources section will have difficulties in accommodating any complex computations along with its defined services. We need to make sure the overhead for maintaining QoS in any IoT system should not be hindering the actual functionalities of the platform.

After exploring the literature of QoS in IoT, in the next chapter we present our designed approach.

Chapter 2

Design Approach for supporting QoS in Fog layer

The proposed solution we present in this chapter is an attempt to overcome the major drawbacks of addressing QoS in IoT platforms, which were faced during the real-life deployment of our AMI-Lab IoT platform and properly identified in the literature review. These drawbacks and the faced issues are the basic stone in the process of defining the requirements to provide a stable QoS implementation in IoT architecture. One of the key drawbacks is the limited focus on QoS for fog layer, with the major issues of self-learning ability (ASL) and resource constraints (HCR) in this layer. These identified issues in the review and confirmed through our real-life deployment, helped us define the vulnerable points, the list of requirements, and the associated QoS metrices for fog layer~~.~~

In our analysis of the literature review, we noticed that most of the existing QoS solutions are regarding network layer or cloud layer, with few records on fog layer. However, according to the challenges faced in our real-life deployment of AMI-Lab IoT platform, the fog layer is linked to a large number of QoS related issues. Therefore, we will present in this chapter our effort to stabilize the fog layer.

Following, we will quickly present the fog layer in order to highlight the internal components which are the source of the vulnerable issues in the fog system. After that we will discuss the lessons learned from our real-world deployments. Based on our analysis of the literature review and lessons learned we will depict the point of failures in fog layer. Finally, we will propose our approach in tackling the distinct point of failures.

2.1 Internal Components of Fog Layer

The fog layer is represented as the physical gateway that acts as a hub for multiple devices with different communication protocols. It is tasked with preprocessing the acquired data before sending it to the cloud layer via the network layer. The fog layer consists of components which are responsible for carrying out these functionalities. The major QoS metrics that has to be targeted on addressing through the fog layer components are: (i) **latency** in the fog network, (ii) **scalability** of the fog layer, (iii) **resource constraints** management in the fog layer, and (iv) the **offline capability** of the fog system (see section 1.2.2).

Following, we will discuss the internal components of a fog node and discuss the points of vulnerability in them.

A) Fog layer: Single Node

A fog node typically consists of a data acquisition model, a data pre-processing model, and a data communication model. The common areas of vulnerability in such systems are insufficient resources, reliability of the running services in the node, availability, security, offline capability of the node (Table 3, section 1.2.2, page 17). To monitor these points of failures we need to understand the exact internal components of the fog nodes. Following, these components are presented at an abstracted sub-layer.

B) Abstract sub-layers of Fog nodes

The internal functionalities of the fog nodes are divided into three abstract layers (Figure 12).

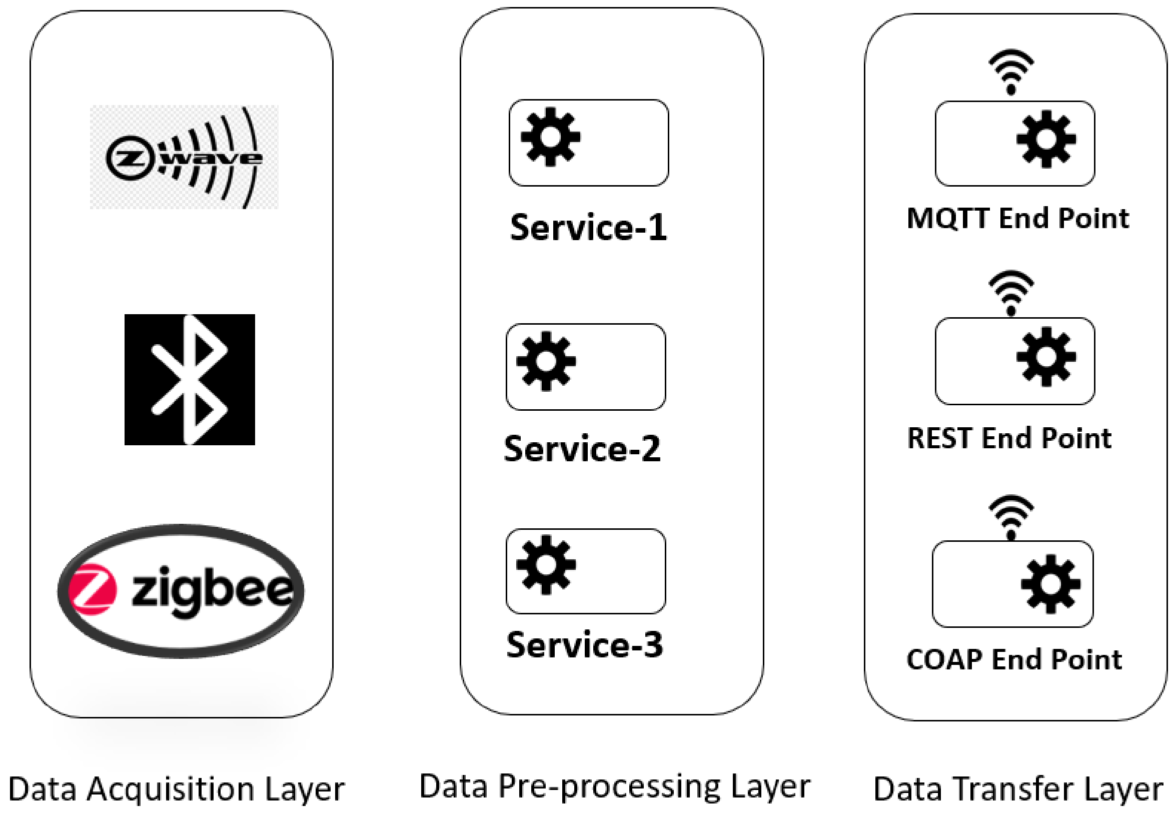


Figure 12. Abstract layers in Fog node

The first layer is the **data acquisition** layer which is responsible for collecting data from different edge devices and is capable of supporting more than one communication protocol. Next, comes the **data pre-processing layer**, the raw data obtained from heterogeneous sensors are pre-processed in this layer. The third layer is the **data transfer layer**, after the data has been preprocessed it is sent to the cloud layer via different protocols in the network layer. So, the data transfer layer provides the end point for these network layer communication protocols. Following, we have an overview of the vulnerabilities in the three internal components of the fog layer.

C) Vulnerable Points

The vulnerable points for the data acquisition layer are mostly related to the support mechanism for numerous communication protocols. The edge devices are a point of vulnerability in this sublayer as the acquiring of data from the edge devices is a key functionality and in scenarios where the edge devices malfunction the data acquisition layer is held responsible.

For the data pre-processing layer, the vulnerable points are the services running to pre-process the data. Any failures in the running services can cause the layer to malfunction and raise issues.

As for the data transfer layer, the communication protocols are again the major points of vulnerability. This layer is the key between the fog layer and the cloud layer and hence and issues caused in this layer will cause the data to stop flowing altogether. This layer also has a higher risk of security.

The vulnerable points were also discovered during the real-world deployment of our IoT platform. In the next section, we will discuss the lessons learned through this real-world deployment.

2.2 Lessons learned from the AMI-Lab deployment.

Our AMI-Lab has been involved in developing and deploying an IoT platform (AMI-Platform) to promote better delivery of services in smart cities. It targets helping older adults to lead an independent and purposeful life through ambient assistive technologies [13]. Our team came across several challenges along the deployment of this AMI-Platform in real setting, which in turn motivated us to come up with the approach of establishing a fog layer with QoS metrices. Following, we discuss the hurdles faced in the fog layer.

A) Missing Data Issue

In our deployment, we have encountered situations when the data was not observed in our dashboards and on further investigations it was found out that the vulnerable points were lack of resources in the fog nodes or the lack of stable Internet connectivity. Another point of failure for the missing data issue is the edge device failures. The batteries of the wireless edge devices have a limited time range of availability and thus causing the devices to stop sending data after a fixed time period.

B) Offline Capability Issue

The AMI platform did not support offline capabilities which is an essential QoS fog layer requirement in any IoT platform. In one scenario we discovered that there was no communication from the fog nodes due to the lack of Internet connection. This resulted in loss of data in that period of time as there were no mechanisms to handle the offline capability.

C) Data Interoperability Issue

The AMI platform supported numerous edge devices with heterogeneous mode of communication. The heterogeneous nature in turn led to extensive pre-processing in the data pre-processing layer consuming more resources which slowed down the delivery of services.

D) Resource Constraint Issues

The platform supported devices which have high flow of data frequency and due to the resource constraints in fog layer there was a significant amount of delay added to the data before transmission to the cloud layer. On one occasion due to memory leak in services, where a log statement was logging continuously and in a matter of less than 20 hours the physical memory of the fog node was overloaded causing the node to shut down all its services.

We have discussed in this section the issues faced in the fog layer and the vulnerable points in the fog system through our real-life deployment. In the next section, we will discuss the most common point of failures that are the possible reasons for the vulnerabilities in the fog layer.

2.3 Most common point of failures in fog layer

We detail in this subsection the most common issues and their point of failures, according to our lessons learned from our deployment:

A) Missing data issue

The most common issue in any fog node is the **missing data issue**, in which the cloud layer suddenly stops receiving data from the fog nodes. There can be several points of failures for this issue.

- In the data acquisition layer, the hardware connected to the fog node might malfunction because of which there are missing data [41]. The missing data issue can be caused by several service failures in the data pre-processing layer as well.
- Similarly, in the data transfer layer we have our network protocol implementations which can be erroneous leading to data even after being pre-processed not being able to reach the cloud layer. It could be due to erroneous configurations between fog layer and the cloud layer, mismatch of certificates, etc. [22].
- The other causes for this issue are external causes for example, loss of the Internet connection which leads to no data transfer to the cloud or power outage resulting in the complete shutdown of the node itself.

B) Data interoperability

The other type of failures that are encountered in the data pre-processing layer is the issue of **data interoperability.** The lack of data interoperability means that data can't be effectively exchanged across different edge devices and IoT systems. The main point of failure is in the data acquisition sub-layer.

- In the data acquisition layer, the raw data acquired from the different sensors which have different data formats and does not support interoperability with other data formats. The readability and the processing of such data become extensively difficult across the entire IoT platform and is not suitable for cross-platform usage [24].

C) Computation Issues due to Resource Constraints

Resource constraints are another set of point of failures that adds to major issues faced in this layer. The computation issues are mainly found in the data-preprocessing layer.

- In the data-preprocessing layer the services implemented for pre-processing of data sometimes require a lot of CPU and memory resources which might not be available in the fog nodes. This issue causes the node to hang up or stop working in cases of resource overloading which in turn blocks other services [42].

D) Offline Capability Issue

The offline capability of a fog node is to provide knowledge of the system's uptime and downtime along with fallback mechanisms to recover loss of data. The point of failure for this issue is mainly external –

- Power outages causing complete device shutdown.
- Internet failure that disrupts the communication between the fog layer and the cloud layer.
- Unavailability of the server in the cloud layer.

The next section comprises our proposed approach to the presented points of failures and issues discovered during our literature review and real-world deployment.

2.4 Proposed Approach

The section starts with the overview of our approach to counter the issues of fog layer and then we discuss the implementation ideas with real-time examples.

2.4.1 Overview of the Approach

The fog layer can consist of a single node acting as the physical gateway to the cloud layer or it can have multiple nodes in a distributed manner. Therefore, our proposed approach is a two-step solution. The <u>first step</u> targets issues for a IoT fog layer with a single physical node where we introduce agents in each sub-layer of the fog to manage all points of failure. The <u>second step</u> is to extend the idea to a distributed environment in the fog layer.

Another major part of our solution is the agents / small components and tools implementation approach that are not built for any specific QoS metric but as a method to enforce one or a number of QoS metrics in any platform~~.~~ Our approach thus provides the properties of scalability and maintainability to the fog layer. For the implementation of smaller agents and components, we refer to the concept of autonomic computing which promotes the agent-based approach in our fog layer. Then we will present the roles and functionalities of a typical agent for QoS. Continuing, we will provide examples of the QoS agents built into the real-world deployment of AMI platform.

2.4.2 Autonomic Computing and Agent-Based for QoS

The agent-based approach is derived from the concept of autonomic computing. The concept of autonomic computing was first defined by IBM. The term "autonomic" comes from biology like the autonomic nervous system, which is capable of handling most of the involuntary body movements and to maintain the coordination among them. The ideal goal of systems is to have computing environments that evolves without the need for human intervention (addressing the issues of ASL and AFT) [43]. Through autonomic computing the system is able to realize four major types of self-management attributes- self-configuration, self-optimization, self-healing, and self-protection. The table below shows how the four self-management systems help achieve automaticity [33].

These properties of the autonomic computing are the key features that were used in our approach to counter the major drawbacks encountered in our real-life deployment. The property of self-configuration helps us to deploy multiple fog nodes at the same time, the property of self-healing helps us to monitor and diagnose services running in the fog nodes with very less user interaction. Security threats are dealt by the self-protection property of the autonomic computing.

Table 8. Four aspects of self-management systems.

Property	**System Definition**
Self-configuration	The systems, capable of handling self-install and can set to default configurations to fulfill the required needs, are said to have the self-configuration property.
Self-optimization	The systems that can make proactive changes resulting in increasing performance of the system are said to have the self-optimization property.
Self-healing	The systems having this property can detect and diagnose problems with a certain level of fault tolerance.
Self-protection	The systems that can protect itself from malicious attacks and offer security, privacy and data protection is said to have the self-protecting property.

Next, we will depict how we can integrate this concept of autonomic computing to build autonomous agents to help implement our solution for QoS.

A) Agent Approach

The concept of autonomic computing promotes the idea of reducing user interventions in an anomaly situation by developing small agents to perform the tasks usually carried by the user. The usual steps that are carried by the agents are inspection of the issue, notification process and resolving process of the issue if known.

The two main types of agents used to create several autonomic systems are Reactive agent and Deliberative agent. When the system constantly senses the environment and takes reaction, we use the reactive agent in such cases while, on the other hand, to create an autonomic system with predefined goals we use the deliberative agent [33].

In our approach we use a hybrid agent model where agents are reactive to account for any real time failures and also has a predefined set of rules to maintain the stability and reliability of the system. We discuss about our hybrid agent-based approach to stabilize the fog layer with QoS in the next section.

B) Agent for QoS

We propose a model following the parasitic approach as a solution to overcome the failures discussed in the section 2.2 (failures pertaining to resource constraints, network failures, etc.). The parasitic approach is based on the analogy of a parasite, an organism living on another organism and deriving nutrients from it. Similarly in our approach we attach a QoS agent to each of the services in all the abstract layers of the fog nodes (Figure 12).

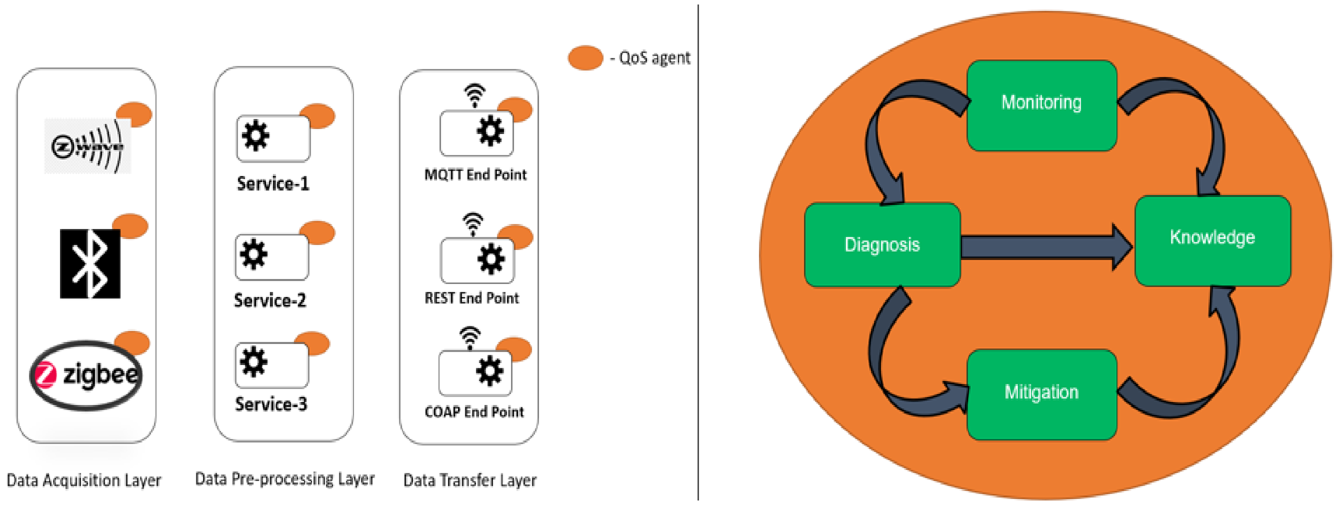


Figure 13. Abstract layers in Fog nodes with QoS agents

Each QoS agent consists of four main components (and their associated roles):

1) monitoring components, in charge of monitoring all points of failures in the system,
2) diagnostic component, runs an analysis on the failure caused and checks for possible solutions,
3) mitigation or reacting component, which is used for taking action to overcome the failure autonomously and
4) the knowledge base, this component is used for keeping records of all failures and solutions that were taken by the system to help build a better diagnostic module in the long run.

The internal structure of the QoS agent (Figure 13) is based on a self-healing model as described in our paper [13]. These QoS agents are coupled with each of the services present in the abstract layers of the fog system. This approach leads us to pinpoint our point of failure whenever the IoT platform encounters an anomaly. The QoS agent corresponding to the failed service raises an alarm and performs the necessary steps to overcome the issue. From the user point of view, we can navigate to the source directly instead of spending days trying to detect the exact point of failure (ASL, AFT).

Following we present the example of missing data QoS agents in order to illustrate the above presented roles.

C) Example of QoS Agent roles for the missing data issue

In the context of the missing data issue, the QoS agents in the data acquisition layer monitors (monitoring role) the acquisition of the raw data from the heterogeneous communication adapters. With the knowledge of the average amount of data that is supposed to be received for a particular period of time from a specific adapter, the QoS agents notify (mitigation role) the admin in case of abrupt changes in the amount of data acquired. For example, a sensor is supposed to be sending the temperature and humidity of a room at an interval of twenty minutes. Now, it was observed that the sensor is not sending any data for a period of more than an hour which is definitely an anomaly and thus needs to be investigated. There are some involuntary actions (AFT) that the QoS agents can take in such scenarios, such as restarting (mitigation role) the data acquisition service or the service operating the device adapter which is supposed to be receiving the data from the sensor. After taking these pre-emptive measures, the QoS agent continues to monitor the acquisition and if the issue still persists, the logical conclusion (ASL) is that there must be a hardware failure either on the sensor side or the device adapter is failing to operate correctly.

To address the point of failure in the data pre-processing layer for the missing data issue, we can have QoS agents monitoring each service running in this layer. The failure of these services might cause the data to be missing in the overall data flow. Thus, the agents are tasked to monitor the status of these services on failure they take the pre-emptive measure of restarting the services and then informing the admin for possible anomalies that can hamper the overall QoS of the system.

In the data transfer layer, IoT platforms incorporate several communication protocols for example Message Queue Telemetry Transport (MQTT) or the hypertext transfer protocol (HTTP) protocol. The endpoint of these protocols is managed by services in the data transfer layer. They are tasked with creating the necessary socket connections with proper security measures and our QoS agents are monitoring these services if they have stopped due to some bugs or errors which would result in the complete stoppage of data flow in the IoT platform.

Following we discuss the QoS agents that we have developed in our fog nodes, and we discuss how they can be used across all different IoT platforms.

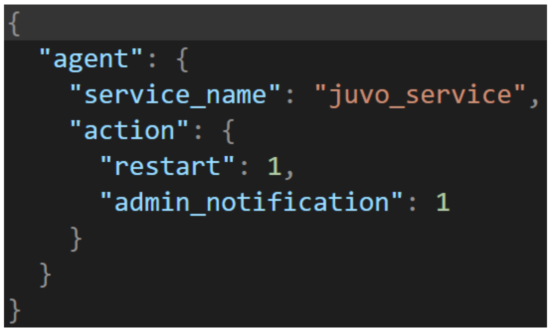
2.4.3 Agents to support QoS in single nodes

We describe in this subsection the QoS agent which is in charge of catering to the point of failures related to fog layer (defined in section 2.3). These agents are dynamic in nature as they are not tied to a particular resource, the idea is to have generic agents with configurations to manage any specific resource (running services, Internet connectivity, memory resources, etc.).

A) QoS agent for service monitoring

The QoS agent for service monitoring is used for proper monitoring of services running in the fog layer with pre-emptive measures and notification system to inform the admin (AFT). We create a generic agent which takes in as parameters the name of the services and which actions to be taken in cases of failures. We use a JSON config file as an input for these agents to keep it independent from any specific platform (ASL)(Figure *14*). We can set the action parameters to 1 or 0 denoting whether these actions are to be taken by the agent or not.

For example, in the case of missing data issue the service monitoring agent can help pinpoint the source of the issue without delay.



```json
{
  "agent": {
    "service_name": "juvo_service",
    "action": {
      "restart": 1,
      "admin_notification": 1
    }
  }
}
```

Figure 14. *QoS agent input configuration for service monitoring*

B) QoS agent for the Internet connection

In the case of the Internet connection issue, the data is being collected from the sensors and being pre-processed in the fog node but is unable to get transferred to the cloud layer, now there are two steps in addressing this issue: The first step is to try and restart the Wi-Fi or Ethernet adapter in the node and to check if the connection is up or not (AFT). We create a QoS agent to monitor the Internet connection and take the necessary action in case of failure (ASL) [45].

C) QoS agent for data recovery

The QoS agent for internet connection does not guarantee the reliability of the platform in case of the Internet failure as the internet provider fails to provide the internet connectivity. So, as a remedy to such cases, we built a data recovery agent. This agent has multiple functionalities, it starts from storing data locally in the fog node to syncing data to the cloud nodes on a periodic basis and to use minimal fog node resources in doing so (AFT). This agent has a sublayer in each of the services in the data pre-processing layer, it has a local database setup and a local storage service which is mainly used to manage the database and perform the syncing of data with the cloud database (Figure 15). The data recovery micro-agent is a plugin to each of the pre-processing services and the main functionality of these agents is to put the pre-processed data in the local database. After the data is put in the database, the local storage service is tasked to sync with the cloud database at a periodic interval.

