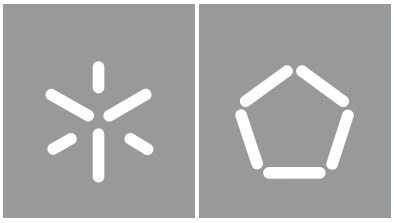
Universidade do Minho
Escola de Engenharia

Bruno Manuel Chaves Martins

Cloud-based IoT as a Service

dezembro de 2021

Cloud-based IoT as a Service
Bruno Manuel Chaves Martins
UMinho | 2021



Universidade do Minho
Escola de Engenharia

Bruno Manuel Chaves Martins

Cloud-based IoT as a Service

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho efetuado sob a orientação do(a)
Paulo Jorge Freitas de Oliveira Novais
Bruno Filipe Martins Fernandes

dezembro de 2021

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros

Acknowledgments

In the first place, I would like to thank my supervisor, professor Paulo Novais for accepting me and the suggested topic and, for guiding me during its development providing every tool I needed to perform great work. The amount of technical knowledge I acquired during this time was huge, and for that, I'm forever grateful.

I want to thank my family for always supporting me in every aspect of the walk I began all these years ago. Their motivation and belief in me were critical for my success. Without them, this major achievement in my life wouldn't be possible.

I also would like to thank professor Bruno Fernandes, who performed a key part by helping me every step of the way and by always being available to steer me in the right direction when I needed it. His wisdom and words of encouragement were crucial to making me perform at my best.

To my closest friends, who were always there when I needed it, cheering me on this journey and helping me get through some stressful moments.

Bruno Martins

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

IoT como um serviço baseado na Cloud

Internet of Things (IoT), está-se a tornar cada vez mais parte das nossas vidas e, quando aliada a computação na *Cloud*, torna-se uma ferramenta muito poderosa devido a remover stress computacional de pequenas placas ineficientes. Software como um serviço já representa uma grande parte do nosso dia-a-dia com empresas como a Netflix a aplicar o conceito de forma muito bem-sucedida. No contexto de IoT, este conceito ainda não está globalmente disseminado.

A natureza heterogénea dos dispositivos representa um grande desafio para os conseguir integrar num sistema na *Cloud*. Os diferentes formatos e tipos de dados enviados para um *middleware* são difíceis de processar e, como consequência, leva a que exista uma grande pressão no programador para que todos os sensores sejam suportados.

Ao longo deste estudo são exploradas variadas arquiteturas de forma a ser possível desenhar um sistema eficiente e, como os diferentes protocolos de comunicação afetam a rede em termos de *overhead* e fiabilidade. O sistema concebido, baseado em toda o estudo realizado, consiste numa aplicação para salas inteligentes que infere quantas pessoas estão lá dentro através de *Probes* de *WiFi*, disponibiliza essa informação a utilizadores e é verificada a possbilidade de utilização de algoritmos de *Machine Learning* como forma de optimizar resultados. O sistema desenhado permite aos programadores adicionar outros dispositivos sem ter de se preocupar como as mensagens são recebidas, apenas necessitando de adicionar a lógica que extrai o conhecimento dos dados. No que toca à área de *crowdsensing* deste trabalho, a precisão do sistema foi melhorada quando comparando com outros algoritmos estudados.

Palavras-chave: Processamento na Nuvem, Internet das Coisas, Aprendizagem automática, Software como um Serviço, Sondagem de Wifi

Abstract

Cloud-based IoT as a Service

Internet of Things, or IoT, is becoming more and more a part of our lives and when allied with cloud computing it becomes a very powerful tool by removing the computing stress from the small energy-efficient boards. Software as a Service is already a major part of our day-to-day lives with companies like Netflix successfully applying this concept. In IoT this type of concept is not widely applied.

The heterogeneous nature of devices poses a big challenge to integrate them in a cloud system, the different data and formats sent to a middleware are hard to process and puts pressure on the developer to ensure all sensors are supported.

Throughout this study we explore and design different types of architectures for efficient applications and how the different communication protocols affect the network when it comes to overhead and reliability. The conceived system, theoretically grounded on the research work, consists of a smart room application that senses how many people are inside a space through the process of *WiFi* Probing makes that information available to a user and makes use Machine Learning algorithms as a way to improve results. The design system allows developers to easily add new types of devices to the network without needing to worry how the messages are received, only needing to add domain logic to extract knowledge from the data. Regarding the crowdsensing aspect of this work, the accuracy of the system was improved when compared to other algorithm.