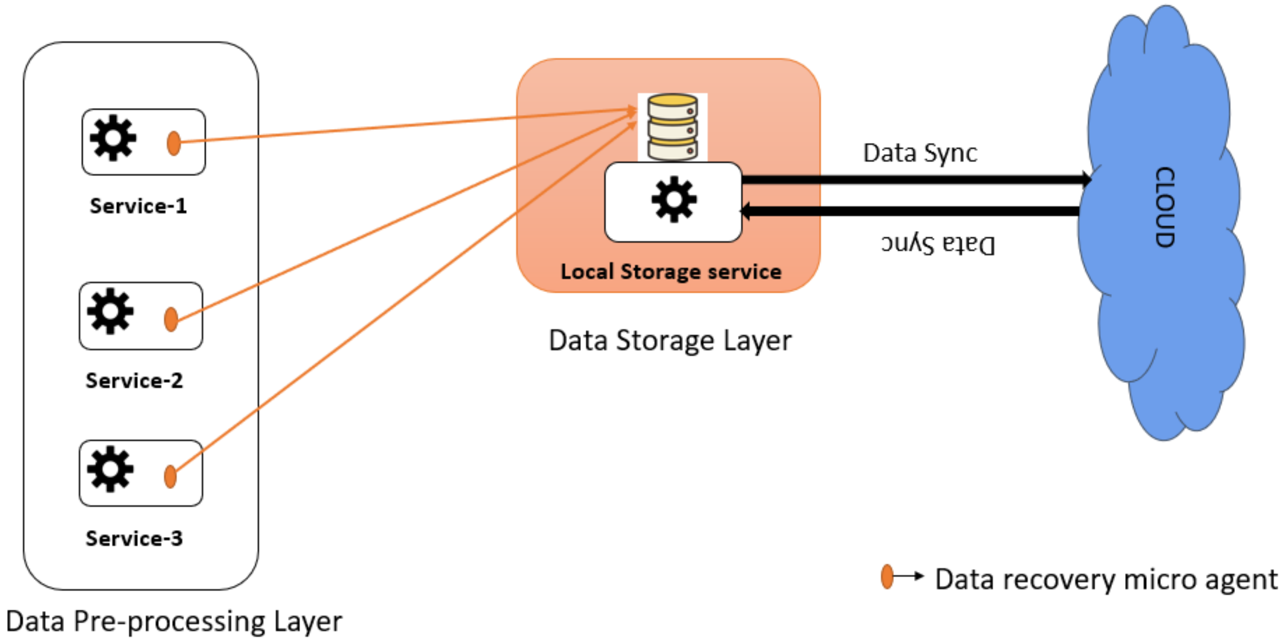


Figure 15. Data Recovery QoS Agent

D) QoS agent for resource constraint

The fog nodes have limited resources for computation and storage purposes (HCR). In order to maintain the reliability of the platform, we have to manage the resource utilization of the nodes by all the services running in it such that a particular service or a group of service does not cause over saturation of resource usage, thus causing a breakdown of the node. According to our analysis, the main resources that need to be targeted are CPU, disk space and physical memory usages. Therefore, we built QoS agents to monitor and maintain them.

a) QoS agent for CPU

A QoS agent running as a separate service inside the fog node monitoring the CPU usage at a definite interval, a similar model was proposed in "**Self-healing Approach for IoT Architecture: AMI Platform**" [45]. This agent takes in as an input the threshold CPU usage for a particular node and an interval to perform this check. At a particular interval if it was noticed that the current CPU usage is more than the threshold value the agent takes action to counter overloading of the node. It goes into a deeper level of searches where it checks the CPU usage of all the running services in the node and selects the heaviest process running in the system. It restricts the usage of that particular service to half of the CPU resource. The service also notifies the admin about such scenarios to have a proper optimization of the service as soon as possible.

b) QoS agent for disk usage

Similar to the agent for monitoring the CPU usage we have an agent for monitoring the disk space usage. This is the secondary memory space, or the external memory attached to the fog node. The agent takes in the threshold value for the disk usage and the interval at which the check needs to be performed. Now, it was observed that this space is overloaded mainly by logs and different secondary files that the running services might produce at run time. So, the pre-emptive action that the agent takes in case of overloading is that it truncates all the log files and stores the last 100 lines of each service logs in a separate file. The admin on notification can reach out to the node while having the latest logs and the reliability of the node is also not compromised.

c) QoS agent for physical memory usage

The input format for the QoS physical memory agent is the same as the prior resource constrain agents, it requires the threshold value and the interval value. There are not many pre-emptive actions that can be taken in case of an overloading of the RAM or the physical memory. The best solution for such cases is to perform a reboot of the fog node. However, unsupervised reboots of the nodes are not advised as it might enter an infinite loop of reboots due to some unknown reason. Hence, the action taken by this agent is to provide the admin with a high alert notification to solve the problem on an immediate basis.

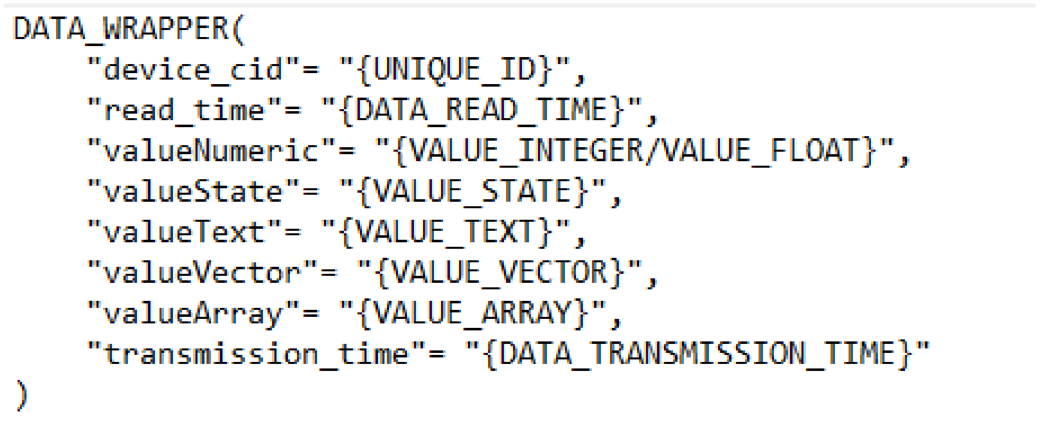
E) QoS agent for heartbeat

This agent is built with the idea to always monitor the status of the fog node. This agent although runs independently in the fog node and can be used across different platforms, it requires a twin agent in the cloud. An "alive" message is sent by the agent in the fog node to the agent in the cloud node which on receiving the message sends back an "acknowledgement" message. Both of these agents in the fog and the cloud nodes have a timer called "alive" timer and "acknowledgement" timer respectively. The timers work on the same philosophy, so if the timer runs out of the specified time period it raises an alarm. The admin is notified with a high alert message denoting the fog node or the cloud node being down and needs to be powered back in or restarted to re-establish a connection (AFT). This agent is mainly used as a tool for the offline capability of a fog node.

F) QoS data model agent for data interoperability

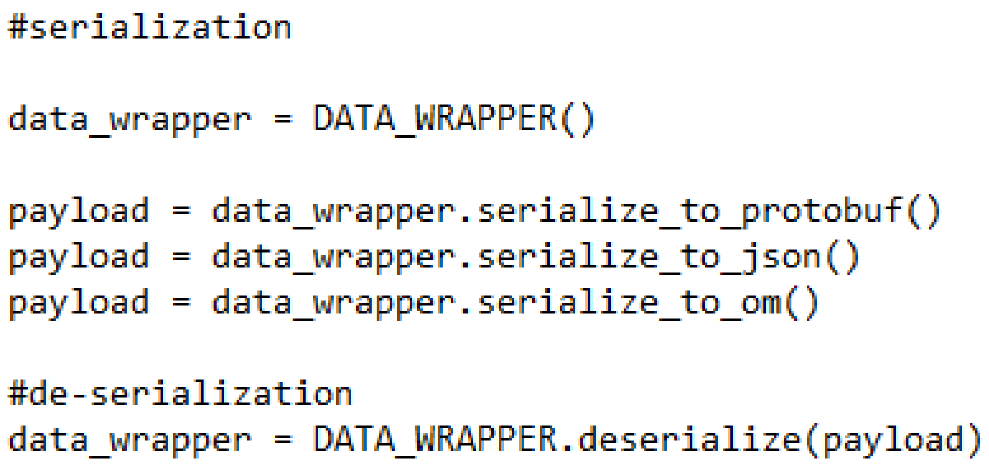
The raw data acquired by the data acquisition layer has different data formats due to the heterogeneous nature of the different edge devices and sensors. To be able to use these data throughout the pipeline of the IoT platform and even for cross-platform usage we need to have a specific data format else it becomes exceedingly problematic in handling the data interoperations. So, in the data pre-processing layer, we have a QoS agent which wraps all these different data formats into a specified data model that is easily readable and can be interoperated on various platforms. The data model used is a generic and dynamic model which supports all known data types (known till date). The QoS agent comes with its own data encryption methods along with conversion methods between different encryption, which is a very essential requirement for network communications in regard to security. This QoS agent can be used across the entire IoT platform to maintain the same data structure throughout the IoT pipeline.

This agent is mainly focused on converting all different data formats to a specific data model which resolves the data **interoperability** issue and also helps maintain a singular data format throughout the IoT platform. We depict the structure used in AMI platform to wrap the acquired data from the edge layer (Figure 16) and some method conversions provided by this agent (Figure *17*).



```
DATA_WRAPPER(
    "device_cid"= "{UNIQUE_ID}",
    "read_time"= "{DATA_READ_TIME}",
    "valueNumeric"= "{VALUE_INTEGER/VALUE_FLOAT}",
    "valueState"= "{VALUE_STATE}",
    "valueText"= "{VALUE_TEXT}",
    "valueVector"= "{VALUE_VECTOR}",
    "valueArray"= "{VALUE_ARRAY}",
    "transmission_time"= "{DATA_TRANSMISSION_TIME}"
)
```

Figure 16 Data model semantics



```
#serialization

data_wrapper = DATA_WRAPPER()

payload = data_wrapper.serialize_to_protobuf()
payload = data_wrapper.serialize_to_json()
payload = data_wrapper.serialize_to_om()

#de-serialization
data_wrapper = DATA_WRAPPER.deserialize(payload)
```

Figure 17. *Encryption provided by the QoS Agent*

Discussion

We have discussed in this section numerous issues related to a singular node fog layer. We discussed the approach to address these issues caused by different points of failures. While this approach is good to address the issues faced, it is not scalable for larger domain of services. With a singular fog node, the major setback that remains is the lack of extensive resources (HCR). QoS management comes with an overhead of resources in the fog layer. The monitoring aspect of the QoS agents can be addressed to using a singular fog node, however, the agent being able to resolve the issue might not be possible as it would require more memory or processing units to be able to perform the necessary tasks. For example, we handled the offline capabilities by adding an agent to manage a local data storage component but the limit for the data storage unit is very less and is not sustainable for a moderately longer time period. A solution to this drawback situation is a distributed system which consists of multiple fog nodes providing more resources to work with in a distributed fog layer. Following we discuss distributed fog layer in support to IoT QoS.

2.4.4 Support for QoS in Distributed Fog Layer

To highlight the importance of the need for distributed fog system to ensure QoS, we can present the following example of Bluetooth from our real-life deployment in regard to scalability and reliability. AMI platform supports Bluetooth devices, and a single fog node is able to scan for any nearby device while acquiring data from the already registered devices. But it had a limit to a number of devices it can scan and acquire data at the same time. The drawback in this implementation was that the Bluetooth adapter was only able to support one to two Bluetooth device at a time and crashed during higher rates of data transfer. As a solution we developed a QoS agent, which monitored the service and the pre-emptive measure for failure scenarios was to restart the service. Although the pre-emptive measure resolved the issue at hand, it had an impact on the number of uptime and downtime of the service resulting in loss of data.

Another example from our deployment is the time taken to deploy our IoT system components in the fog nodes. On an average it took about thirty minutes to install the system components in a single fog node option, and to deploy multiple nodes it became very difficult for a user to take care of it.

With these examples in mind, we move forward to the next section which includes the QoS implementations in a distributed environment with multiple fog nodes locally connected to one another in the same network.

Overview of QoS in Distributed Fog layer

The idea behind our distributed fog layer is to overcome the challenges faced by a singular node in the fog layer. The approach for the distributed Fog layer is not to build anymore small QoS agents but to have one generic distribution model which can be used to not only perform distribution functionalities in between the fog nodes but also to enforce QoS metrices in more than one node at a time (Figure 18).

We have enlisted several QoS metrices in our literature review, which any IoT platform should take into consideration to have a better overall system performance. In order to enforce these QoS metrices, we propose a dynamic distribution model along with an engine supporting the model which provides the implementation of the above-mentioned QoS metrices.

In the next section, we discuss our distributed approach detailing the distribution model and its internal components.

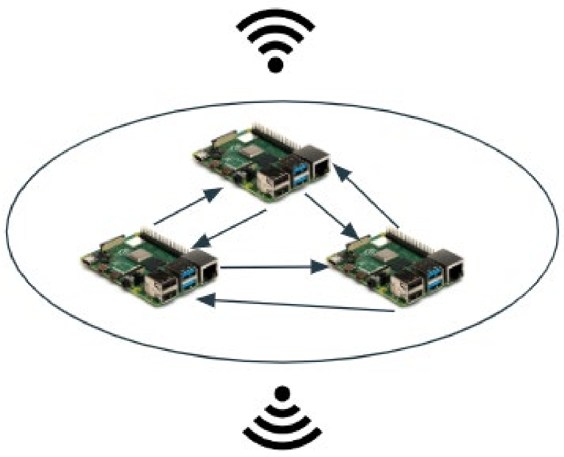


Figure 18. Gateway nodes in a distributed fog system

Distributed Model

The main goal of the model is not to have a one-time distributed system which is able to handle any fog layered complexities, but to provide a base on which we can develop and scale our distributed model with the capability to adjust to any unforeseen circumstances. Our approach is to provide the tools required by QoS agents to make the desired distributed system with QoS enablement.

The major aspects that will be targeted in our approach is the distribution of resources and the ability for a proper management and monitoring system across all the nodes. The distribution model follows a demand-action architecture (Figure 19). The demand from the external environment (in the form of an input model) gets converted into tasks by our distribution engine which is then carried out by our system to meet the desired results. The model also provides an acknowledgement after carrying out the action and this is provided through an output model. Along with the input and output model, the distribution engine (Figure 19) consists of three main internal components –

- Communication engine: The engine that is tasked with all communications across the distributed system. It also manages the security aspect to block access to the distributed system from outside fog nodes.
- Distributed Library engine: The main task for this engine is to convert the incoming communications to the respected tasks. This library engine is also used to handle erroneous or abnormal communications.
- Remote Hosts Engine: This engine has the specific task of keeping the fog node updated with the distributed environment. Also has a security layer to prevent from outside attacks.

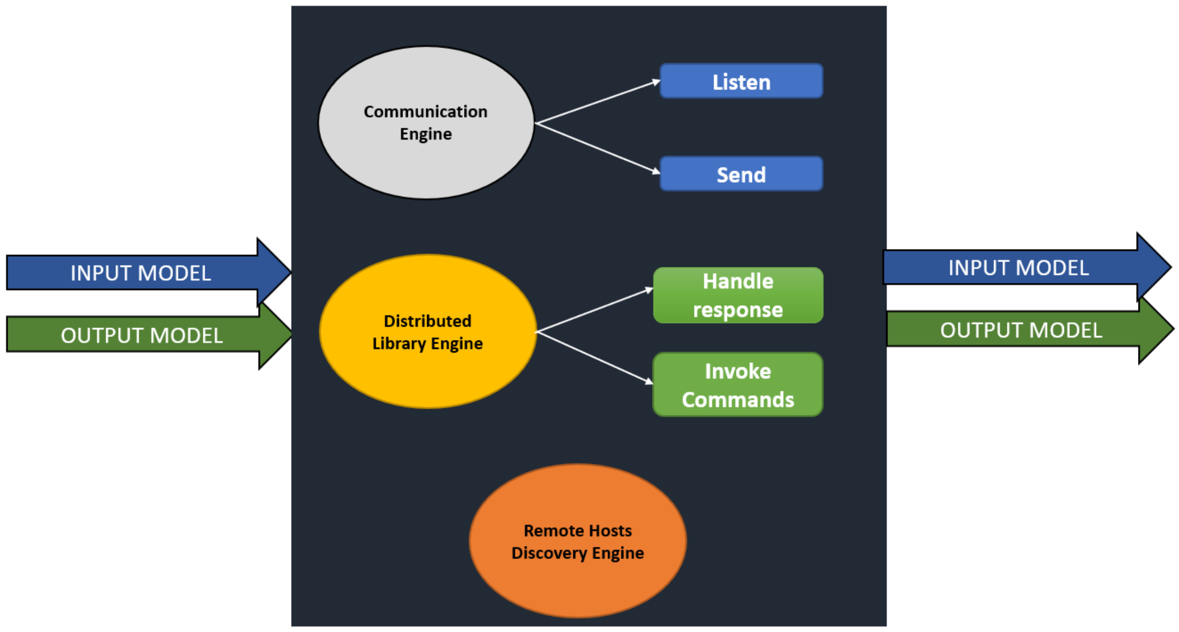
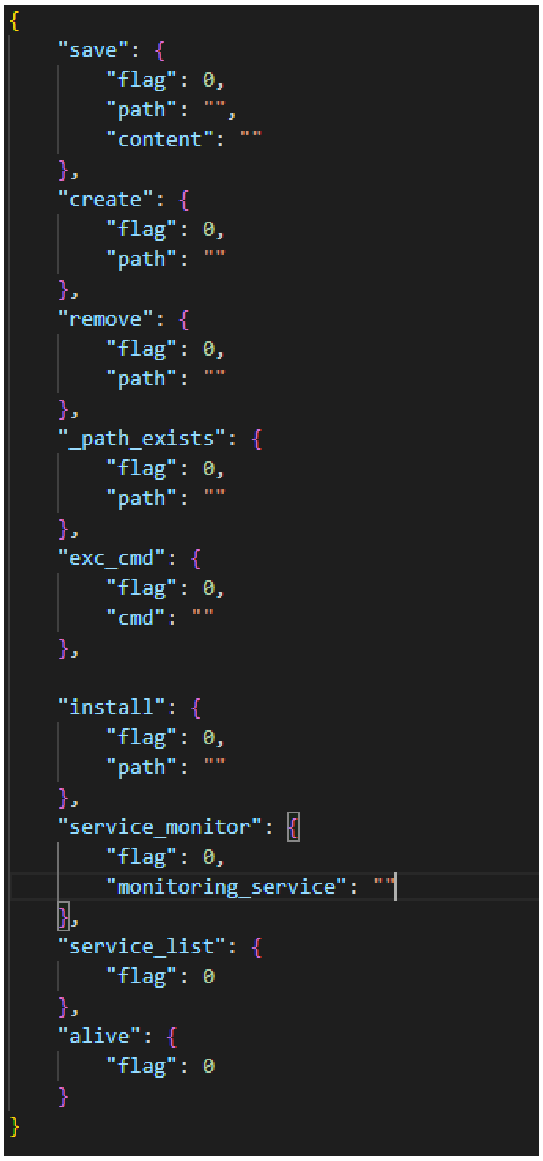


Figure 19. *Distribution Engine: Internal Structure*

Following, we discuss in detail the internal components of the distributed model and then we will discuss how these components help provide a stable model for QoS implementations in the fog layer.

A) Input Model

The input model is a measure of communication in the distributed environment. For the interoperability of multiple nodes in the distributed system, we designed a generic and dynamic model of communication. It is built in a JSON format, with the key value architecture. The key denotes the name of the action or task that is intended to be performed at any other node present in the distribution environment. From our real-life deployment, we observed that most of the issues due to any point of failure required the same actions as part of their solutions. Now, to automatize this step, we have this model equipped with the most commonly used actions by any user to fix issues Figure 20. The model itself is not static in nature and hence addition or removal of tasks is very easily achievable due to the simplicity of the structure of our model (based on JSON format), no complex data format is required to update the model.



```
{
    "save": {
        "flag": 0,
        "path": "",
        "content": ""
    },
    "create": {
        "flag": 0,
        "path": ""
    },
    "remove": {
        "flag": 0,
        "path": ""
    },
    "_path_exists": {
        "flag": 0,
        "path": ""
    },
    "exc_cmd": {
        "flag": 0,
        "cmd": ""
    },

    "install": {
        "flag": 0,
        "path": ""
    },
    "service_monitor": {
        "flag": 0,
        "monitoring_service": ""
    },
    "service_list": {
        "flag": 0
    },
    "alive": {
        "flag": 0
    }
}
```

Figure 20 *Input Model*

We integrated a set of features in the input model (detailed in the table below with the required parameters for each feature and the achievable outcomes (Table 9) which are used for QoS applications. These features are used by QoS agents to carry out their roles in the fog nodes.

Table 9. Input Model Features

Feature Name	**Parameters**	**Function**	**QoS application**
Save	**flag**: indicates the status, whether this feature is used or not. **path**: The absolute path for the content to be saved. **content**: The content to save. (Json, string, byte strings)	**Saving** any content to a specific path in any connected local or remote nodes.	QoS agents can use this feature to share configuration and status of each node to one another providing a better QoS monitoring system.
Create	**flag**: indicates the status whether this feature is used or not. **path**: The absolute path of the file.	**Creating** a file in a remote node at a specified position.	QoS agent in one node with proper knowledge of an issue can help another QoS agent in other nodes by creating files and sharing information.
Remove	**flag**: indicates the status whether this feature is used or not. **path**: The absolute path of the file.	**Deleting or removing** a file in a remote node from a specified position.	QoS agents can remove unwanted files causing bugs and issues in the system, for example unwanted log files overrunning the memory.
Path_exists	**flag**: indicates the status whether this feature is used or not. **path**: The absolute path of the file.	**Checking** if the specified path is existing in the remote node.	QoS agents can use this feature to verify the existence of files and folders which can be used as a security measure to check for vulnerabilities in the system.
Exc_cmd	**flag**: indicates the status whether this feature is used or not. **cmd:** the command to be executed.	**Execute** a specific command in the remote node.	QoS agents can execute any command in any node for any tasks. For example, one node can execute a command to restart a particular service in another node if need be.
Install	**flag**: indicates the status whether this feature is used or not. **path:** Path for the script to be installed.	**Install** the specified scripts.	This feature is specific to QoS agents handling the challenge of fast deployments. QoS agents can install all the required services from an IoT platform by distributing the tasks in the different nodes and running

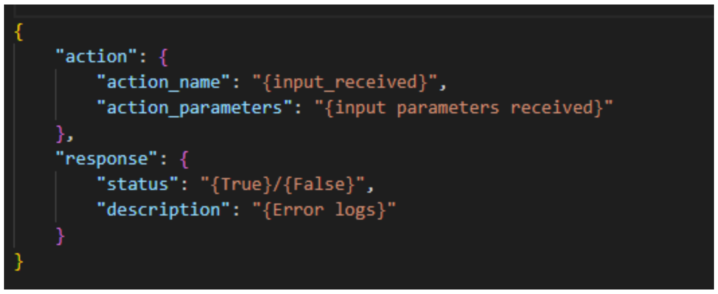
Table 9. Input Model Features

			installations on them. This limits user intervention to install each service on the designated nodes.
Service_monitor	**flag**: indicates the status whether this feature is used or not. **monitoring_service**: name of the service to be monitored.	**Gets** the status of the service running in the remote or local node.	QoS agents use this feature to monitor the status of the services running in other nodes.
Service_list	**flag:** indicates the status whether this feature is used or not.	**Gets** the list of running services from the remote nodes.	QoS agents are able to communicate with each other by providing a list of running services in their respective nodes.
Alive	**flag:** indicates the status whether this feature is used or not.	**heartbeat** features to check if a node is alive or not.	It's a ping property of all nodes used by QoS agents to monitor the heartbeat of all nodes.

Next, we discuss the response data model corresponding to this input model. When any action is being carried out through the input model in one of the nodes, it sends back the response in the specific data format.

B) Output Model

The output model is built with the same key value architecture in the JSON format. This model, on the other hand, is static as it is used as a reply mechanism to the tasks executed corresponding to some inputs. The output model has two features denoting the action which was executed and the response of the system to that action (Figure *21*).



```
{
    "action": {
        "action_name": "{input_received}",
        "action_parameters": "{input parameters received}"
    },
    "response": {
        "status": "{True}/{False}",
        "description": "{Error logs}"
    }
}
```

Figure 21. Output Model

Table 10. Output model features

Feature Name	Parameters	Function
Action	**action_name**: specifies which input feature was executed. **action_parameters**: the parameters used for the execution.	To enable the engine on the other end to understand the context of the response
Response	**status**: True or False depending on the success of execution. **description**: Error messages if any.	To notify the result after the execution of the input model.

The table above (Table 10) describes the attributes of the output model of what each parameter is defined as and their functionalities. It provides in detail how the features can be used by the distributed model as an acknowledgement mechanism in correspondence to the input model.

The output model is used as a means by the distributed model to verify the tasks carried out through the input model were completed successfully or not. Based on the response QoS agents can take necessary actions.

In the next part we are going to depict how these input and output models are being used by the distribution model, along with the fault tolerance mechanisms and tools to build new QoS agents on top of this model.

C) Distribution Library

The distribution library is used as a mapper between the input features described in the input model to the tasks that the features demand to be carried out in the operating system. As developers we come across a number of used daily tasks and commands that are performed in order to maintain or monitor a system. The idea is to provide the platform with the knowledge base to carry out required functionalities without a user intervention (fault tolerance abilities). The task list defined in our input model has methods corresponding to them in this distribution library. The methods are executed with a layer of exception handling. For instance, the input model has the “save” feature to be executed, so we are trying to save some content to a remote node at specific path, but the path seems to be not present at that location, in such cases the exceptions are handled, and the path is first created and then the content is saved. It again becomes a little tricky when we need to append some content to an already existing file, we need to make sure the format of the file is not broken. For example, we are trying to append a new content to an already existing file with other values, now if we simply add the content to that file without proper checking it might break the file structure as the two different content might not be compatible with one another. On first glance this seems to be a very small aspect, but the library being able to handle such exceptional scenarios adds to its overall stability. The flowchart of the save feature is represented in Figure 22.

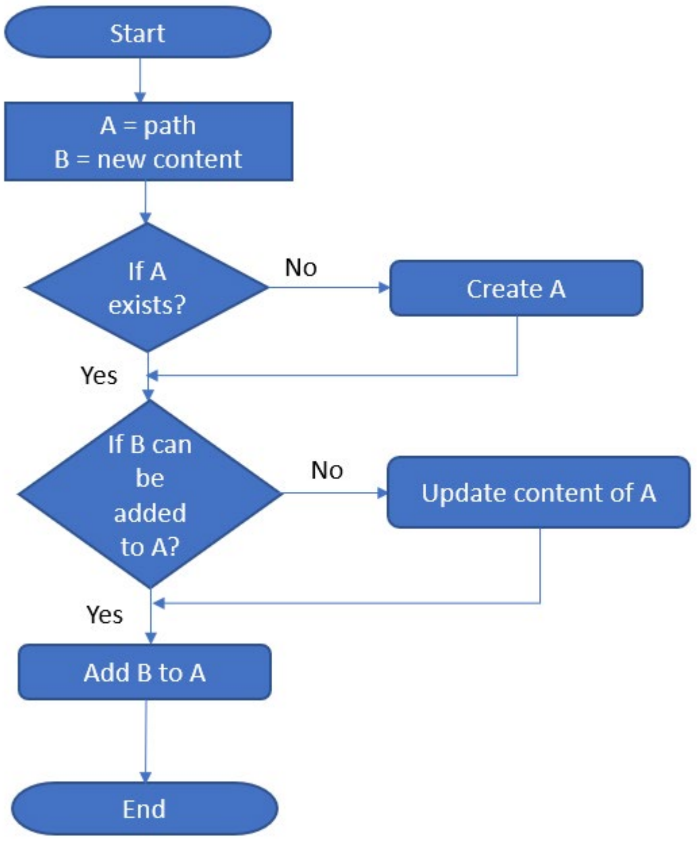


Figure 22. Flow chart for "save" feature in the distribution library.

Another example of fault tolerance in the model we have handled the 'install' feature with several layers to be able to handle more than one real-time scenario. The method corresponding to this feature is tasked to install script at a particular location of the node (Figure 23). However, we know that there exist numerous types of scripts with different methods of invoking each script. This library takes into consideration this aspect and is able to distinguish the type of script dynamically on the run time and thus takes necessary steps to compile if required or just simply execute the script. For now, the library supports four different types of scripts – Python, Ansible, C, and Java.

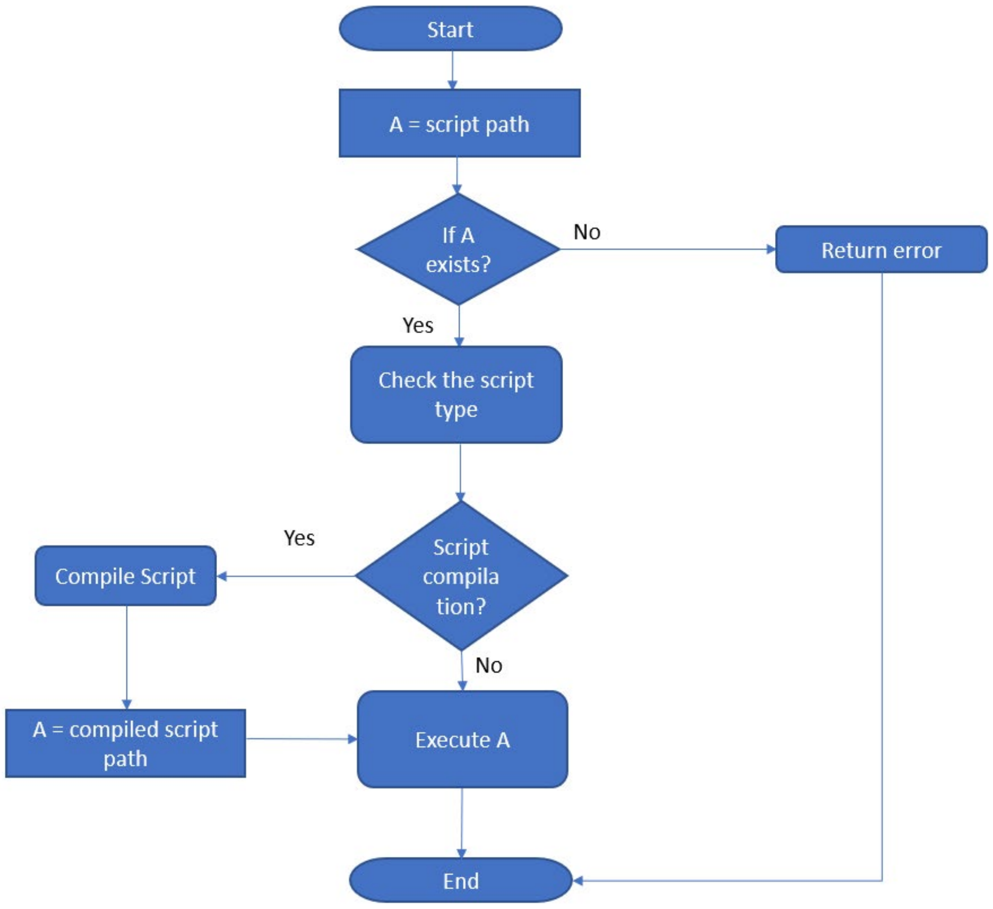


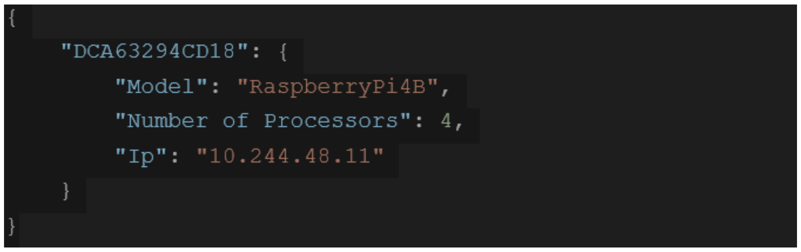
Figure 23. Flow chart for "install" feature in the distribution library

Another key point of this library is its **synchronous** nature. An input model with multiple tasks can be sent by some node to be executed in another node. Now the execution of the first task might take up a long time and might end up hampering or delaying the execution of other tasks. Keeping this in mind, the library has a secured thread execution of independent tasks which does not block the execution of other tasks.

D) Distribution Engine

The distribution engine is built with the idea of having a global singleton service with all the necessary modules to enforce our QoS metrices in a distributed system. The internal structure of the engine (Figure 19) shows the three major components of the engine, the **communication driver**, the **distributed library driver**, and the **remote hosts discovery** driver. This engine is also tasked with providing a level of abstraction to the user where in the user can just lay down their set of demands and requirements and the engine handles the translation of those demands into actions. This also adds a layer of security where in no user has direct access to our distribution engine itself.

The very first key functionality of a distribution system is to build its own identification and this engine caters to this requirement. Each fog node will have its own identity (Figure 24) built from the system statistics and can be shared with other hosts or fog nodes as metric for identification and security.



```
{
    "DCA63294CD18": {
        "Model": "RaspberryPi4B",
        "Number of Processors": 4,
        "Ip": "10.244.48.11"
    }
}
```

Figure 24. A typical fog node identification format

a) Remote Hosts Discovery

In a distributed system, it is crucial to know the environment. This means all the nodes in the distributed system should have the capability to know the existence of all the other nodes. To complete this step, we set up the engine with a host discovery driver (e.g., ‘systemd-resolve’, 'resolv.conf', 'avahi') which searches for all the other hosts connected to the same local network and are taking part in the distributed system. For the discovery of other hosts, we have used "Avahi", which is an open source zero-configuration implementation. The implementation of avahi also includes a multicast DNS/DNS-SD service discovery feature.

The functionality of this module is to register our devices in each of the other participating fog nodes using the avahi daemon.

In the run time the remote host discovery component in the distribution engine browses for the particular DNS names used to register our devices. The traditional way of registering a particular device is by using the communication protocol as the domain name, for example, if the devices are using MQTT protocol we register the devices as '_mqtt_tcp.loacal' or for HTTP protocols the registration is done as '_http_tcp.local' and it is binding on a particular port so the avahi browse service searches for domains with '.local' extensions and if they match the entire registration name it returns the device IP and MAC address. Hence, any other fog node which is not a part of the distribution system will not be registered as a device which serves as a measure to provide security against any phishing mechanism.

b) Distributed Library Driver

In the previous section, we have discussed the concept of the distribution library and the functionalities that can be achieved from it. In this part, we will discuss the necessary driver tools implemented in the distribution engine to control the flow of the library. All the methods described in the library are triggered by this driver in the engine, it derives the input or output model from the communication driver and is able to parse the model to be able to invoke the particular library method. We have two separate modules in this driver, the first is called *CommandAction* which is responsible for converting the input requirements to suitable service-related tasks in the system, and the second module is the *ResponseAction* and its tasked to handle the responses or outputs provided by other hosts in the distributed system. As our engine is able to parse and handle both the input and output model, we have two distinct modules for each model to have the least code complexity and better readability.

The *CommandAction* component is operated when the distribution engine receives an input model from users in the external environment or other host devices internal to the system. This driver has a mapping of all supported input features and their corresponding executable methods. So, all commands are carried out by this *CommandAction* module.

On the other hand,the *ResponseAction* module is used to handle the output model being sent from other nodes. This module has failure mechanisms implemented in it. For example, the action "_path exists" was executed in some host device but the response status was "False" with some error description. We can have an autonomic computing component of self-configuration implemented in this method where we can resend an input model with the 'create' feature set to true for that specific path. Another scenario, for the feature "service_monitor," if the status of a particular service in some host device was received as false meaning not active, we can send another input model with 'exc_cmd' feature enabled with the command for restarting the service. The default fault handling mechanism which is implemented in all the methods in the *ResponseAction* module is the notification protocol to inform the admin about the issue. This creates a knowledge base with all the different features and their respective failure which can later be used for developing newer self-healing concepts in the form of QoS agents.

c) Communication Driver

The communication driver is the basic communicator of the engine which has the responsibility of managing all the communication in between all the different host devices connected to the same network. There are a number of communication protocols that can be used in our model. The idea is to provide dynamicity to the distributed system as our model is not restricted to any particular protocol. By removing the dependency on any specific protocol, our model is in accordance with the QoS metric of scalability and reliability. The logic used behind the communication protocol structure is generic and is not dependent on a particular protocol, in our solution, we have used the messaging protocol called Message Queue Telemetry Transport (MQTT). However, the same messaging structure can be used by another protocol for example with the base implementation hypertext transfer protocol (HTTP) we can use our logic for all data messages going in and out of the engine. A brief introduction to MQTT, it is based on a publish/subscribe model. In this type of model, we have a client with the capability of publishing and subscribing to topics. Topics are message recognition paths and is set to be unique for each type of messages. The communication driver has two main components, the remote listener and the remote publisher. The listener component is the subscription to various topics and is used to wait for any communication from other hosts. The distinctive feature here is the ability to handle any data rate which seems challenging as each communication protocol has their own mechanisms to handle data rates. To counter this dependency, we have a generic queue implementation and so the distribution engine is not tied to any particular set of protocols. We can have another communication protocol implementation, for example the queue could be listening to a HTTP client and the flow of the distribution engine will be unhinged. Small internal modules such as this helps to make this system less dependent and specific for a particular platform but in turn enables it to be used as a building block for any other service or system in general. The remote publisher component, as the naming suggests, is used as a message transferring module in between different host devices. It is further divided into two parts- the commands publishing section and the response publishing section. The key difference between the two methods is commands publishing refers to the input model features being published to different hosts while the latter is used for the output model transmission. The reason for this segregation is mainly code optimization and for having better readability. In case of input model which consists of a varied number of features it is not viable for the entire block of features to be transmitted each time for a particular feature in need. It would also require more bandwidth which is definitely in violation of the basic QoS requirements. So, in this case the flag variable is checked for each feature and then only the feature in use is sent to the destined host. For the second case, the output model has a fixed structure, and it has no particular specification for a transfer, so the method is clearly a separate entity for publishing it to defined destinations. One more key feature is that for publishing we need to have different clients for publishing as each client can bind to one destination address at a time, so instead of creating a client for each transmission we record a mapping of each destination host address and a client corresponding to it and hence it can be reused each time for transmission to that specific host.

The next question is how this publish feature can be reused for other communication protocols thus not binding it to any specific domain or functionality. In future there can be another communication client with a HTTPs protocol implementation or Constrained Application Protocol (CoAP) implementation. The only aspect that needs to be updated is the client parameters, if it’s different from the MQTT protocols, otherwise no change is required as the message body and the topic since everything is generic. The only way to make sure the topic which is the message path in terms of HTTP APIs, does not involve any hardcoded values specific to any domain. Hence, we develop a unique path with host ids for both sender and receiver and the targeted service name which corresponds to the message body.

2.5 Discussion

We propose in this chapter our approach to overcome the challenges discussed of the thesis related to building a QoS enabled fog layer in an IoT platform. The first step was to define all the failures in an IoT platform. So, we discuss a number of failures in the fog layer of IoT platforms which we learned from our real-life deployment and through our literature review. These failures led us to the exact point of failures in the fog layer. As a solution to the discovered point of failures, we devised QoS requirements, and we developed two approaches to fulfill these requirements. The first approach is to enable QoS in a fog layer with singular fog node. In the fog nodes we developed granular modules which are generic and dynamic in nature to handle the different point of failures. Although we were able to establish a stable architecture for addressing the QoS requirements in the singular node fog layer, we soon faced the big hurdle of resource constraints and to overcome that issue we proposed our second approach where we used a distributed fog system comprising of multiple fog nodes with different resources and different capacities providing more options and pathways to manage and monitor all the QoS requirements. Our distributed approach comprised of a distribution model. We discussed in this chapter the internal components of the distribution model and its usage in providing the modules built in our first approach, an extended ability to overcome the difficulties and challenges pertaining to resource constraints and better management in the fog layer. Thus, with the two proposed approaches, we were able to provide a QoS enabled fog layer.

In our review of the existing solutions closer to our proposed approach of the distributed model in the fog layer, we have found one distributed fog system called ‘KubeEdge’ which provides distributed services in fog layer. KubeEdge is the most widely used solution until now [46]. Following, we will discuss KubeEdge and highlight the key differences between it and our model.

KubeEdge is a proposed infrastructure in the fog environment of IoT. It is mainly an extension of the existing cloud functionalities of Kubernetes [47] to the edge or fog layers of IoT. The cloud resources are shared among the fog layer nodes and the cloud layer servers [47]. This concept was built to overcome the fog layer challenges with singular fog nodes. The KubeEdge consist of a network protocol stack, a distributed storage, and a service for synchronization along with an agent at the edge. The features targeted by KubeEdge are - **Kubernetes-native support** which manages edge applications and edge devices in the cloud with fully compatible Kubernetes APIs, **Cloud-Edge Reliable Collaboration** which ensures reliable message delivery without loss over unstable cloud-edge network, **Edge Autonomy** ensuring edge nodes running autonomously and the applications in the edge run normally, when the cloud-edge network is unstable, or edge is offline and restarted, **Edge Devices Management** managing edge devices through Kubernetes native APIs, **Extremely Lightweight Edge Agent** (EdgeCore) for running on resource-constrained edge [49].

Compared to our distributed model, KubeEdge requires the use of dockerized containers having all the requirements fulfilled. Therefore, it’s not viable enough to be used dynamically with any other existing IoT platforms without bringing an extensive amount of overhead. In comparison to KubeEdge, we do not require setting up docker images thus occupying more space in the system. Our distribution model has a smaller memory size capable of handling similar functionalities of KubeEdge.

The KubeEdge does not have any QoS model in its architecture, so it is not capable of handling unknown situations. There is no self-learning model included in the architecture which is a key aspect of our QoS distributed model. Hence, in comparison to KubeEdge our system is more dynamic in supporting QoS in the fog layer.

In the next section, we will discuss the technical evaluation of our proposed approaches and observe the QoS metrices being satisfied with our distribution model.

Chapter 3

Technical Evaluation

We discuss in this section the evaluations and practical implementations with focus on how to use our distributed model to achieve different functionalities and how the QoS metrices can be enforced using the model. Following, we start by discussing the principles used to validate our approach.

3.1 QoS evaluation techniques in distributed service models

According to our literature review, there is no specific work addressing QoS for the fog layer. Most of the work in the literature has been done in the provisioning of QoS for the cloud layer [35] (i.e., including integration, separation, transparency, asynchronous resource management and performance principle). Therefore, one part of our research was to extend the application of the current QoS principles (applied in the cloud layer) to the fog layer. Following, we discuss our extension of these QoS principles.

The integration principle states that quality of services should have proper maintenance, configuration, and prediction capabilities throughout all the architectural layers. The QoS modules being transferred from one layer to another must consist of QoS configurations, resource guarantees and proper flow maintenance [35]. The separation principle depicts that all processes must have their own architectural structures. Some processes are prone to use high data transfer rate with low latency while some processes might have low data rate with low latency. In this example having separate architectural flow maintains the QoS [35]. The concept of abstraction is being stated by the transparency principle. The IoT services should not have direct knowledge of any background mechanisms of QoS monitoring and maintenance [35]. According to the asynchronous resource management principle, we should have a proper distinction between the functionality of each architectural components, especially the components which manages the access and control modeling system. In distributed environment communication, we have essential time constraints and having separate functionalities in the architectural components would help overcome any deadlock scenarios [35]. The performance concept is the measurement of following a generally agreed standard for the implementation of all architectural components. For example, a set of rules for routing protocols, data formats, criteria for avoiding multiplexing, etc. [35].