Keywords: Cloud, Internet of Things, Machine Learning, Software as a Service, *WiFi* Probing.

Contents

List of Acronyms

AMQP Advanced Message Queuing Protocol. 17, 44, 46, 47

ANN Artificial Neural Network. 22

AP Access Point. 20, 28

API Application Programming Interface. 12, 43, 46, 58

BLE Bluetooth Low Energy. 21

CoAP Constrained Application Protocol. 14, 16, 17

HTTP Hypertext Transfer Protocol. 13, 16

IoT Internet of Things. iv, v, 1–3, 5, 7, 10–16, 18, 22, 25, 26, 29, 32, 33, 44, 57–59

JSON JavaScript Object Notation. xiv, 13, 28, 32, 36, 40

MAC Media Access Control. 3, 20, 27, 38

ML Machine Learning. 2, 3, 5, 7, 12, 22, 33, 43, 48, 53, 55–57, 59

MQTT Message Queuing Telemetry Transport. xi, xiii, xiv, 14–18, 26, 31, 44–46

REST RESTful. 40, 58

RSSI Received Signal Strength Indication. xii, 3, 20, 21, 27, 48–51, 53

SaaS Software as a Service. 1, 2, 32, 58

SOA Service Oriented Architecture. 9, 12, 29, 30, 37

TCP Transmission Control Protocol. 17

UDP User Datagram Protocol. 16, 17

WSN Wireless Sensor Network. 1

XML Extensible Markup Language. 13

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In this chapter, it will be addressed the context and motivation of this work, as well as the project's objectives and research methodology. Finally, the paper structure will be described.

1.1 Context and Motivation

Internet of Things or IoT, is the expression that describes the interconnectivity of objects and sensors to the internet to create a grid of devices that collect, share, and, in some cases, actuate in the environment to provide a better surrounding (Asir et al., 2016). These types of technologies are already deeply ingrained in our society, from smart home appliances to sensors spread all around cities to track many society metrics (Fernandes et al., 2020a). For example, Yuan et al. (2011) studied crowd density and distribution basing their research in Wireless Sensor Network (WSN) affirming that this kind of research is critical in human safety monitoring, traffic awareness, smart tours in museums and in a variety of others applications. The growth of the use of these technologies is exposing many challenges still open in the realm of what types of technologies to utilize, what is the best architecture to apply in each scenario (Naveen, 2016), and the repercussions of deployment, and on non-functional requirements.

Software as a Service (SaaS), is a somewhat old concept but widely spread nowadays. It consists of a company producing its own services and software to provide to a customer for a fee. These end-to-end solutions usually are provided via the web to the client and nowadays represent a large part of the market cap in IoT. It is projected that IoT as a service will grow to the extent as software as a service has become (Asir et al., 2016). *Cloud* computing is a major pillar of SaaS strategies because only with a robust infrastructure, a company can provide a service with the level of quality that is needed nowadays.

The integration of IoT and the *Cloud* is a challenge in itself, although the *Cloud* allows an elastic runtime infrastructure, self-service, and offers fine-grained IoT resources when systems scale, a whole new array of problems need to be addressed such as configuring new resources when they need to be added and how to scale down in off-peak times (Asir et al., 2016). IoT and *Cloud* are very different types of technologies in a sense that IoT is characterized by many distributed devices with limited computing abilities and storage and *Cloud* computing is described as a network with close to unlimited storage capabilities and processing

capabilities (Atlam et al., 2017).

1.2 Goals

The main objectives of this thesis are:

- Evaluate the different kinds of approaches to IoT and SaaS based on the *Cloud* in various contexts;
- Implement an end-to-end *Cloud*-based IoT as a Service system utilizing sensors that are as minimally intrusive as possible;
- Understand how to integrate a heterogeneous physical layer with a *Cloud*-based system with the proper protocols ensuring that adding different types of sensors when needed is a seamless action;
- Compare and choose the correct types of communication patterns and protocols that can be applied in each scenario;
- Understand Machine Learning (ML) algorithms in an IoT environment that produces data in real-time.