3.2 Validation of QoS Agents in Fog Node

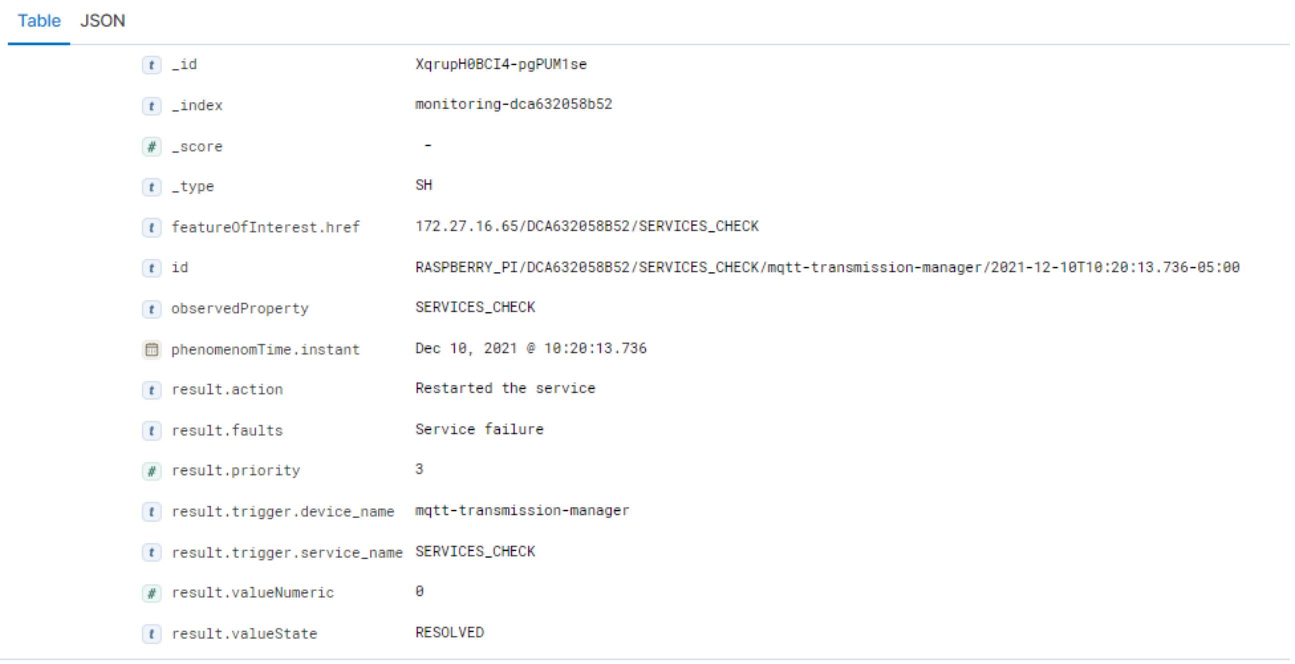
In this section we will discuss the results achieved through the AMI platform especially focusing on the results obtained by our QoS agents in the fog node. We deployed the AMI platform for a project duration of six months at an apartment in Sherbrooke, Canada. The objective of the project was to monitor the behavior change of a person based on the type of medication received at different stages of medical interventions. The hardware used for the tests and deployments is a Raspberry Pi4 with Raspbian OS. Next, we describe the tests and results for each QoS agents in a single fog node.

3.2.1 service monitoring

The test goals are to show the QoS agent for service monitoring is able to monitor the status of the running services in a fog node and is able to react in cases of anomaly situations. The QoS agent is crucial for maintaining **system reliability**.

For this test we set up a Raspberry Pi with a set of services running. We built the configuration for the QoS agent to monitor a particular service.

Table 11. Service check notifications in server database



_id	XqrupH0BCI4-pgPUM1se
_index	monitoring-dca632058b52
_score	-
_type	SH
featureOfInterest.href	172.27.16.65/DCA632058B52/SERVICES_CHECK
id	RASPBERRY_PI/DCA632058B52/SERVICES_CHECK/mqtt-transmission-manager/2021-12-10T10:20:13.736-05:00
observedProperty	SERVICES_CHECK
phenomenomTime.instant	Dec 10, 2021 @ 10:20:13.736
result.action	Restarted the service
result.faults	Service failure
result.priority	3
result.trigger.device_name	mqtt-transmission-manager
result.trigger.service_name	SERVICES_CHECK
result.valueNumeric	0
result.valueState	RESOLVED

Once the QoS agent was running, we opened up another terminal to manually switch the status of the services from ‘active’ to ‘inactive.’ The QoS agent was able to notice this change and took action to restart the services.

The status was checked again by the QoS agent to make sure that the service was running and then reported to the admin about the occurrence of the issue and that it was resolved due to the action taken by it (Table 11). The ‘phenomenon.instant’ in Table 11 shows the instant at which the service named ‘mqtt-transmission-manager’ (in the field ‘result.trigger.device_name’ of Table 11) had a failure. The fault is mentioned in the ‘result.faults’ in Table 11. The action taken by the agent is reflected in ‘result.action’ field. In this test the agent restarted the service to resolve the issue. The final state of the service is reflected in the field ‘result.valueState’ of Table 11.

The test was successful, the QoS agent was able to satisfy the requirements of the test goals. With respect to the requirements of the test, the QoS agent detected the inactive state of the running service, was able to inform the admin and even reacted to the situation by restarting the service which in turn resolved the status of the service.

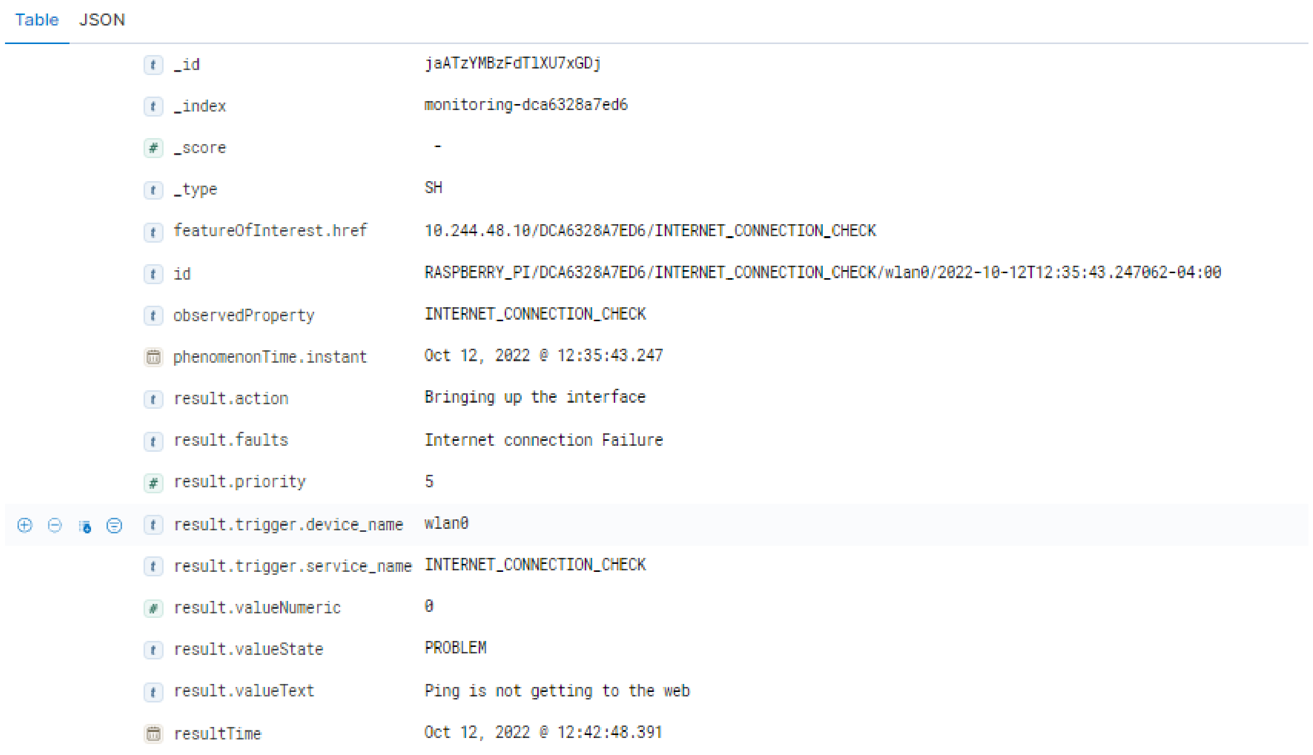
3.2.2 Internet connectivity

The goal for this test is to observe the QoS agent being able to detect whether internet connection is present and is then able to react to the situation. The main QoS metric targeted from this agent is to resolve the **offline capacity** of the platform.

The AMI platform had an existing self-healing component monitoring this point of failure, however, the action taken during the operation was to try and switch the Wi-Fi interface up and down hoping to reconnect to the network. While the measure taken is a must go to operation, but it failed in providing the knowledge of the up time and down time. We have built on this property a QoS agent which not only takes care of the pre-emptive measure but also records the timestamp of first disconnection until the internet connection is back up which is then sent to the server to inform it about the actual time the node went down which is an essential information to have. We manually disconnected a running Raspberry Pi for a period of seven minutes and the agent was able to specify the exact down time (Table 12). The 'phenomenonTime.instant' is the first recorded time of the occurrence of the issue and when the connection was brought back it sent the current time in 'resultTime.' The fault is mentioned in the 'result.faults' in Table 12. The action taken by the agent is reflected in 'result.action' field. In this test the agent tried to bring the wifi or ethernet interface up. The field 'result.valueText' shows the exact issue that led the agent to state that the internet connection is unavailable. The final state of the service is reflected in the field 'result.valueState' of Table 12.

The experiment was successful as the QoS agent was not only able to detect the absence of the internet connection but was also able to inform the admin about the exact time at which the connection was disrupted (Table 12).

Table 12. Internet connection status by QoS agent



_id	jaATzYMBzFdTlXU7xGDj
_index	monitoring-dca6328a7ed6
_score	-
_type	SH
featureOfInterest.href	10.244.48.10/DCA6328A7ED6/INTERNET_CONNECTION_CHECK
id	RASPBERRY_PI/DCA6328A7ED6/INTERNET_CONNECTION_CHECK/wlan0/2022-10-12T12:35:43.247062-04:00
observedProperty	INTERNET_CONNECTION_CHECK
phenomenonTime.instant	Oct 12, 2022 @ 12:35:43.247
result.action	Bringing up the interface
result.faults	Internet connection Failure
result.priority	5
result.trigger.device_name	wlan0
result.trigger.service_name	INTERNET_CONNECTION_CHECK
result.valueNumeric	0
result.valueState	PROBLEM
result.valueText	Ping is not getting to the web
resultTime	Oct 12, 2022 @ 12:42:48.391

3.2.3 Data recovery

The goal of the test is to successfully retrieve data from the fog nodes after the nodes have been offline for a period of time. This agent is another very crucial component for having **offline capabilities** and maintaining the **data consistency** in IoT platforms.

In our deployment of AMI platform in the Sherbrooke apartment, we came across a huge challenge in December 2021, when there was a security concern with the Apache Log4j implementation. Due to this security breach, all the servers around the globe were compromised and had to be shut down which was the case with our servers as well. This meant that there would not be any data being transferred to our cloud storage. However, with our implementation of the QoS agent for data recovery, we were not only able to avoid data loss but also were able to get the data to our cloud storage as soon as the servers were up without any technical intervention.

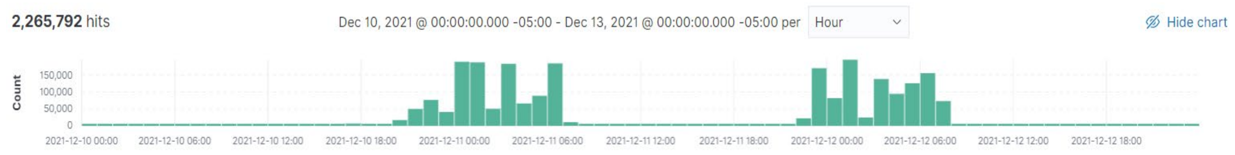


Figure 25. Data recovery during Log4j vulnerability

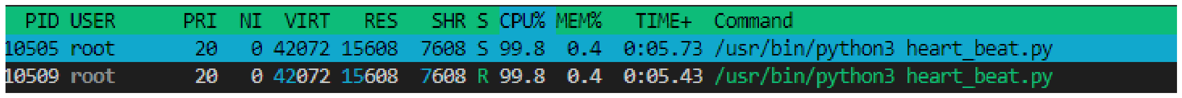
The pattern for the collection of data during the vulnerable period speaks greatly about the reliability of our platform (Figure 25). This agent goes hand in hand with the previous agent for monitoring the Internet connection, while the prior agent is essential for pre-emptive measures and building a proper knowledge base the current agent is mainly focused on maintaining the data consistency of the system in the IoT platform.

The data recovery QoS agent worked perfectly in a real-life deployment showcasing the successful recovery of data. The data during the infamous log4j vulnerability where all our servers were required to shut down until further investigation was successfully recovered and is depicted in (Figure 25).

3.2.4 QoS agent: resource constraint

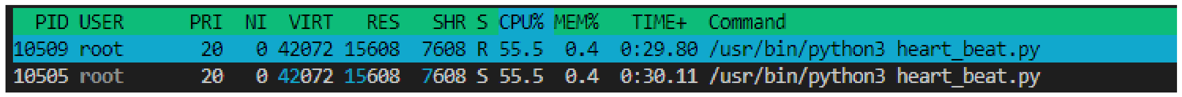
The test goals are to observe the QoS agent being able to monitor the resources of the fog node and is able to react to overloading scenarios to avoid the fog nodes from stop functioning. This agent targets the QoS metric of resource constraints and thus provides a stable and reliable platform.

The AMI platform came equipped with resource constrain measures, where in the platform was able to raise alerts in cases of overuse of CPU and disk resources. However, it was not sufficient to satisfy the QoS requirement of the system with lower resource capacities. The QoS agent for resource management is subdivided into three categories, as discussed in chapter 2. In this test, we try to run a service which initially takes more CPU resource around 99.8% (Figure 26) than the allowed threshold value and we capture the CPU resources being cut to 55.5% for that service by the agent in Figure 27.



```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
10505 root       20   0 42072 15608  7608 S 99.8  0.4  0:05.73 /usr/bin/python3 heart_beat.py
10509 root       20   0 42072 15608  7608 R 99.8  0.4  0:05.43 /usr/bin/python3 heart_beat.py
```

Figure 26. Initial CPU% by a process



```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
10509 root       20   0 42072 15608  7608 R 55.5  0.4  0:29.80 /usr/bin/python3 heart_beat.py
10505 root       20   0 42072 15608  7608 S 55.5  0.4  0:30.11 /usr/bin/python3 heart_beat.py
```

Figure 27. CPU usage after the QoS agent action

Also, for the knowledge base we inform the server for the issue (Table 13). The ‘phenomenonTime.instant’ is the exact moment when the agent recorded the cpu usage for the service to be more than the threshold. The fault is mentioned in the ‘result.faults’ in Table 13. The action taken by the agent is reflected in ‘result.action’ field. The field ‘result.valueText’ shows the exact issue that led to the agent to raise a flag for excessive CPU usage. The final state of the service is reflected in the field ‘result.valueState’ of Table 13.

Table 13. Notification for higher usage of CPU resource

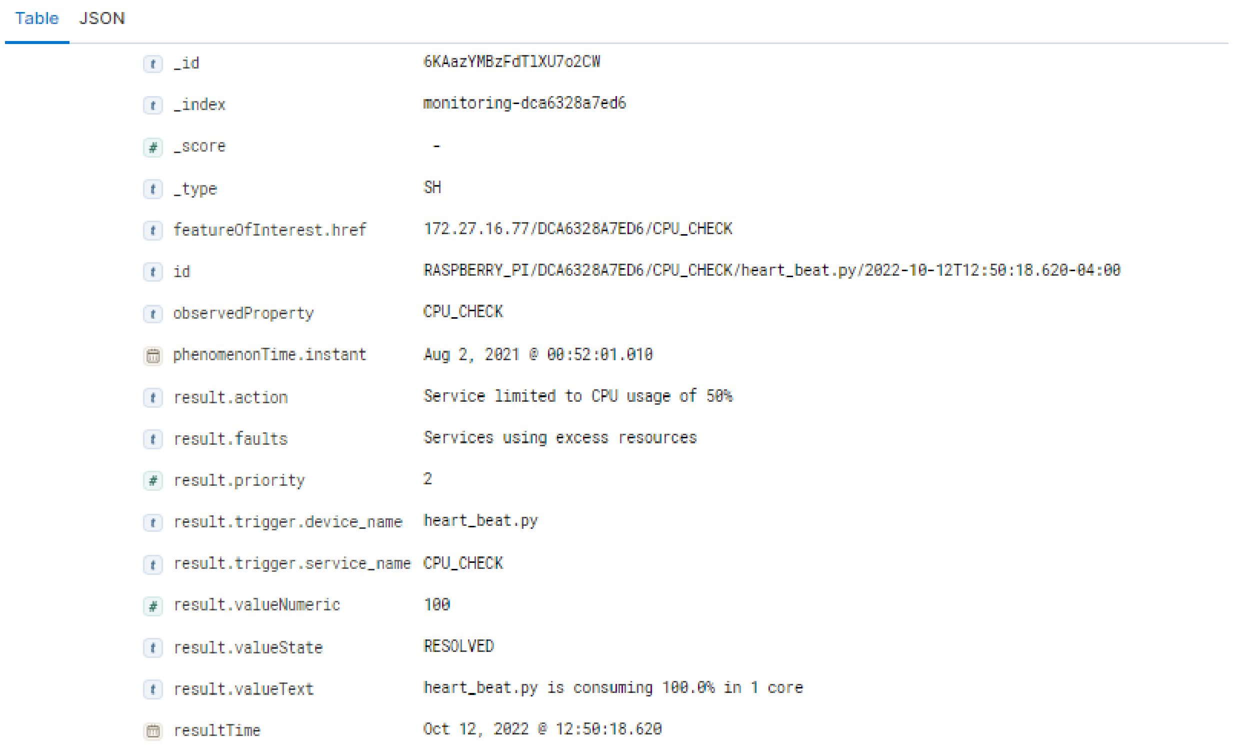
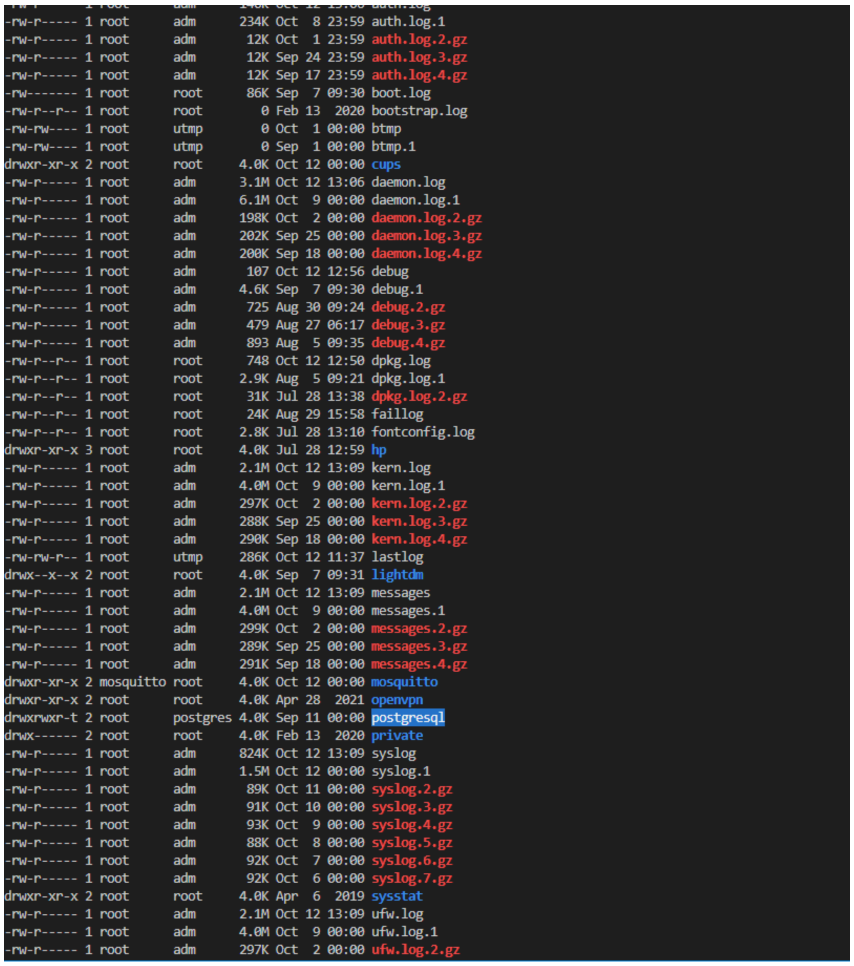