The case study of this work will be in the realm of smart rooms. It is a mostly unexplored side of informing people on what is the state of the room regarding the occupancy levels. The spaces where this system might be implemented consist, for example, of coworking spaces, library study spaces, or offices.

1.3 Research Hypothesis

This work aims to validate that IoT and the *Cloud* are two concepts that can easily be integrated, more specifically in the context of smart rooms approaching the problem from an architectural point of view. On the other hand, the results of this study, in the realm of smart rooms, can improve the quality of life of people by informing them about a number of variables that can be sensed. Lastly, the usage of sensors as minimally intrusive as possible is also a big key takeaway of this thesis.

1.4 The Case Study

This thesis case study is to develop and end to end solution that can infer how many people are inside a room in every moment using non invasive sensors and create a solution to inform other people or systems of that state.

1.4.1 Problems

The problem that this is attempting to solve falls into two categories. The major problem is the integration of heterogeneous IoT devices, each one with its characteristics, with the *Cloud* to provide a service. The secondary problem is related to crowdsensing, the lack of solutions that can easily infer how many people are in an indoor space shows space for investigation and improvement.

IoT integration with the *Cloud* has a lot of unanswered questions, and even the ones that have a solution are not always widely accepted as the best. The middleware layer that aggregates the data from different sensors shows the most space for improvement. Currently, there are no standardized norms to construct a middleware for IoT systems. The choice of communication protocol is one of the few problems that have a standardized set of solutions to select. To create a service around IoT is needed a solid architecture that can accommodate certain requirements such as elasticity and scalability. Depending on the problem at hand the answer might be very different. When creating an IoT service is expected a maximization of use with the data gathered. To do so, the creation of a ML algorithm is a way of doing it. This brings out a new problem from an architectural point of view.

The second part of this works problem is crowdsensing. Various types of approaches were analysed, with different sensors, accuracy, and even costs. The developed solution regarding the case study will try to take into account every problem to have a system that preserves the privacy of people, leverages technology to be cost-effective when gathering the data, and other key aspects.

1.4.2 Challenges

In the *Cloud* and IoT side of things, the choice and design of architecture is the major challenge. This decision is one of the most complex choices because it creates constraints that can impact future development. To roll back an architecture after development has already begun is very costly and might put the whole work in danger. In the communication side of things, there are standardized options to follow when deciding on what protocol to utilize but there is still the need to analyze the if deployment environment can handle the constraints imposed by some of them in terms of memory usage and, network latency and bandwidth.

From a crowdsensing point of view, the creation of a system that infers the presence of people indoors is an intricate process. Because this work aims to be non-intrusive, the use of *WiFi* probing is one of, if not, the only option to do so. When sensing through *WiFi* is very difficult to set an RSSI threshold to decide if a person is inside or left the room but is still near. Nowadays, with Media Access Control (MAC) randomization implementation in devices such as smartphones becomes more difficult to conclude if a certain MAC belongs to one person.

The implementation of ML is always a challenge no matter the field. The choice of an algorithm that suits the problem must be backed by research to avoid any future surprises that may lead to poor results. Like in the architecture choice, a late major change in this component is very costly and difficult.

1.4.3 Requirements

Being outlined the case study and, the problems and challenges attached to it the following requirements must be implemented in the proof of concept:

- Detect the presence of devices and associate them to people in an non-intrusive way;
- Sensors send data through the internet not relying on local networks;
- Sensor abstraction, only the data sent must obey to some parameters;
- Management and authentication of sensors;
- Management of different rooms in which the sensors are deployed;
- Offer historic data;
- Develop a web app to show real time data and previous history.

1.5 Research Methodology

For the platform execution, is used an interactive and incremental development approach. This means that each development cycle, or iteration, includes a new functionality but, previously implemented features can be modified to fix undetected problems in previous cycles (Vijayalakshmi, 2011). In Figure 1 is a detailed diagram of the iterative and incremental development model.

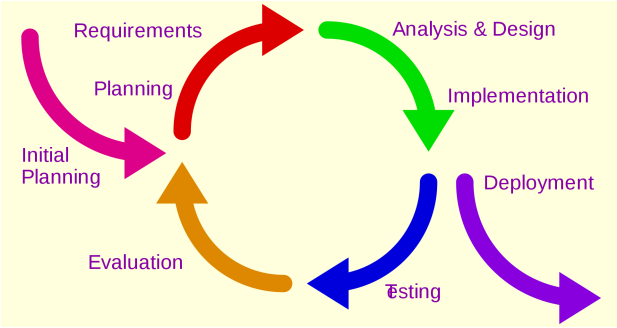


Figure 1: Iterative and incremental development model. Obtained from Ibrahim (2020).

The first task to achieve the platform goal is deciding a general architecture for the system evaluating the different possibilities to make an adequate choice.

While designing the system's architecture the approach to the development will be bottom-up, concentrating efforts on the sensing of the room and only moving to the next component after a solid base is established. The development of the physical layer can be done along side the architecture because the embedded code in sensors is not subject to architectural constraints in the early stages.

The next component will be a middleware to allow integration with the *Cloud*. This layer of the system, according to all research performed until this point, will be critical to the success of the system.

Afterward, the focus will be shifted to the ML model to ensure all knowledge gathered from the sensed data is utilized to provide the users of the system a better experience. The conception of these types of models have a development cycle of their. This project will develop all ML components following the *Cross-Industry Standard Process for Data Mining (CRISP-DM)* to achieve the best results possible. The different cycles and phases of this process are shown in Figure 2.

Figure 2: CRISP-DM cycle. Obtained from Chapman et al. (2000).

Lastly, to be able to provide a service, a business component will be developed as well as a website to be accessed by users to check the state of rooms where the system is deployed.

1.6 Document Structure

The current chapter (number one) makes an introduction to this work. Also explains, from different perspectives, the problem, and its challenges in order to find possible problems that may surface in the future as well as the soon to be developed work in this dissertation and the methodologies that will be followed. The rest of the document is composed of four more chapters.

Chapter number two reviews the state of the art and other relevant research to this thesis. Beginning with an overview of IoT and *Cloud* architectures to achieve integration between the two. After it's analyzed crowdsensing research with focus on *WiFi* techniques, and, at the end of the chapter is made an investigation on the ML topic.

Chapter number three describes the developed work as well as a detailed insight on most decisions take throughout, regarding technologies architectural patterns and algorithms.

Chapter number four presents the results of the previous chapter and some improvements made to optimize the final result.

Chapter number five presents the conclusions taken from this work as well as considerations for a future work.

Chapter 2

State of the art

This chapter addresses the current state of the art concerning the relevant topics of research related to this work, more specifically how crowdsensing can be performed in smart environments, IoT integration with the cloud and also ML techniques that are applied in an IoT context.

2.1 IoT integration with the Cloud

In the realm of IoT architectures, there are two that stand out, Cloud and Fog. A Cloud-based architecture offers flexibility and scalability as well as infrastructure, platforms, software, and storage, the Cloud also offers the possibility of data-mining, ML, and data visualization services (Sethi and Sarangi, 2017). On the other hand, in a fog architecture approach, sensors perform work that is otherwise done by servers, such as, processing data and analyzing it (Asir et al., 2016). As this work has in its goals to remove stress from sensors and move it to more powerful machines, from now on, only the Cloud will be considered.

2.1.1 Architectures

From an architectural point of view, the majority of systems are based on a monolithic architecture according to Lai et al. (2019), currently, this is not the most efficient way of handling the large amount of data gathered by sensors because a monolithic approach is not easily scalable or deployed. This type of system has its advantages, not dealing with a massively distributed system of various services is one of them. However, microservices architectures are in rapid growth due to some advantages. Lai et al. (2019) states that microservices, due to their distributed environment are a more resilient solution, given that if the system is well-engineered a failure of a component doesn't mean the failure of the system as a whole. Scalability, a topic mentioned earlier normally, is less of a hassle because we are dealing with smaller components often in a container (Pahl et al., 2020).