Table JSON

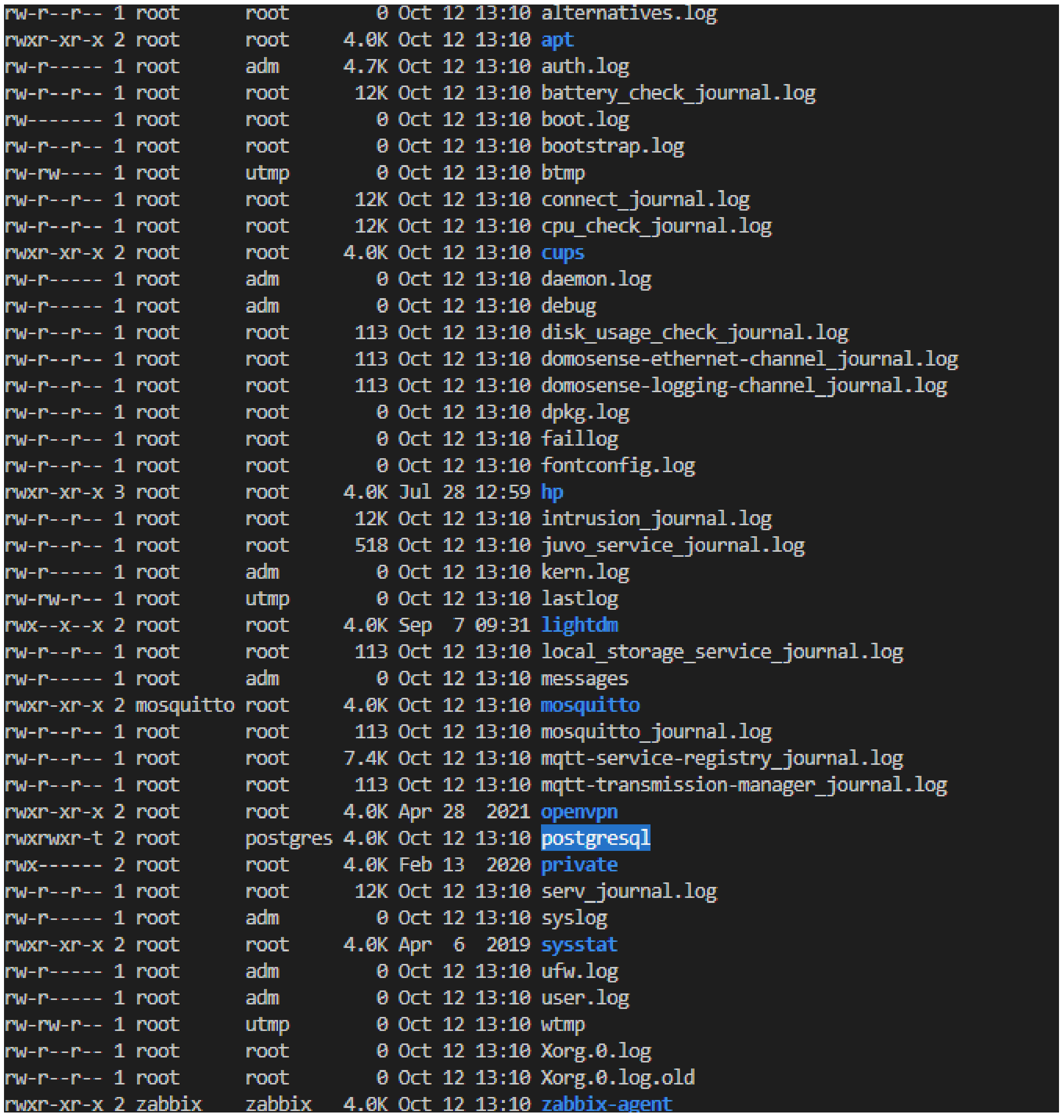
_id	6KAazYMBzFdT1XU7o2CW
_index	monitoring-dca6328a7ed6
_score	-
_type	SH
featureOfInterest.href	172.27.16.77/DCA6328A7ED6/CPU_CHECK
id	RASPBERRY_PI/DCA6328A7ED6/CPU_CHECK/heart_beat.py/2022-10-12T12:50:18.620-04:00
observedProperty	CPU_CHECK
phenomenonTime.instant	Aug 2, 2021 @ 00:52:01.010
result.action	Service limited to CPU usage of 50%
result.faults	Services using excess resources
result.priority	2
result.trigger.device_name	heart_beat.py
result.trigger.service_name	CPU_CHECK
result.valueNumeric	100
result.valueState	RESOLVED
result.valueText	heart_beat.py is consuming 100.0% in 1 core
resultTime	Oct 12, 2022 @ 12:50:18.620

The disk space management by the QoS agent was tested during our deployment in the Sherbrooke apartment as we met with a challenge of memory leak in the system which filled the daemon log files and crashed the node. For this test we lowered the threshold value to 20 percent to trigger the action from the QoS agent. We observe the log files before and after the QoS agent takes action. Before the QoS agent the disk space was filled with unmonitored logs (Figure 28). With the QoS agent we were able to save the latest logs for all our running services and truncated all the log files to free up disk space (Figure *29*). We can see that after the action there are new journal logs for services and the unnecessary log files have been truncated.



```
-rw-r----- 1 root      adm      234K Oct  8 23:59 auth.log.1
-rw-r----- 1 root      adm       12K Oct  1 23:59 auth.log.2.gz
-rw-r----- 1 root      adm       12K Sep 24 23:59 auth.log.3.gz
-rw-r----- 1 root      adm       12K Sep 17 23:59 auth.log.4.gz
-rw------- 1 root      root      86K Sep  7 09:30 boot.log
-rw-r--r-- 1 root      root        0 Feb 13  2020 bootstrap.log
-rw-rw---- 1 root      utmp        0 Oct  1 00:00 btmp
-rw-rw---- 1 root      utmp        0 Sep  1 00:00 btmp.1
drwxr-xr-x 2 root      root     4.0K Oct 12 00:00 cups
-rw-r----- 1 root      adm      3.1M Oct 12 13:06 daemon.log
-rw-r----- 1 root      adm      6.1M Oct  9 00:00 daemon.log.1
-rw-r----- 1 root      adm      198K Oct  2 00:00 daemon.log.2.gz
-rw-r----- 1 root      adm      202K Sep 25 00:00 daemon.log.3.gz
-rw-r----- 1 root      adm      200K Sep 18 00:00 daemon.log.4.gz
-rw-r----- 1 root      adm       107 Oct 12 12:56 debug
-rw-r----- 1 root      adm      4.6K Sep  7 09:30 debug.1
-rw-r----- 1 root      adm       725 Aug 30 09:24 debug.2.gz
-rw-r----- 1 root      adm       479 Aug 27 06:17 debug.3.gz
-rw-r----- 1 root      adm       893 Aug  5 09:35 debug.4.gz
-rw-r--r-- 1 root      root      748 Oct 12 12:50 dpkg.log
-rw-r--r-- 1 root      root     2.9K Aug  5 09:21 dpkg.log.1
-rw-r--r-- 1 root      root      31K Jul 28 13:38 dpkg.log.2.gz
-rw-r--r-- 1 root      root      24K Aug 29 15:58 faillog
-rw-r--r-- 1 root      root     2.8K Jul 28 13:10 fontconfig.log
drwxr-xr-x 3 root      root     4.0K Jul 28 12:59 hp
-rw-r----- 1 root      adm      2.1M Oct 12 13:09 kern.log
-rw-r----- 1 root      adm      4.0M Oct  9 00:00 kern.log.1
-rw-r----- 1 root      adm      297K Oct  2 00:00 kern.log.2.gz
-rw-r----- 1 root      adm      288K Sep 25 00:00 kern.log.3.gz
-rw-r----- 1 root      adm      290K Sep 18 00:00 kern.log.4.gz
-rw-rw-r-- 1 root      utmp     286K Oct 12 11:37 lastlog
drwx--x--x 2 root      root     4.0K Sep  7 09:31 lightdm
-rw-r----- 1 root      adm      2.1M Oct 12 13:09 messages
-rw-r----- 1 root      adm      4.0M Oct  9 00:00 messages.1
-rw-r----- 1 root      adm      299K Oct  2 00:00 messages.2.gz
-rw-r----- 1 root      adm      289K Sep 25 00:00 messages.3.gz
-rw-r----- 1 root      adm      291K Sep 18 00:00 messages.4.gz
drwxr-xr-x 2 mosquitto root     4.0K Oct 12 00:00 mosquitto
drwxr-xr-x 2 root      root     4.0K Apr 28  2021 openvpn
drwxrwxr-t 2 root      postgres 4.0K Sep 11 00:00 postgresql
drwx------ 2 root      root     4.0K Feb 13  2020 private
-rw-r----- 1 root      adm      824K Oct 12 13:09 syslog
-rw-r----- 1 root      adm      1.5M Oct 12 00:00 syslog.1
-rw-r----- 1 root      adm       89K Oct 11 00:00 syslog.2.gz
-rw-r----- 1 root      adm       91K Oct 10 00:00 syslog.3.gz
-rw-r----- 1 root      adm       93K Oct  9 00:00 syslog.4.gz
-rw-r----- 1 root      adm       88K Oct  8 00:00 syslog.5.gz
-rw-r----- 1 root      adm       92K Oct  7 00:00 syslog.6.gz
-rw-r----- 1 root      adm       92K Oct  6 00:00 syslog.7.gz
drwxr-xr-x 2 root      root     4.0K Apr  6  2019 sysstat
-rw-r----- 1 root      adm      2.1M Oct 12 13:09 ufw.log
-rw-r----- 1 root      adm      4.0M Oct  9 00:00 ufw.log.1
-rw-r----- 1 root      adm      297K Oct  2 00:00 ufw.log.2.gz
```

Figure 28. Disk usage reaching the threshold value

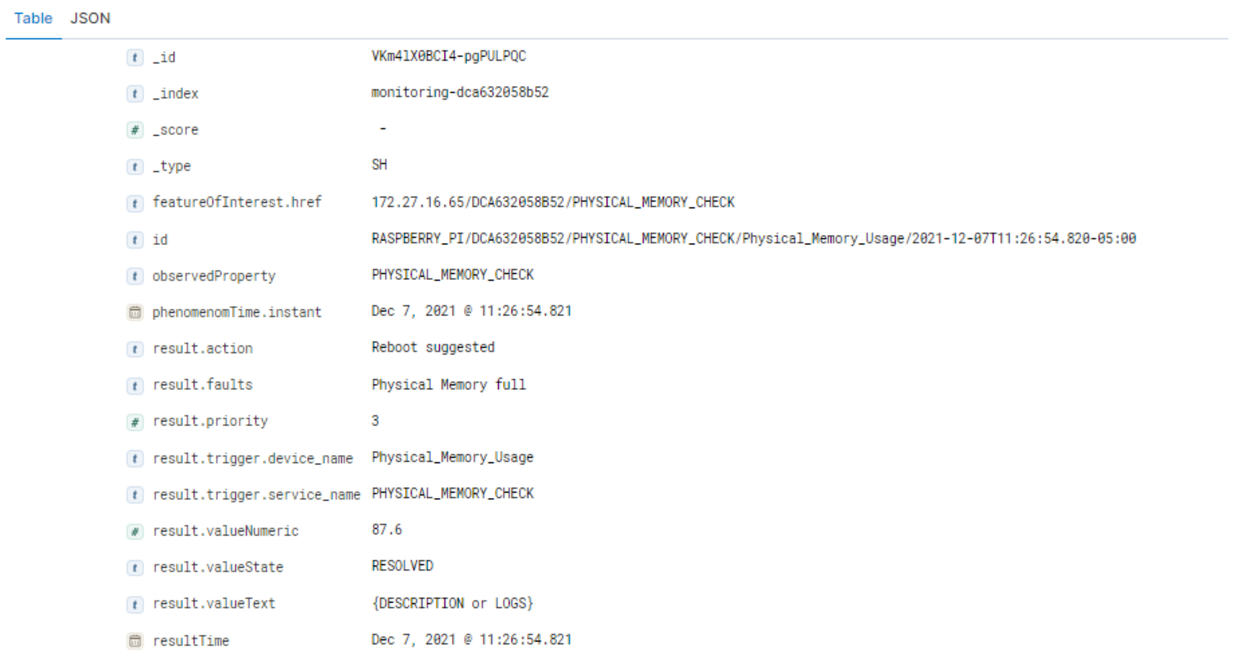


```
rw-r--r-- 1 root      root        0 Oct 12 13:10 alternatives.log
rwxr-xr-x 2 root      root     4.0K Oct 12 13:10 apt
rw-r----- 1 root      adm      4.7K Oct 12 13:10 auth.log
rw-r--r-- 1 root      root      12K Oct 12 13:10 battery_check_journal.log
rw------- 1 root      root        0 Oct 12 13:10 boot.log
rw-r--r-- 1 root      root        0 Oct 12 13:10 bootstrap.log
rw-rw---- 1 root      utmp        0 Oct 12 13:10 btmp
rw-r--r-- 1 root      root      12K Oct 12 13:10 connect_journal.log
rw-r--r-- 1 root      root      12K Oct 12 13:10 cpu_check_journal.log
rwxr-xr-x 2 root      root     4.0K Oct 12 13:10 cups
rw-r----- 1 root      adm         0 Oct 12 13:10 daemon.log
rw-r----- 1 root      adm         0 Oct 12 13:10 debug
rw-r--r-- 1 root      root      113 Oct 12 13:10 disk_usage_check_journal.log
rw-r--r-- 1 root      root      113 Oct 12 13:10 domosense-ethernet-channel_journal.log
rw-r--r-- 1 root      root      113 Oct 12 13:10 domosense-logging-channel_journal.log
rw-r--r-- 1 root      root        0 Oct 12 13:10 dpkg.log
rw-r--r-- 1 root      root        0 Oct 12 13:10 faillog
rw-r--r-- 1 root      root        0 Oct 12 13:10 fontconfig.log
rwxr-xr-x 3 root      root     4.0K Jul 28 12:59 hp
rw-r--r-- 1 root      root      12K Oct 12 13:10 intrusion_journal.log
rw-r--r-- 1 root      root      518 Oct 12 13:10 juvo_service_journal.log
rw-r----- 1 root      adm         0 Oct 12 13:10 kern.log
rw-rw-r-- 1 root      utmp        0 Oct 12 13:10 lastlog
rwx--x--x 2 root      root     4.0K Sep  7 09:31 lightdm
rw-r--r-- 1 root      root      113 Oct 12 13:10 local_storage_service_journal.log
rw-r----- 1 root      adm         0 Oct 12 13:10 messages
rwxr-xr-x 2 mosquitto root     4.0K Oct 12 13:10 mosquitto
rw-r--r-- 1 root      root      113 Oct 12 13:10 mosquitto_journal.log
rw-r--r-- 1 root      root     7.4K Oct 12 13:10 mqtt-service-registry_journal.log
rw-r--r-- 1 root      root      113 Oct 12 13:10 mqtt-transmission-manager_journal.log
rwxr-xr-x 2 root      root     4.0K Apr 28  2021 openvpn
rwxrwxr-t 2 root      postgres 4.0K Oct 12 13:10 postgresql
rwx------ 2 root      root     4.0K Feb 13  2020 private
rw-r--r-- 1 root      root      12K Oct 12 13:10 serv_journal.log
rw-r----- 1 root      adm         0 Oct 12 13:10 syslog
rwxr-xr-x 2 root      root     4.0K Apr  6  2019 sysstat
rw-r----- 1 root      adm         0 Oct 12 13:10 ufw.log
rw-r----- 1 root      adm         0 Oct 12 13:10 user.log
rw-rw-r-- 1 root      utmp        0 Oct 12 13:10 wtmp
rw-r--r-- 1 root      root        0 Oct 12 13:10 Xorg.0.log
rw-r--r-- 1 root      root        0 Oct 12 13:10 Xorg.0.log.old
rwxr-xr-x 2 zabbix    zabbix   4.0K Oct 12 13:10 zabbix-agent
```

Figure 29. Disk space management by QoS agent

Next, the physical memory management is generally a defense mechanism in case of certain services using huge RAM space. However, the only solution for such cases is to raise a high alert to the admin and the admin going on site to take care of the issue (Table 14). The ‘phenomenonTime.instant’ is the exact moment when the agent recorded the physical memory usage to be more than the threshold. The fault is mentioned in the ‘result.faults’ in Table 14. The action suggested by the agent is reflected in ‘result.action’ field. Here the agent suggests to the user to make a reboot in order to resolve the issue.

Table 14. An example of RAM space being almost full



_id	VKm41X0BCI4-pgPULPQC
_index	monitoring-dca632058b52
_score	-
_type	SH
featureOfInterest.href	172.27.16.65/DCA632058B52/PHYSICAL_MEMORY_CHECK
id	RASPBERRY_PI/DCA632058B52/PHYSICAL_MEMORY_CHECK/Physical_Memory_Usage/2021-12-07T11:26:54.820-05:00
observedProperty	PHYSICAL_MEMORY_CHECK
phenomenomTime.instant	Dec 7, 2021 @ 11:26:54.821
result.action	Reboot suggested
result.faults	Physical Memory full
result.priority	3
result.trigger.device_name	Physical_Memory_Usage
result.trigger.service_name	PHYSICAL_MEMORY_CHECK
result.valueNumeric	87.6
result.valueState	RESOLVED
result.valueText	{DESCRIPTION or LOGS}
resultTime	Dec 7, 2021 @ 11:26:54.821

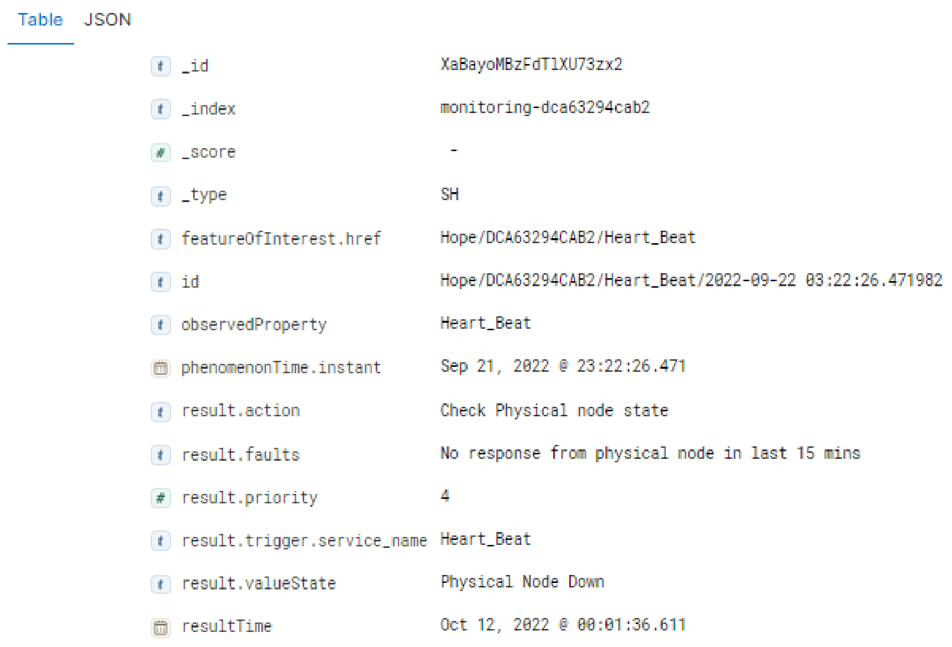
The tests were successful for all the monitored fog node resources. For the CPU, disk and RAM resources, the QoS agents were able to detect the overloading situation and react to it by reducing the usage and then also provided the admin with the knowledge of the situation.

3.2.5 Heartbeat

The goal is to test the fog node and cloud being able to acknowledge the presence of one another at all times. This agent helps in providing the system **availability**, **stability** of the IoT platform and hence the overall **reliability**.

The **heartbeat** agent is tested mainly from the server side, we deployed a Raspberry Pi4 in our lab and after the initial connection to our virtual node in the cloud we turned down the OpenVPN connection which resulted in no communication between the fog node and the cloud node. The cloud node not being able to detect the presence of the corresponding fog node for a period of time, informed the admin server with a high alert message denoting the fog node being down (Table 15). The 'phenomenon.instant' in Table 15 shows the first instance at which the cloud was not able to get any acknowledgement from the fog node.

Table 15. Example notification for heartbeat



_id	XaBayoMBzFdT1XU73zx2
_index	monitoring-dca63294cab2
_score	-
_type	SH
featureOfInterest.href	Hope/DCA63294CAB2/Heart_Beat
id	Hope/DCA63294CAB2/Heart_Beat/2022-09-22 03:22:26.471982
observedProperty	Heart_Beat
phenomenonTime.instant	Sep 21, 2022 @ 23:22:26.471
result.action	Check Physical node state
result.faults	No response from physical node in last 15 mins
result.priority	4
result.trigger.service_name	Heart_Beat
result.valueState	Physical Node Down
resultTime	Oct 12, 2022 @ 00:01:36.611

The fault is mentioned in the 'result.faults' in Table 15. In this test the fault was detected by the cloud agent which tried communicating with the agent in the fog node for 15 minutes but was not able to do so. After 15 mins the cloud agent notified the admin and thus the 'resultTime' is different from phenomenon.instant. The action suggested by the agent is reflected in 'result.action' field. The final state of the service is reflected in the field 'result.valueState' of Table 15 and the service name is mentioned in the field 'result.trigger.service_name' of Table 15.

The test was successfully executed as the heartbeat agents were able to detect the absence of the other agent and raised an alarm for the admin.

In the next section, we will discuss the tests performed on the distributed model and validate our mode with the results.

3.3 Validation of the Distributed model

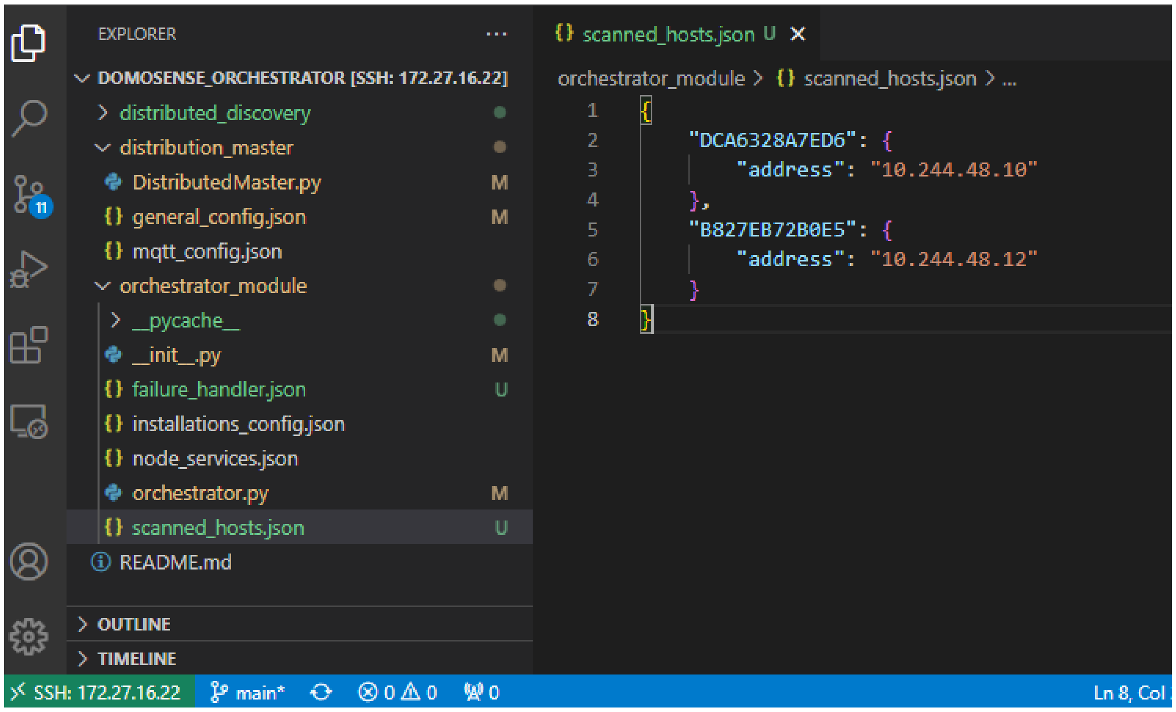
For the test of our distribution model, we set up three fog nodes consisting of two Raspberry Pi4 and one Raspberry Pi3. Each had the Raspbian OS setup with initial package installations. We will be showcasing the usage of the model to counter the drawbacks that we came across in the singular node fog system. The main targeted QoS metrices are reliability, scalability, offline capacity, security.