Al-Debagy and Martinek (2019) performed a comparative analysis between a monolithic and microservices approach. At the beginning of this work is stated that microservices can rely on different technologies to achieve a set of desired goals, as mentioned earlier if one part of the system fails it doesn't affect the

rest of the system, scaling is more accessible in microservices when compared to monolithic applications, deployment is easier allowing each service to be deployed independently and lastly, it helps companies aligning its architecture with their organizational structure reducing human costs. Furthermore, this work executed three types of scenarios to compare to each other. *JHipster* was chosen to produce web applications used in these tests. To determine performance differences was utilized *JMeter*. The metrics used for this comparison were response time and throughput. Response time is the time difference between a request and its reply. Throughput is the number of requests that can be satisfied per second. The first scenario was load testing, it started with 100 threads increased gradually every 2 minutes until 7000 threads were reached. In the second scenario was performed concurrency testing in which every service was used at the same time starting with sending 100 requests to each service increasing this value gradually until 1000 requests for each service were met. Finally, in the third scenario was tested the endurance of the systems starting with 1000 threads increased gradually every 10 min, in this final test it was analyzed two different service discovery technologies *Consul* and *Eureka*. After this experiment the following results were recorded:

- **Load Testing**: The results showed similar performance between microservices and monolithic architectures. Regarding throughput the monolithic application started better when the number of threads was 100, that is due to the communication between services in microservices but when the number of requests was increased the responses per second became even. In the later stages, the performance deteriorated steadily and on average, the difference between the two architecture was a marginal 0.87%. Response time was another measured metric, there was not a big difference, the response time increased linearly as the requests grew. The final metric was the number of fulfilled requests per thread, the monolithic architecture started with the slightest advantage in a small number of threads environment, but when the number of threads was increased both architectures started sending fewer responses, in the end, microservices were able to respond to more threads in comparison to the monolithic architecture;

- **Concurrency Testing**: In this test, regarding throughput, the monolithic architecture performed better than the microservices one, it showed a 6% better performance on average, this result is contradicting some other studies. When it comes to response time difference there was no significant difference;

- **Endurance test**: This test showed that *Consul* is a better tool for service discovery than *Eureka* having significantly higher throughput and lower average response time.

This work concluded that under normal load these two architectures have similar performance, when loads are small a monolithic architecture is recommended otherwise a microservices approach should be considered. Regarding the service discovery aspect, although not crucial to this thesis, showed that *Consul* offers better performance services.

From a components point of view, there isn't a consensus on how responsibilities should be shared among the different modules but a layered component architecture was recurrent in the approaches ana-

lyzed. The number of layers may vary but according to Sethi and Sarangi (2017), between 3 and 5 layers is an appropriate division.

Lin et al. (2017) performed a survey analyzing different kinds of layered architectures. This work stated that when using three layers they should be divided as follows: Perception layer, Network layer, and Application layer. The Perception layer is composed of the sensors and actuators present in the system, this is the bottom layer that interacted with the environment in the real world, this component has the responsibility of gathering data that will later be sent to the layers above, this layer is the the "*Things*" in *Internet of Things*. The Network layer can also be named transportation layer, serves as the receiver of information from the Perception layer and then is its job to distribute it to desired applications through efficient routing connecting sensors with servers, this layer can also perform some kind of data processing to make transportation easier. Finally, the Application layer, or Business layer, is responsible for receiving data from the Network layer and perform the desired actions to fulfill the objectives of the system. These goals can range from using the data received to train and make predictions about a future state of the environmental surroundings, the sensors to the display and storage of relevant live data for future human analysis. Figure 3 illustrates this the placement of each component.

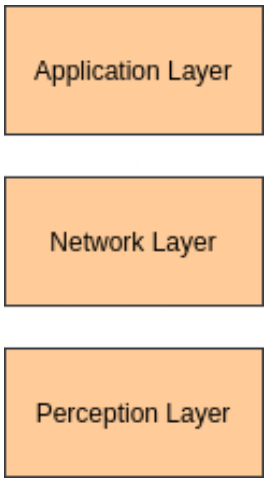


Figure 3: Three Layer Architecture diagram.

The same survey includes a Service Oriented Architecture (SOA), focusing on designing an orchestrated set of services in which making reusability a major part of this type of architecture. When it comes to layers, is added one, a Service layer, between the Application and Network layer. The Service layer acts as an interface or middleware to perform various tasks such as service discovery or service management making sure the service is provided efficiently.