We choose one of the Raspberry Pi4 as the master node while the other Raspberry Pi4 and Raspberry Pi3 act as the slave nodes. The distribution model can be used to implement any number of services, for our evaluation of metrices we have implemented the default services which are tasked to create and maintain the communication channel in the distributed system. It is with the help of these services we are able to satisfy the intended QoS metrices.

3.3.1 Establishing a Communication Channel

The goal of the test is for the fog nodes being able to identify all the other fog nodes in the distributed environment.

The first functionality in our distribution engine is the remote host discovery service which is capable of recognizing all the registered fog nodes and saves that in a file in the system for future use (Figure 30). Thus, no manual configuration is needed at the master node level for it to recognize the other participating nodes.

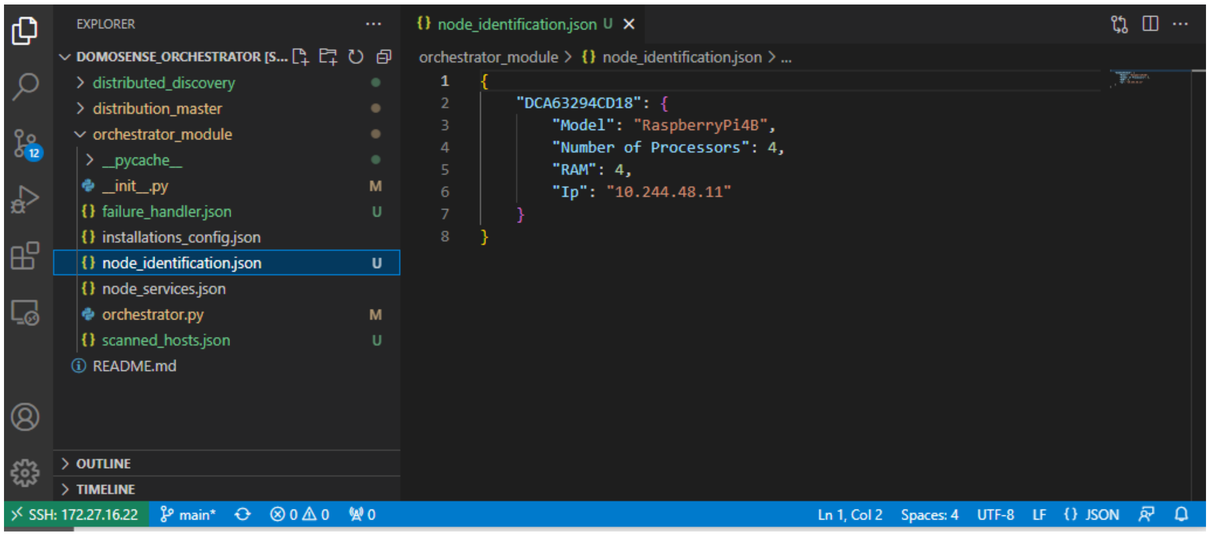


```
{
    "DCA6328A7ED6": {
        "address": "10.244.48.10"
    },
    "B827EB72B0E5": {
        "address": "10.244.48.12"
    }
}
```

Figure 30. Master node discovering remote hosts

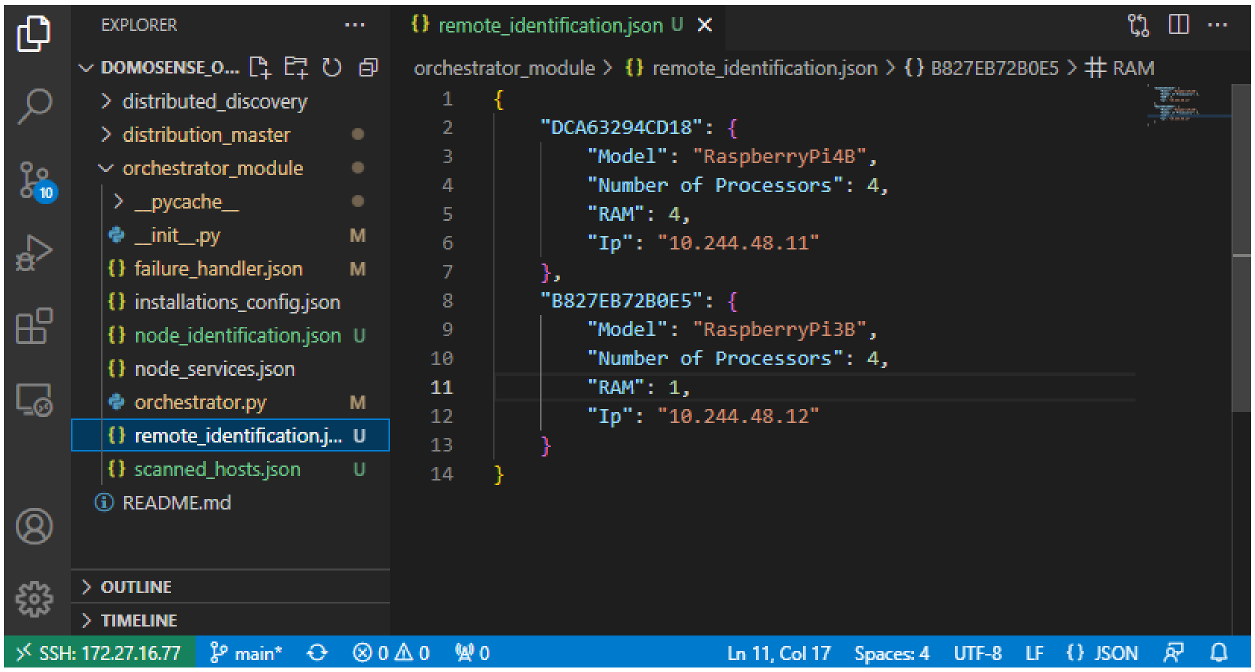
Each node has a standard data format (model, number of processors, RAM, Ip address) for their identification which helps to build a proper management system (Figure 31). The nodes share their identification to one another, thus setting up the communication path in between them (Figure *32*). The QoS metric that is satisfied in this service is the layer of security. The discovery service only accepts hosts with a defined host name so any external node will not be a part of the distributed system. The slave nodes will be directed by the master node to have the identification of other nodes so even in this case the communication loop is closed to the nodes participating in the distributed system.

The test was successful, we observed all the fog nodes being able to detect the other fog nodes in their configuration files with details of their respective internal system components.



```
{
    "DCA63294CD18": {
        "Model": "RaspberryPi4B",
        "Number of Processors": 4,
        "RAM": 4,
        "Ip": "10.244.48.11"
    }
}
```

Figure 31. Node identification built by each fog node



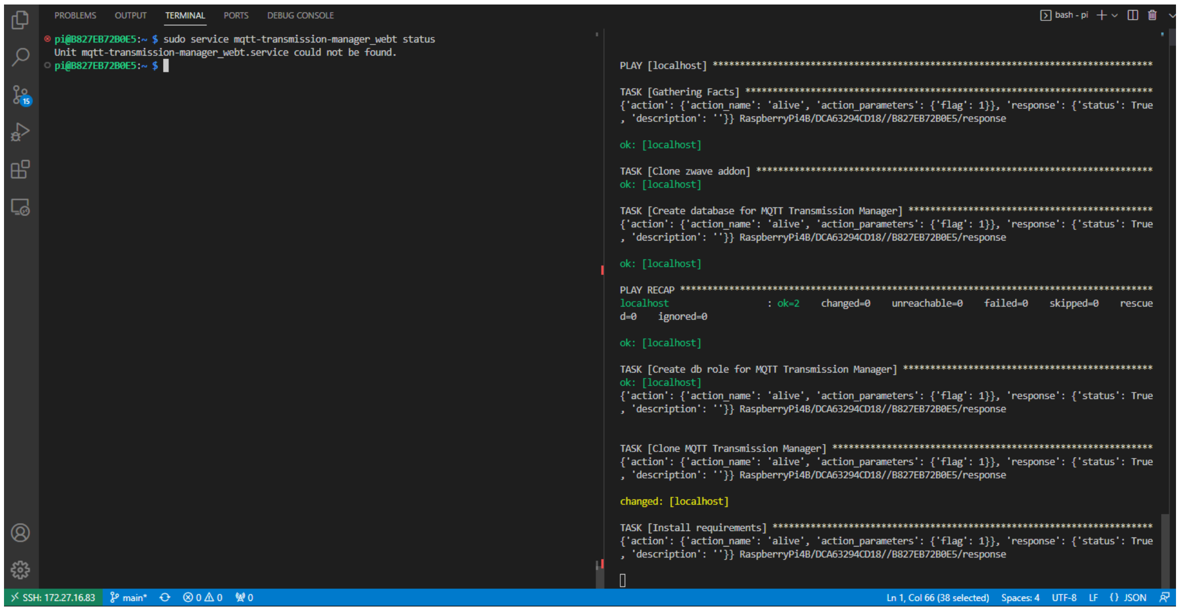
```
{
    "DCA63294CD18": {
        "Model": "RaspberryPi4B",
        "Number of Processors": 4,
        "RAM": 4,
        "Ip": "10.244.48.11"
    },
    "B827EB72B0E5": {
        "Model": "RaspberryPi3B",
        "Number of Processors": 4,
        "RAM": 1,
        "Ip": "10.244.48.12"
    }
}
```

Figure 32. Identification of remote fog nodes

3.3.2 Autonomous installations

The goal of the test is to install services in remote hosts directed by the master node in the distributed environment. With this validation we target the QoS metric of **scalability** in which we are able to deploy multiple nodes at run time without user intervention.

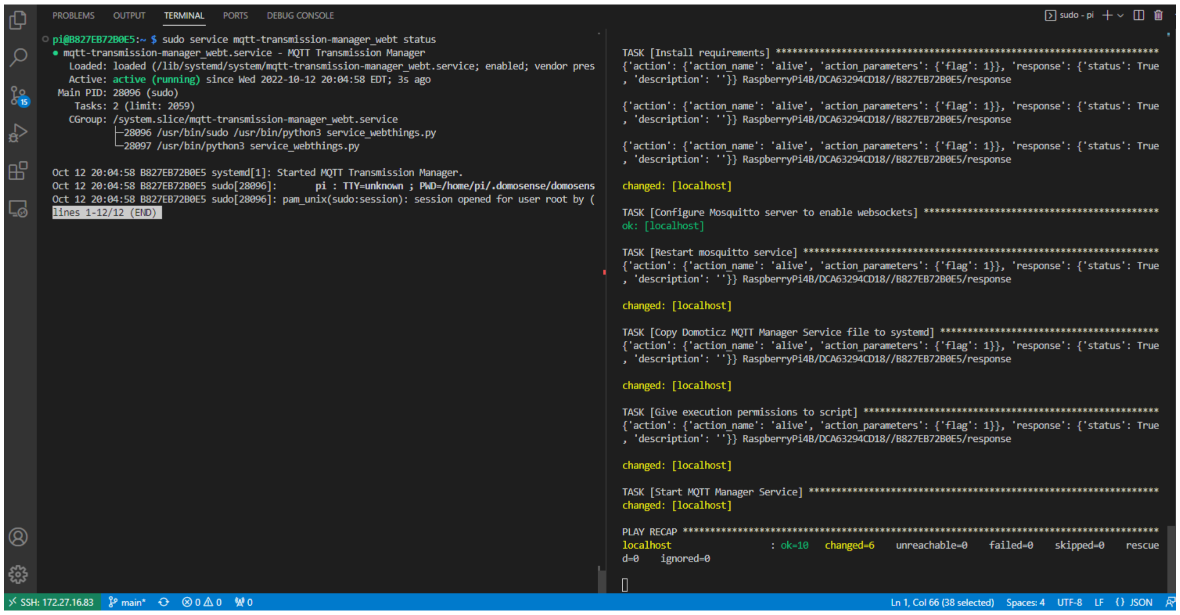
In section 2.4.3 we came across this challenge in our real-life deployment where we had to hand build all the node installations and that took several hours to complete. In this part we will see dynamic distribution of services to different nodes based on a configuration file stating the requirements of each node. For this experiment we took three fog nodes and were able to get their required set of services running after the distribution of tasks from the master node was completed. We check the status of a service before and after the installation script is invoked by the master node. The status of the service is observed to be not found while the installation is in progress(Figure 33) and at the instant the service installation is completed we can see the active status for the same service (Figure 34).



```
pi@B827EB72B0E5:~ $ sudo service mqtt-transmission-manager_webt status
Unit mqtt-transmission-manager_webt.service could not be found.
pi@B827EB72B0E5:~ $
```

```
PLAY [localhost] ******************************************************************************

TASK [Gathering Facts] ************************************************************************
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

ok: [localhost]

TASK [Clone zwave addon] **********************************************************************
ok: [localhost]

TASK [Create database for MQTT Transmission Manager] ******************************************
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

ok: [localhost]

PLAY RECAP ************************************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescue
d=0    ignored=0

ok: [localhost]

TASK [Create db role for MQTT Transmission Manager] *******************************************
ok: [localhost]
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response


TASK [Clone MQTT Transmission Manager] ********************************************************
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

changed: [localhost]

TASK [Install requirements] *******************************************************************
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

```

Figure 33. Service status while installation is in progress



```
pi@B827EB72B0E5:~ $ sudo service mqtt-transmission-manager_webt status
● mqtt-transmission-manager_webt.service - MQTT Transmission Manager
   Loaded: loaded (/lib/systemd/system/mqtt-transmission-manager_webt.service; enabled; vendor pres
   Active: active (running) since Wed 2022-10-12 20:04:58 EDT; 3s ago
 Main PID: 28096 (sudo)
    Tasks: 2 (limit: 2059)
   CGroup: /system.slice/mqtt-transmission-manager_webt.service
           ├─28096 /usr/bin/sudo /usr/bin/python3 service_webthings.py
           └─28097 /usr/bin/python3 service_webthings.py

Oct 12 20:04:58 B827EB72B0E5 systemd[1]: Started MQTT Transmission Manager.
Oct 12 20:04:58 B827EB72B0E5 sudo[28096]:       pi : TTY=unknown ; PWD=/home/pi/.domosense/domosens
Oct 12 20:04:58 B827EB72B0E5 sudo[28096]: pam_unix(sudo:session): session opened for user root by (
lines 1-12/12 (END)
```

```
TASK [Install requirements] ***************************************************************
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

changed: [localhost]

TASK [Configure Mosquitto server to enable websockets] ************************************
ok: [localhost]

TASK [Restart mosquitto service] **********************************************************
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

changed: [localhost]

TASK [Copy Domoticz MQTT Manager Service file to systemd] *********************************
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

changed: [localhost]

TASK [Give execution permissions to script] ***********************************************
{'action': {'action_name': 'alive', 'action_parameters': {'flag': 1}}, 'response': {'status': True
, 'description': ''}} RaspberryPi4B/DCA63294CD18//B827EB72B0E5/response

changed: [localhost]

TASK [Start MQTT Manager Service] *********************************************************
changed: [localhost]

PLAY RECAP ********************************************************************************
localhost                  : ok=10   changed=6    unreachable=0    failed=0    skipped=0    rescue
d=0    ignored=0
```

Figure 34. Service status when installation is completed.

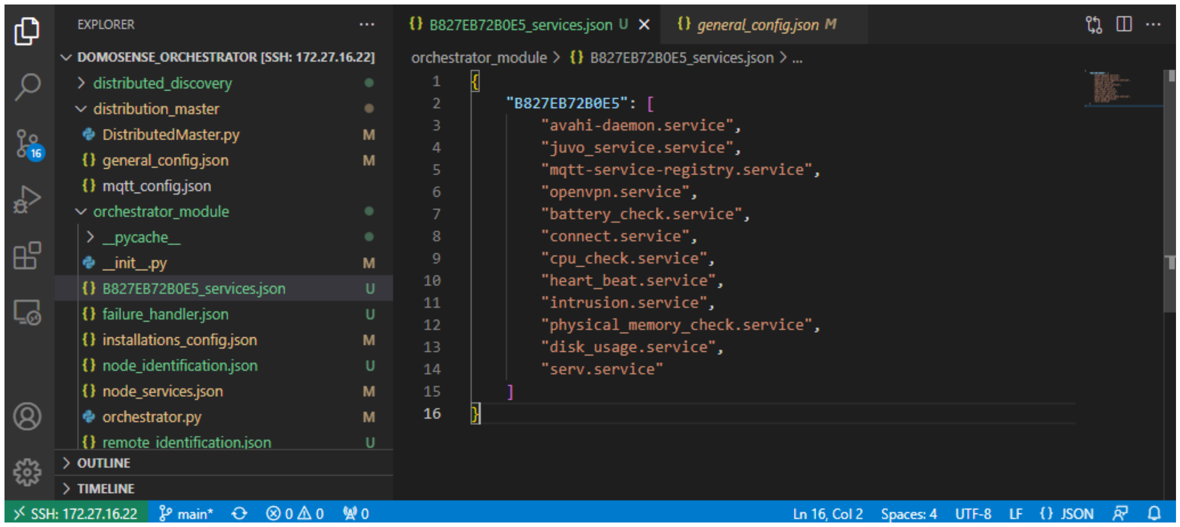
The service named 'mqtt-transmission-manager_webt' was successfully installed in a remote node after being communicated by the master node to run that service installation in that particular node. With regards to time, as all the nodes are capable of installing their tasks at the same time, the entire deployment takes on an average the same time as one node which in our case of implementation is approximately 20 minutes for three nodes which, on the other hand, would have taken around 20 mins for each node so approximately 60 minutes.

3.3.3 QoS monitoring and Management

The goal of the test is to prove that the master node has the capability to monitor and manage all the services running in all the remote hosts. This feature helps in proper monitoring of all services in the distributed system, thus maintaining system **reliability** with **less resource utilization**.

All the slave nodes have a feature in which they create a list of all the running services in the node and send it to the master node (Figure 35). The master node thus has an eagle-eyed view of all the services that are being provided by the entire distributed system. Referring to the QoS agent of service monitoring in singular fog nodes (section 3.4.3 A), we extend that idea here where the QoS agent in master node is capable of monitoring the service status of all running services in the distributed system. Say the QoS agents in every node takes up x% of the entire resource in the distributed system, but with this feature it is reduced to x/number of nodes percent.

In this test we can showcase that the master node has the knowledge of all the running services in the distributed system, thus resulting in a success for the test.



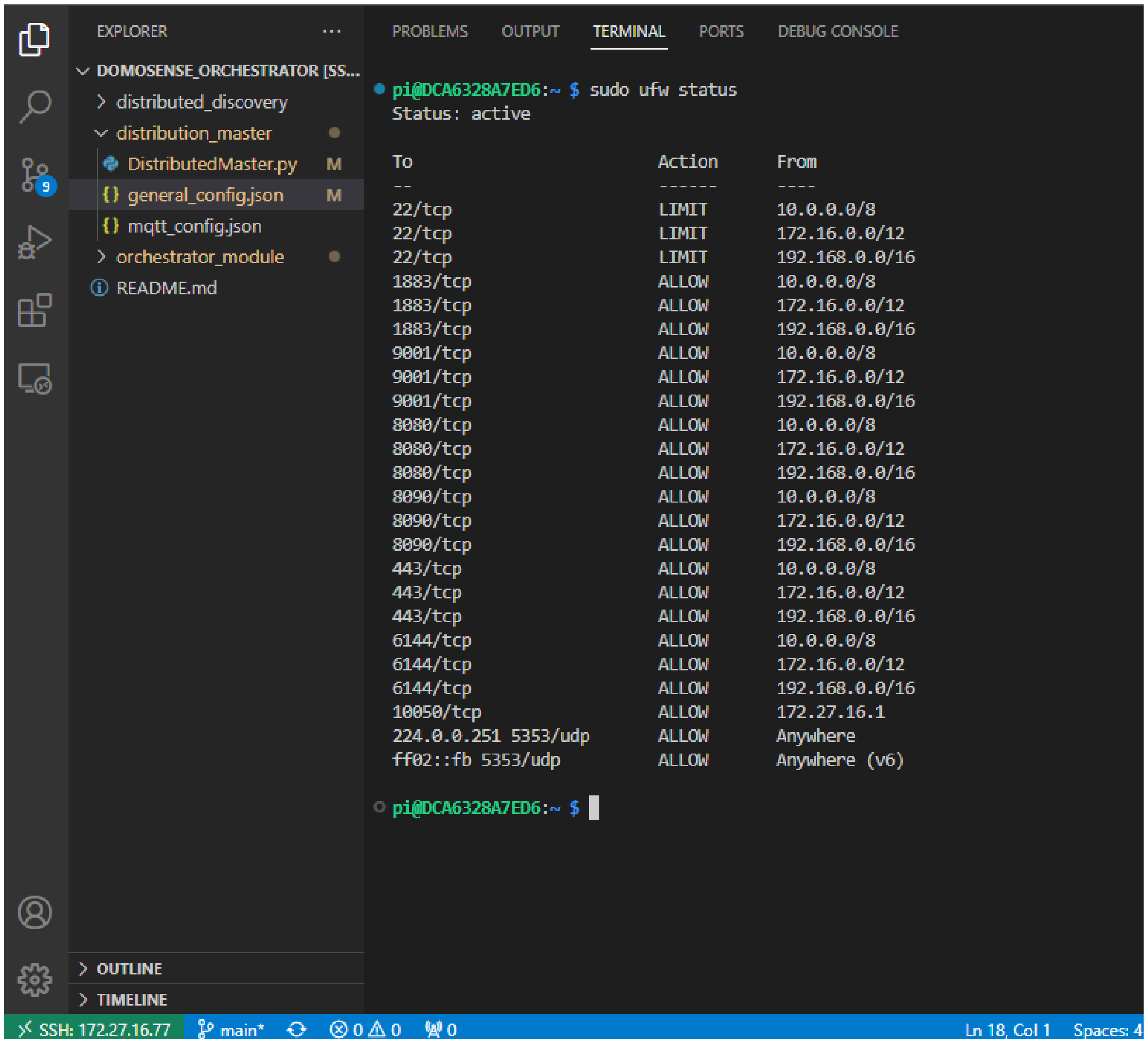
```
{
    "B827EB72B0E5": [
        "avahi-daemon.service",
        "juvo_service.service",
        "mqtt-service-registry.service",
        "openvpn.service",
        "battery_check.service",
        "connect.service",
        "cpu_check.service",
        "heart_beat.service",
        "intrusion.service",
        "physical_memory_check.service",
        "disk_usage.service",
        "serv.service"
    ]
}
```

Figure 35. List of running services of a fog node in the master node

3.3.4 Over the air update (OTA)

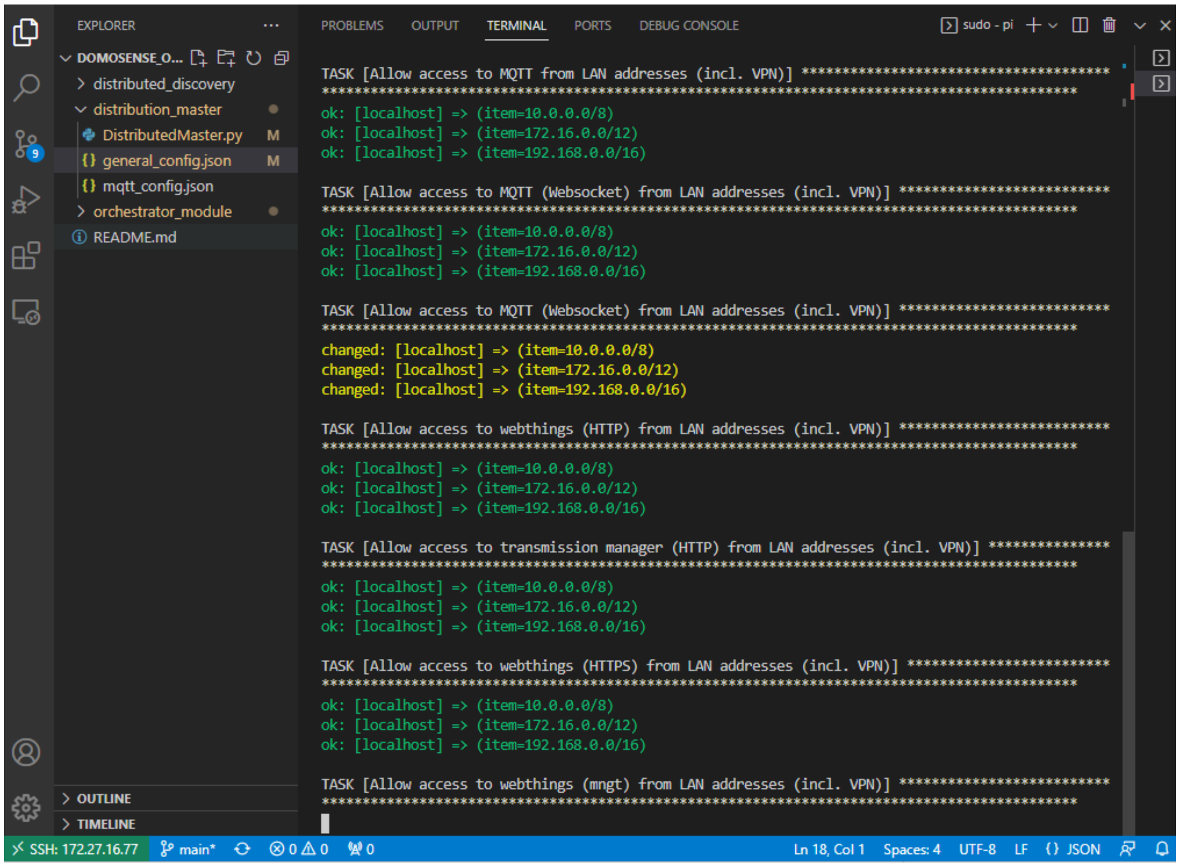
The goal of the test is to prove that any fog node has the ability to update resources or services in another node using our distributed model at any point in time. The QoS metrices satisfied from this feature is the **scalability** of the system to support an over-the-air update in any IoT system which can help maintain the **reliability**, **availability** of the system at all times.

Over the air update is a crucial feature that is a big part of all IoT systems in which we have updates being pushed from the cloud layer to our fog layer. We take that idea and validate it in our distribution model by enabling updates from the master node to other slave nodes. The service can be further extended to other IoT layers as the file transfer mechanism is independent from any specific domain. For our validation, we have used a firewall ansible script to open specific ports for communication. The initial script had a set of ports defined in it while the update script has the newer definition of another port. We monitor the ufw rules in the system before the update (Figure 36). We can observe the slave node executing the updated script with an updated ufw rules in which we enabled port 9002 for tcp communications (Figure 37). We again check the ufw status after the completion of the update and we see the updated rules for the port 9002 (Figure 38).



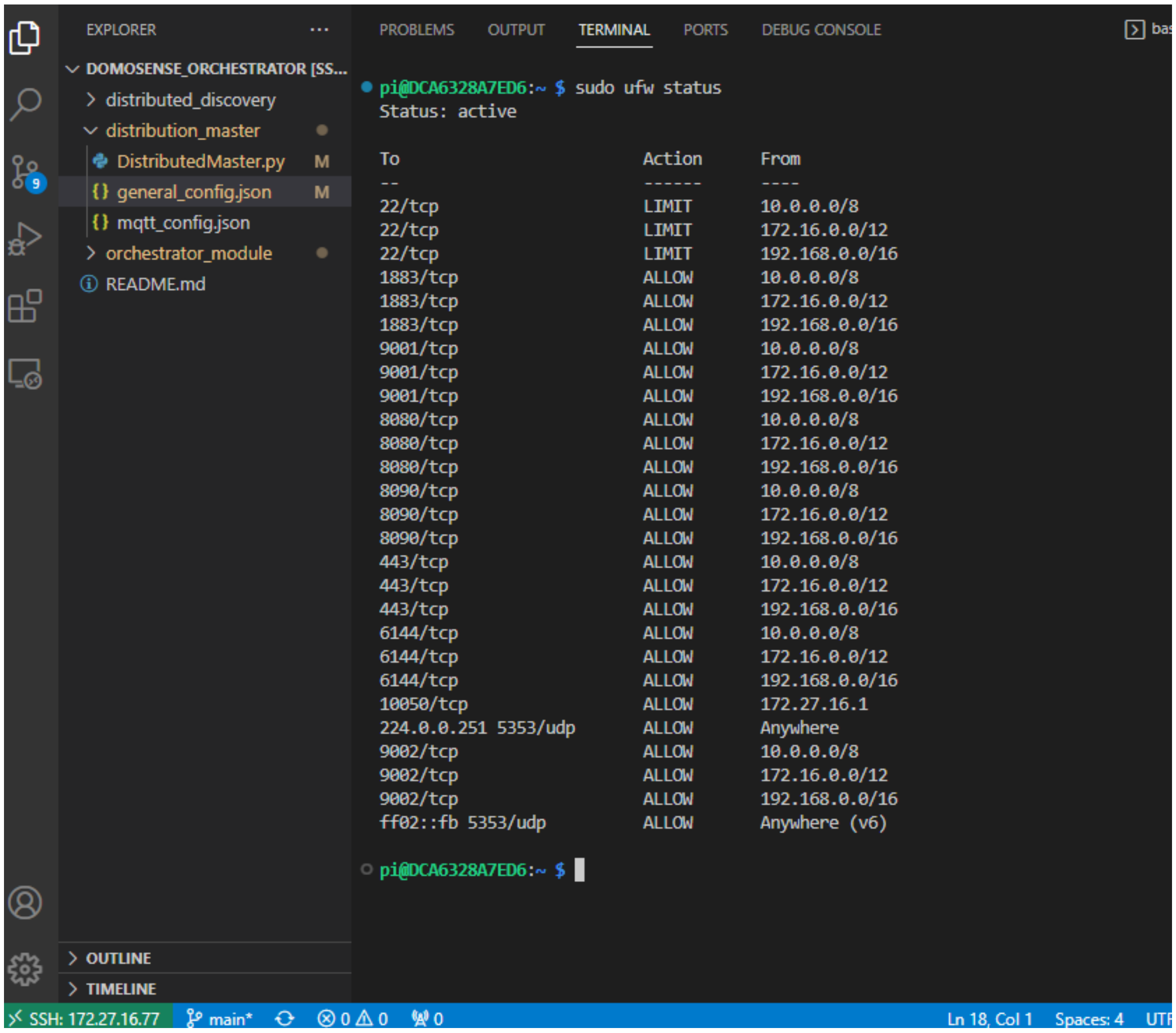
```
pi@DCA6328A7ED6:~ $ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
22/tcp                     LIMIT       10.0.0.0/8
22/tcp                     LIMIT       172.16.0.0/12
22/tcp                     LIMIT       192.168.0.0/16
1883/tcp                   ALLOW       10.0.0.0/8
1883/tcp                   ALLOW       172.16.0.0/12
1883/tcp                   ALLOW       192.168.0.0/16
9001/tcp                   ALLOW       10.0.0.0/8
9001/tcp                   ALLOW       172.16.0.0/12
9001/tcp                   ALLOW       192.168.0.0/16
8080/tcp                   ALLOW       10.0.0.0/8
8080/tcp                   ALLOW       172.16.0.0/12
8080/tcp                   ALLOW       192.168.0.0/16
8090/tcp                   ALLOW       10.0.0.0/8
8090/tcp                   ALLOW       172.16.0.0/12
8090/tcp                   ALLOW       192.168.0.0/16
443/tcp                    ALLOW       10.0.0.0/8
443/tcp                    ALLOW       172.16.0.0/12
443/tcp                    ALLOW       192.168.0.0/16
6144/tcp                   ALLOW       10.0.0.0/8
6144/tcp                   ALLOW       172.16.0.0/12
6144/tcp                   ALLOW       192.168.0.0/16
10050/tcp                  ALLOW       172.27.16.1
224.0.0.251 5353/udp       ALLOW       Anywhere
ff02::fb 5353/udp          ALLOW       Anywhere (v6)

pi@DCA6328A7ED6:~ $
```

Figure 36. UFW status before performing update



```
TASK [Allow access to MQTT from LAN addresses (incl. VPN)] ***************************************
**************************************************************************************************
ok: [localhost] => (item=10.0.0.0/8)
ok: [localhost] => (item=172.16.0.0/12)
ok: [localhost] => (item=192.168.0.0/16)

TASK [Allow access to MQTT (Websocket) from LAN addresses (incl. VPN)] **************************
**************************************************************************************************
ok: [localhost] => (item=10.0.0.0/8)
ok: [localhost] => (item=172.16.0.0/12)
ok: [localhost] => (item=192.168.0.0/16)

TASK [Allow access to MQTT (Websocket) from LAN addresses (incl. VPN)] **************************
**************************************************************************************************
changed: [localhost] => (item=10.0.0.0/8)
changed: [localhost] => (item=172.16.0.0/12)
changed: [localhost] => (item=192.168.0.0/16)

TASK [Allow access to webthings (HTTP) from LAN addresses (incl. VPN)] **************************
**************************************************************************************************
ok: [localhost] => (item=10.0.0.0/8)
ok: [localhost] => (item=172.16.0.0/12)
ok: [localhost] => (item=192.168.0.0/16)

TASK [Allow access to transmission manager (HTTP) from LAN addresses (incl. VPN)] ***************
**************************************************************************************************
ok: [localhost] => (item=10.0.0.0/8)
ok: [localhost] => (item=172.16.0.0/12)
ok: [localhost] => (item=192.168.0.0/16)

TASK [Allow access to webthings (HTTPS) from LAN addresses (incl. VPN)] *************************
**************************************************************************************************
ok: [localhost] => (item=10.0.0.0/8)
ok: [localhost] => (item=172.16.0.0/12)
ok: [localhost] => (item=192.168.0.0/16)

TASK [Allow access to webthings (mngt) from LAN addresses (incl. VPN)] **************************
**************************************************************************************************
```

Figure 37. Update of firewall rules in progress



```
pi@DCA6328A7ED6:~ $ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
22/tcp                     LIMIT       10.0.0.0/8
22/tcp                     LIMIT       172.16.0.0/12
22/tcp                     LIMIT       192.168.0.0/16
1883/tcp                   ALLOW       10.0.0.0/8
1883/tcp                   ALLOW       172.16.0.0/12
1883/tcp                   ALLOW       192.168.0.0/16
9001/tcp                   ALLOW       10.0.0.0/8
9001/tcp                   ALLOW       172.16.0.0/12
9001/tcp                   ALLOW       192.168.0.0/16
8080/tcp                   ALLOW       10.0.0.0/8
8080/tcp                   ALLOW       172.16.0.0/12
8080/tcp                   ALLOW       192.168.0.0/16
8090/tcp                   ALLOW       10.0.0.0/8
8090/tcp                   ALLOW       172.16.0.0/12
8090/tcp                   ALLOW       192.168.0.0/16
443/tcp                    ALLOW       10.0.0.0/8
443/tcp                    ALLOW       172.16.0.0/12
443/tcp                    ALLOW       192.168.0.0/16
6144/tcp                   ALLOW       10.0.0.0/8
6144/tcp                   ALLOW       172.16.0.0/12
6144/tcp                   ALLOW       192.168.0.0/16
10050/tcp                  ALLOW       172.27.16.1
224.0.0.251 5353/udp       ALLOW       Anywhere
9002/tcp                   ALLOW       10.0.0.0/8
9002/tcp                   ALLOW       172.16.0.0/12
9002/tcp                   ALLOW       192.168.0.0/16
ff02::fb 5353/udp          ALLOW       Anywhere (v6)

pi@DCA6328A7ED6:~ $
```

Figure 38. UFW status after performing update

The test was successfully executed as the master node was able to update the ufw security rules in a remote node.

3.3.5 Resource management

The goal of the test is to showcase the resource management ability of our distributed system hence providing better service functionality in a distributed environment compared to a single fog node implementation of the same service. We will take the example of the Bluetooth service in our test. We target the QoS metrices of scalability and interoperability through this test.

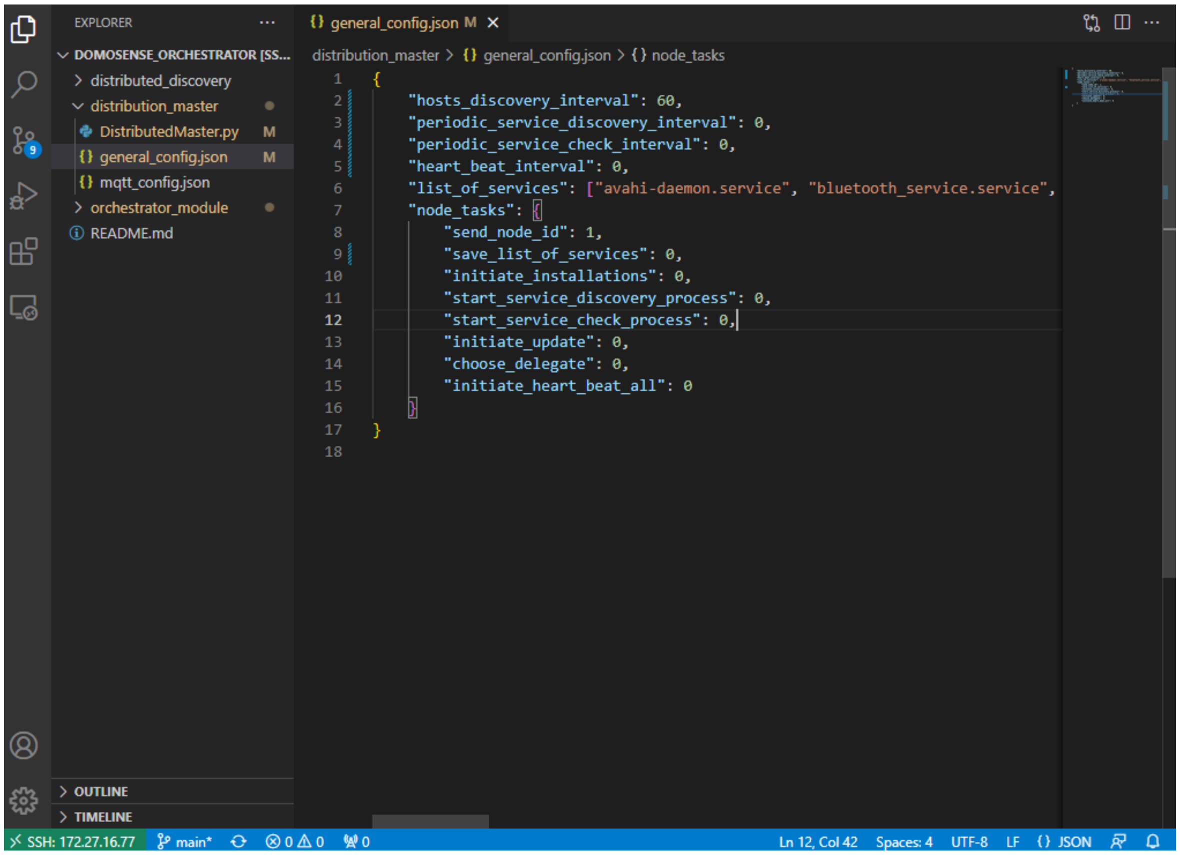
One of the experiments we conducted was to have inter service communications. Based on the challenges faced with Bluetooth in singular nodes, we tasked one to be the discovery node for all the nearby Bluetooth devices and another node just responsible for the acquisition of data from the discovered list of devices. The list of devices was easily transferred with our input model of the distribution tool and saved as a file in the other node for it to run acquisition. The service status for acquisition was stable throughout our experiment phase. The previous limit to our single node implementation was 2 devices and with this implementation it was raised to 6 to 7 devices after which the stability of the Bluetooth driver fails due to concurrent access of the driver at a high rate.

The distribution of the Bluetooth service in more than one node increased the overall stability of the service. Thus, the test was successful.

3.3.6 Delegation of master node

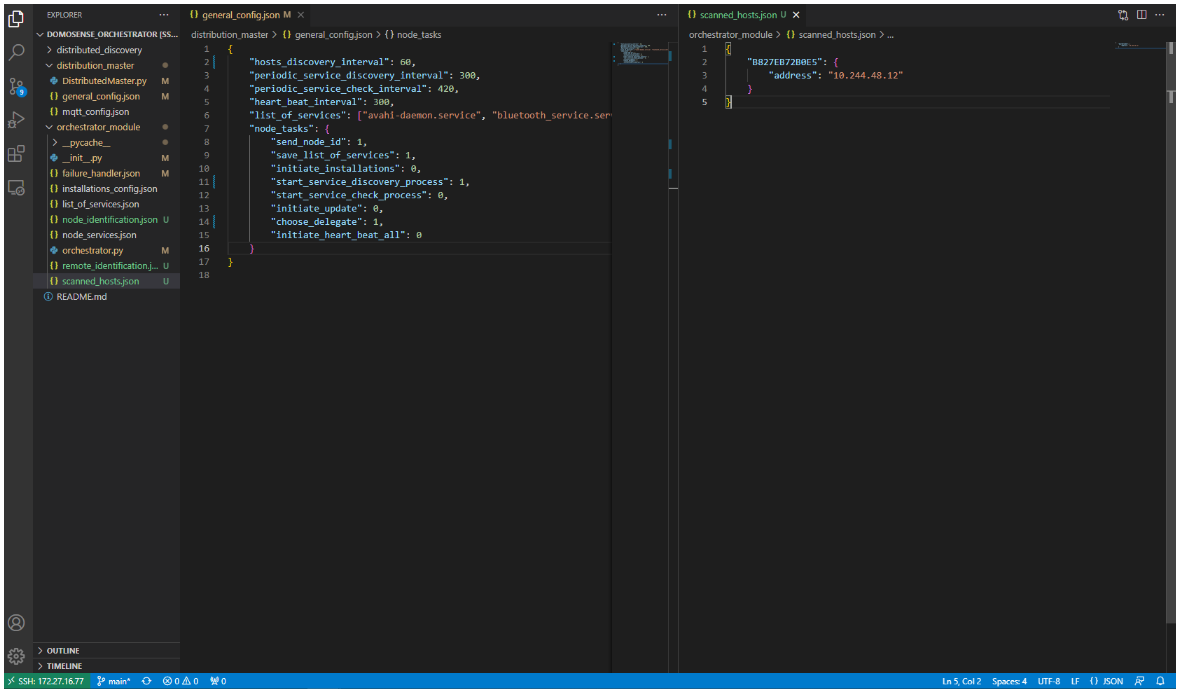
The goal of the test is to check the ability of the distributed system to handle the situations when the master node is offline. Through this test we are trying to prove the **offline capability** of the distributed system and in turn providing for the platform **availability** and **stability**.

One of the major challenges discussed in lessons learned section in Chapter 2 is the offline capability of the system. In case of a distributed system, the master node monitors the heartbeat of all slave nodes, however, the important question is what happens when the master node itself goes down. For this solution the master node selects one slave node as its delegate by saving a configuration file in that node. The initial configuration file of the slave node has different constant values and a different set of tasks activated(Figure 39).We then manually power down the master node while monitoring the slave node. After a specific period of time, when the scanned devices do not show the current master node as alive, we see the initial configuration file for that slave node changes to the master node configurations (Figure 40).



```
{
    "hosts_discovery_interval": 60,
    "periodic_service_discovery_interval": 0,
    "periodic_service_check_interval": 0,
    "heart_beat_interval": 0,
    "list_of_services": ["avahi-daemon.service", "bluetooth_service.service",
    "node_tasks": {
        "send_node_id": 1,
        "save_list_of_services": 0,
        "initiate_installations": 0,
        "start_service_discovery_process": 0,
        "start_service_check_process": 0,
        "initiate_update": 0,
        "choose_delegate": 0,
        "initiate_heart_beat_all": 0
    }
}
```

Figure 39. Initial configuration file of slave node



```
{
    "hosts_discovery_interval": 60,
    "periodic_service_discovery_interval": 300,
    "periodic_service_check_interval": 420,
    "heart_beat_interval": 300,
    "list_of_services": ["avahi-daemon.service", "bluetooth_service.ser
    "node_tasks": {
        "send_node_id": 1,
        "save_list_of_services": 1,
        "initiate_installations": 0,
        "start_service_discovery_process": 1,
        "start_service_check_process": 0,
        "initiate_update": 0,
        "choose_delegate": 1,
        "initiate_heart_beat_all": 0
    }
}
```

```
{
    "B827EB72B0E5": {
        "address": "10.244.48.12"
    }
}
```

Figure 40. Updated configuration file of slave node acting as master node.

The functionality of the master node was carried out by the delegated slave node after the master node went offline. Hence the test was successful.

3.4 Conclusion

In this chapter we discussed the tests and the validation of our QoS agents in a single fog node and we also validated our distribution model with a set of tests which showcased the possibilities of satisfying QoS metrices through the model. We have highlighted the targeted QoS metrices in all our tests. In the introduction, we mentioned our goal was to have a fog layer with QoS implementation and based on our tests we have provided solutions to the major drawbacks reviewed in our literature and real-life deployments. We have also provided the tools for future implementations and improvements and the ability to be integrated in any IoT platform.

Conclusion

The main target of the thesis is the concept of QoS in IoT, with focus on the fog layer. We target an approach to be generic enough for use beyond a specific IoT system. Our approach had two parts, one targeting the single fog node and the other providing a distributed fog layer. In order to create autonomous solutions, we built QoS agents in the form of dynamic modules for the discovered point of failures. With multiple QoS agents handling different aspects of the fog layer system, we were able to achieve an overall stable fog layer.

As for our approach for a singular fog node, we have brought forward a simple but dynamic aspect which is scalable to fit any number of QoS agents which can be reusable for targeting more than metric at a time. The main idea from the beginning was to have reusable structure for enforcing and maintain QoS metrices in IoT platforms. In this regard our approach is able to provide answers to all domains of failures that are known to us currently and even having the ability to incorporate unforeseen point of failures and situations in the future.

Similarly, the major issue faced in the single node fog layer is how to have a proper distribution system for fog layer without having to go to the cloud layer. Our distributed model is not restricted to fog layer only but can be used for distribution at any layer as there are no concepts tying the model to a particular layer of implementation. We validated our model with a set of use cases showcasing the ability of the model to cater to any number of QoS metrices. The dynamicity of the model will allow it for a significant growth in the future.

While this study provides a valuable insight into the need of QoS in IoT and the means to achieve that in the fog layer of the IoT platform, a number of limitations need to be acknowledged:

- The study focuses on a single layer of the IoT platform, while we proposed methods to stabilize one layer, we need to have QoS implementations in all other layers.
- The distributed model does not yet support mechanisms to have communication outside the internal network layer.
- Our distributed model does not include a machine learning model to learn from the knowledge base in our QoS agents.

With these limitations we would have a number of steps to take in the future to extend the work and make it more robust and dynamic.

In future works we should mainly focus on using the knowledge base created by our QoS agents to be able to classify new patterns of failures and possible solutions for such cases. The future work would definitely include establishing QoS in all the other layers using our approach in the fog layer. In particular, it is very essential to have a better implementation of QoS in cloud layer to enhance the stability and reliability of the system. With respect to the distribution model, the future work would include a distribution of local database and a QoS agent managing the entire data of the distributed system, this will help in resource management and provide for better offline capabilities.

Bibliography

[1] T. Kramp, R. Van Kranenburg, and S. Lange, “Introduction to the Internet of Things,” in *Enabling Things to Talk*, A. Bassi, M. Bauer, M. Fiedler, T. Kramp, R. Van Kranenburg, S. Lange, and S. Meissner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–10. doi: 10.1007/978-3-642-40403-0_1.

[2] M. Skilton and F. Hovsepian, “The 4th Industrial Revolution,” Springer, Springer Books, 2018.

[3] S. Nižetić, P. Šolić, D. López-de-Ipiña González-de-Artaza, and L. Patrono, “Internet of Things (IoT): Opportunities, issues and challenges towards a smart and sustainable future,” *Journal of Cleaner Production*, vol. 274, p. 122877, Nov. 2020, doi: 10.1016/j.jclepro.2020.122877.

[4] R. Duan, X. Chen, and T. Xing, “A QoS Architecture for IOT,” in *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, Dalian, China: IEEE, Oct. 2011, pp. 717–720. doi: 10.1109/iThings/CPSCom.2011.125.

[5] O. Akrivopoulos, N. Zhu, D. Amaxilatis, C. Tselios, A. Anagnostopoulos, and I. Chatzigiannakis, “A Fog Computing-Oriented, Highly Scalable IoT Framework for Monitoring Public Educational Buildings,” in *2018 IEEE International Conference on Communications (ICC)*, Kansas City, MO: IEEE, May 2018, pp. 1–6. doi: 10.1109/ICC.2018.8422489.

[6] M. Kavre, A. Gadekar, and Y. Gadhade, "Internet of Things (IoT): A Survey," in *2019 IEEE Pune Section International Conference (PuneCon)*, Pune, India: IEEE, Dec. 2019, pp. 1–6. doi: 10.1109/PuneCon46936.2019.9105831.

[7] Shancang Li, G. Oikonomou, T. Tryfonas, T. M. Chen, and Li Da Xu, "A Distributed Consensus Algorithm for Decision Making in Service-Oriented Internet of Things," *IEEE Trans. Ind. Inf.*, vol. 10, no. 2, pp. 1461–1468, May 2014, doi: 10.1109/TII.2014.2306331.

[8] Poorana Senthilkumar S., B. Subramani, "Study on IoT Architecture, Application Protocol and Energy needs", *International Journal of Scientific Research in Network Security and Communication*, Vol.8, Issue.5, pp.7-12, 2020.

[9] K. Sha, T. A. Yang, W. Wei, and S. Davari, "A survey of edge computing-based designs for IoT security," *Digital Communications and Networks*, vol. 6, no. 2, pp. 195–202, May 2020, doi: 10.1016/j.dcan.2019.08.006.

[10] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," Studies in Computational Intelligence, vol. 546, pp. 169–186, 2014, . doi: 10.1007/978-3-319-05029-4_7.

[11] K. Ma, A. Bagula, C. Nyirenda, and O. Ajayi, "An iot-based fog computing model," Sensors (Switzerland), vol. 19, no. 12, Jun. 2019, doi: 10.3390/s19122783.

[12] M. Lombardi, F. Pascale, and D. Santaniello, "Internet of things: A general overview between architectures, protocols and applications," *Information (Switzerland)*, vol. 12, no. 2, pp. 1–21, Feb. 2021, doi: 10.3390/info12020087.

[13] B. Abdulrazak, S. Paul, S. Maraoui, A. Rezaei, and T. Xiao, "IoT Architecture with Plug and Play for Fast Deployment and System Reliability: AMI Platform," in *International Conference On Smart Living and Public Health*, Springer Science and Business Media Deutschland GmbH, 2022, pp. 43–57. doi: 10.1007/978-3-031-09593-1_4.

[14] C. Cicconetti, M. Conti, A. Passarella, and D. Sabella, "Toward Distributed Computing Environments with Serverless Solutions in Edge Systems," *IEEE Communications Magazine*, vol. 58, no. 3, pp. 40–46, Mar. 2020, doi: 10.1109/MCOM.001.1900498.

[15] J. P. Miguel, D. Mauricio, and G. Rodríguez, "A Review of Software Quality Models for the Evaluation of Software Products," *International Journal of Software Engineering & Applications*, vol. 5, no. 6, pp. 31–53, Nov. 2014, doi: 10.5121/ijsea.2014.5603.

[16] L. Li, M. Rong, and G. Zhang, "An Internet of things QoS estimate approach based on multi-dimension QoS," in *Proceedings of the 9th International Conference on Computer Science and Education, ICCCSE 2014*, Institute of Electrical and Electronics Engineers Inc., Oct. 2014, pp. 998–1002. doi: 10.1109/ICCSE.2014.6926613.

[17] V. N. Talooki and J. Rodriguez, "Jitter based comparisons for routing protocols in mobile ad hoc networks," in *2009 International Conference on Ultra Modern Telecommunications and Workshops*, 2009. doi: 10.1109/ICUMT.2009.5345590.

[18] Guangxiang Yuan and Juan Pan, "Throughput efficiency of opportunistic network coding in wireless networks," in *IET International Conference on Information and Communications Technologies (IETICT 2013)*, Beijing, China: Institution of Engineering and Technology, 2013, pp. 129–134. doi: 10.1049/cp.2013.0045.

[19] A. Montazerolghaem and M. H. Yaghmaee, "Load-Balanced and QoS-Aware Software-Defined Internet of Things," *IEEE Internet Things J*, vol. 7, no. 4, pp. 3323–3337, Apr. 2020, doi: 10.1109/JIOT.2020.2967081.

[20] X. Zhang, J. Lv, X. Han, and D. K. Sung, "Channel efficiency-based transmission rate control for congestion avoidance in wireless ad hoc networks," *IEEE Communications Letters*, vol. 13, no. 9, pp. 706–708, 2009, doi: 10.1109/LCOMM.2009.090861.

[21] D. Singh, G. Tripathi, and A. J. Jara, "A survey of Internet-of-Things: Future vision, architecture, challenges and services," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, Korea (South): IEEE, Mar. 2014, pp. 287–292. doi: 10.1109/WF-IoT.2014.6803174.

[22] A. Alrawais, A. Alhothaily, C. Hu, and X. Cheng, "Fog Computing for the Internet of Things: Security and Privacy Issues," *IEEE Internet Comput.*, vol. 21, no. 2, pp. 34–42, Mar. 2017, doi: 10.1109/MIC.2017.37.

[23] Paulo Teixeira de Sousa and Peter Stuckmann, "Telecommunication Systems and Technologies", Oxford: Eolss; 2009. Oxford: Eolss; 2009, vol. Vol. II.

[24] Kang Lee, "Sensor standards harmonization-path to achiewving sensor interoperability," in *2007 IEEE Autotestcon*, Baltimore, MD, USA: IEEE, Sep. 2007, pp. 381–388. doi: 10.1109/AUTEST.2007.4374244.

[25] B.-Y. Lee and G.-H. Lee, "Service Oriented Architecture for SLA Management System," in *The 9th International Conference on Advanced Communication Technology*, Gangwon-Do, Korea: IEEE, Feb. 2007, pp. 1415–1418. doi: 10.1109/ICACT.2007.358621.

[26] J. Zhou and Y. F. Guo, "Guaranteeing maximum reliability and minimum delay QoS routing based on WF2Q," in *CIS 2009 - 2009 International Conference on Computational Intelligence and Security*, 2009, pp. 545–549. doi: 10.1109/CIS.2009.222.

[27] A. Brogi and S. Forti, "QoS-aware deployment of IoT applications through the fog," *IEEE Internet Things J*, vol. 4, no. 5, pp. 1185–1192, 2017, doi: 10.1109/JIOT.2017.2701408.

[28] F. Samie, V. Tsoutsouras, S. Xydis, L. Bauer, D. Soudris, and J. Henkel, "Distributed QoS management for Internet of Things under resource constraints," in *2016 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2016*, Institute of Electrical and Electronics Engineers Inc., Nov. 2016. doi: 10.1145/2968456.2974005.

[29] A. Rawat, V. Daza, and M. Signorini, "Offline Scaling of IoT Devices in IOTA Blockchain," *Sensors*, vol. 22, no. 4, Feb. 2022, doi: 10.3390/s22041411.

[30] L. Li, M. Rong, and G. Zhang, "An Internet of things QoS estimate approach based on multi-dimension QoS," in *Proceedings of the 9th International Conference on Computer Science and Education, ICCCSE 2014*, Institute of Electrical and Electronics Engineers Inc., Oct. 2014, pp. 998–1002. doi: 10.1109/ICCSE.2014.6926613.

[31] A. Dvornikov, P. Abramov, S. Efremov, and L. Voskov, "QoS Metrics Measurement in Long Range IoT Networks," in *Proceedings - 2017 IEEE 19th Conference on Business Informatics, CBI 2017*, Institute of Electrical and Electronics Engineers Inc., Aug. 2017, pp. 15–20. doi: 10.1109/CBI.2017.2.

[32] P. K. Donta, S. N. Srirama, T. Amgoth, and C. S. R. Annavarapu, “Survey on recent advances in IoT application layer protocols and machine learning scope for research directions,” *Digital Communications and Networks*, vol. 8, no. 5. KeAi Communications Co., pp. 727–744, Oct. 01, 2022. doi: 10.1016/j.dcan.2021.10.004.

[33] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, “A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications,” *IEEE Internet Things J*, vol. 4, no. 5, pp. 1125–1142, Oct. 2017, doi: 10.1109/JIOT.2017.2683200.

[34] R. Duan, X. Chen, and T. Xing, “A QoS architecture for IOT,” in *Proceedings - 2011 IEEE International Conferences on Internet of Things and Cyber, Physical and Social Computing, iThings/CPSCom 2011*, 2011, pp. 717–720. doi: 10.1109/iThings/CPSCom.2011.125.

[35] M. A. Abd-Elmagid, N. Pappas, and H. S. Dhillon, “On the Role of Age of Information in the Internet of Things,” *IEEE Communications Magazine*, vol. 57, no. 12, pp. 72–77, Dec. 2019, doi: 10.1109/MCOM.001.1900041.

[36] I. Udoh and G. Kotonya, “A Dynamic QoS Negotiation Framework for IoT Services,” in *2019 IEEE Global Conference on Internet of Things (GCIoT)*, Dubai, United Arab Emirates: IEEE, Dec. 2019, pp. 1–7. doi: 10.1109/GCIoT47977.2019.9058418.

[37] R. Aydoğan, D. Festen, K. V. Hindriks, and C. M. Jonker, “Alternating Offers Protocols for Multilateral Negotiation,” in *Modern Approaches to Agent-based Complex Automated Negotiation*, K. Fujita, Q. Bai, T. Ito, M. Zhang, F. Ren, R. Aydoğan, and R. Hadfi, Eds., in Studies in Computational Intelligence, vol. 674. Cham: Springer International Publishing, 2017, pp. 153–167. doi: 10.1007/978-3-319-51563-2_10.

[38] M. C. Hout, M. H. Papesh, and S. D. Goldinger, “Multidimensional scaling,” *Wiley Interdiscip Rev Cogn Sci*, vol. 4, no. 1, pp. 93–103, Jan. 2013, doi: 10.1002/wcs.1203..

[39] L. Li, S. Li, and S. Zhao, "QoS-Aware scheduling of services-oriented internet of things," *IEEE Trans Industr Inform*, vol. 10, no. 2, pp. 1497–1507, 2014, doi: 10.1109/TII.2014.2306782.

[40] Y. Song, S. S. Yau, R. Yu, X. Zhang, and G. Xue, "An Approach to QoS-based Task Distribution in Edge Computing Networks for IoT Applications," in *Proceedings - 2017 IEEE 1st International Conference on Edge Computing, EDGE 2017*, Institute of Electrical and Electronics Engineers Inc., Sep. 2017, pp. 32–39. doi: 10.1109/IEEE.EDGE.2017.50.

[41] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet Things J*, vol. 3, no. 5, pp. 637–646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[42] R. Saha, S. Misra, and P. K. Deb, "FogFL: Fog-Assisted Federated Learning for Resource-Constrained IoT Devices," *IEEE Internet Things J*, vol. 8, no. 10, pp. 8456–8463, May 2021, doi: 10.1109/JIOT.2020.3046509.

[43] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003, doi: 10.1109/MC.2003.1160055.

[44] F. Dahlqvist, M. Patel, A. Rajko, and J. Shulman, "Growing opportunities in the Internet of Things," *McKinsey & Company*, pp. 1–6, 2019.

[45] B. Abdulrazak, J. A. Codjo, and S. Paul, "Self-healing Approach for IoT Architecture: AMI Platform," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Science and Business Media Deutschland GmbH, 2022, pp. 3–17. doi: 10.1007/978-3-031-09593-1_1.

[46] K. Wang, "KubeEdge: Bring Cloud Native Into Edge Computing," Tata Institute Of Fundamental Research (*TFiR)*, Apr. 12, 2023.

[47] C. Carrión, "Kubernetes as a Standard Container Orchestrator - A Bibliometric Analysis," *J Grid Computing*, vol. 20, no. 4, p. 42, Dec. 2022, doi: 10.1007/s10723-022-09629-8.

[48] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with KubeEdge," in *Proceedings - 2018 3rd ACM/IEEE Symposium on Edge Computing, SEC 2018*, Institute

of Electrical and Electronics Engineers Inc., Dec. 2018, pp. 373–377. doi: 10.1109/SEC.2018.00048.

[49] G. A. Qadir and S. R. M. Zeebaree, "Evaluation of QoS in Distributed Systems: A Review", doi: 10.5281/ZENODO.4462245.

Appendix

Implementation Design

We describe in detail in this section our implementation of the QoS agent for data recovery as discussed in chapter 2.4.3. This agent is crucial for having a local database management system in the fog layers.

QoS agent for data recovery

The data recovery agent is built with the idea of storing all data from the edge devices on the fog nodes and then synching it with the cloud. The agent uses SQLite database as its storage component. The data recovery agent gets the data from the services in the fog layer which fetches data from the edge devices, and it looks for an instance of the SQLite database with that particular service name. If the instance does not exist it creates the instance with the instance name as the service name, for example, "{service_name}.db" is the instance name for a pre-processing service called "{service_name}". After the database has been created, the next step is to create the internal tables. The initial approach that we used was to create one table for each service. Although it served the purpose of storing data locally, we had several challenges in regard to management of data and data synching with the cloud. The issue with having a single table was that the size of the database table grew very fast and so parsing the table became extensively difficult. Due to this challenge we decided to create tables based on defined intervals.

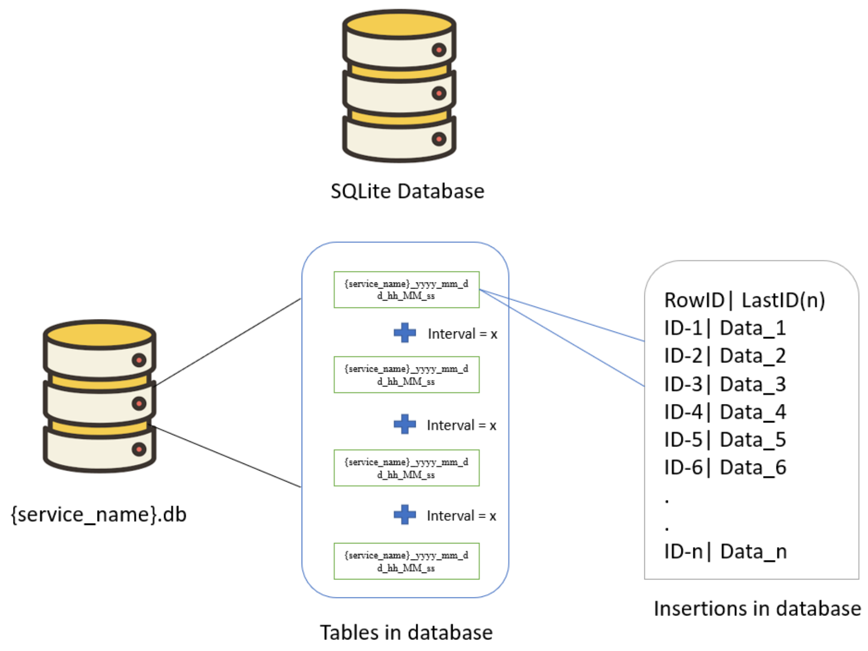


Figure 41. Internal Structure of the database.

For example, if the interval is set as four, we would create a new table for a particular service each four hours. In this way, we can have smaller tables to manage while maintaining the structure of the data in our database Figure 48. These tables are then used to synchronize the data with the cloud. To perform the data storage and synchronization tasks the data recovery agent uses two components: the database library and the local storage module. We detail the functionalities of these two components in the following parts.

a) Data Recovery: Database Library

The database library is another internal component of the data recovery agent which is used to provide the micro agents the functionalities to use the database services with ease. This library supports the basic database services like insertion, deletion, creation of tables, updating tables and managing the database transactions. This library is built with the idea of dynamicity, so any new functionalities can be added with the least possible complexity and can be used by all services in the system. This library is used by the micro agents and other fog services. Next, we will discuss how the agent handles the synchronization task with the help of the local storage module.

b) Data Recovery: Local Storage Module

The local storage module is used to perform the database synchronization with the cloud database. The module uses a defined interval as an input to perform synchronization. The first step of this module is to check the number of database instances we have in our node, it then gets the timestamps of the first entry and the last entry of each table in an instance and sends them to the cloud nodes. The cloud nodes send a list of all the ids in that timeframe and the local storage module then deletes all the received ids from that table and if there are still some ids left in the tables it suggests that the data corresponding to those ids was not sent to the cloud. These data are sent to the cloud thus synching the local database with the cloud database. Upon sending these data we still keep them and do not delete them at this point as it might have not been transmitted properly due to some network issues. The idea is to wait for the next iteration and delete only those data from the local database that we have received in the cloud. This iteration is done on each table in all the instances of the database at a definite interval period.

One more feature that is added in this module is the database rotation feature. One major issue that was observed with this implementation was that by default the fog nodes have limited memory resources and if the connection between the fog node and the cloud nodes is not established for a significant amount of time then it causes the fog nodes to crash due to over saturation of memory resources. Keeping this issue in mind, we came up with the idea of database rotation in which we keep on deleting data from a table in one instance of the database at a particular time. We assume a threshold for the total database memory usage and on reaching that threshold this functionality is triggered where in we rotate the database to free up memory space.

This process results in permanent data loss but it's one of the most optimized ways to process such resource constraint trade-offs. It can be further optimized in a distributed architecture of fog layer.