Sethi and Sarangi (2017) proposes two types of layered architectures, the same three layers Lin et al. (2017) analyzed and another five-layer. In this five-layer architecture, the Application and Perception layers remain the same and are added three more layers, the Transport, Processing, and Business layer. The Transport layer is responsible for the transmission of data between the Perception and Processing layers through networks, this layer differs from the network because there isn't any processing happening here. The Processing layer, also called a middleware layer, analyses, and stores large amounts of data transported from the perception layer through the transport layer. The Business layer manages the system as a

whole, from the user's privacy to the IoT system and profit models. In Figure 4 can be seen the reshaped architecture.

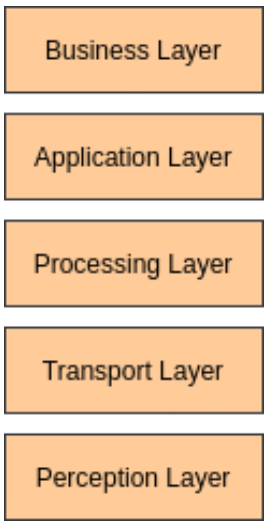


Figure 4: Five Layer Architecture diagram.

Architectures aren't one size fits all but, for example, El Hakim (2018) analyses IoT's ecosystems and presents the same Five-layer architecture as a general approach to this type of problem.

Nastic et al. (2014) looks at the integration of IoT with the Cloud in a finer detail introducing Software-Defined IoT units. These units are an abstraction of low-level components allowing the creation of a well-defined API interface allowing a fine grain resource network. Asir et al. (2016) also performed research on how IoT infrastructure, including Sensors, Actuators, Communication Brokers and Gateways are becoming software-defined and controlled. These small units can be controlled and provisioned in runtime via well-defined API layers, they also encapsulate functional requirements and non-functional requirements in each one. The units can be based on OS-virtualization such as Virtual Machines or in a container-based solution. In Figure 5 can be seen as a generalized diagram of this concept.

Summarizing Nastic et al. (2014) work, the principles of Software Defined IoT are:

- API encapsulation;
- Fine grain consumption;
- Policy based configuration and specification;
- Automated provisioning;
- Cost awareness;
- Elasticity support.

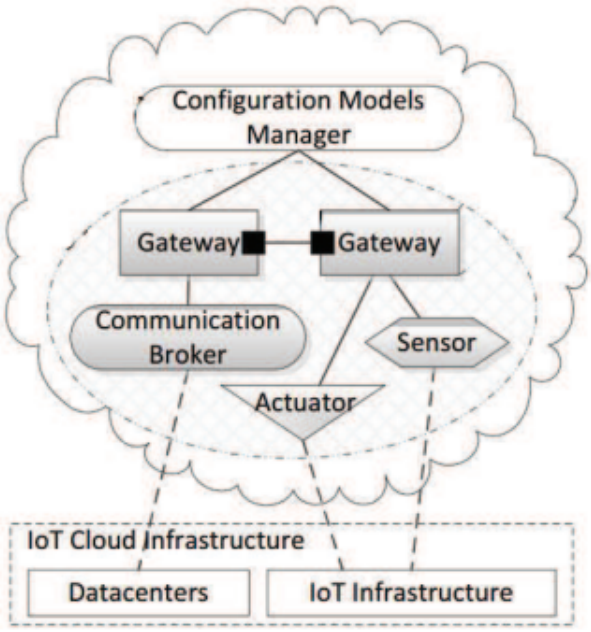


Figure 5: Software defined units diagram. Obtained from Asir et al. (2016).

2.1.2 Middleware

When it comes to integrating sensors and actuators a middleware layer is often needed in order to convert data coming from the sensors to a more friendly format (Lin et al., 2017; Preuveneers and Novais, 2012). This is what it's called interoperability, Desai et al. (2014) divided interoperability into three fields:

- Network Layer Interoperability;
- Semantic Interoperability;
- Syntactical Interoperability.

Network interoperability can be defined as the network protocols needed to maintain a connection between physical devices, these protocols range from *Bluetooth* through *ZWave* and *ZigBee*. Semantic interoperability refers to the different data formats that are produced from heterogeneous nodes and need to be aggregated into a format that contains a level of semantic annotation, this process requires a lot of manual human work to develop an application. Syntactical interoperability refers to the communication process present in the IoT field that allows the physical layer to stay connected to the cloud layers of systems.

Sethi and Sarangi (2017) work analyzed the challenges addressed by middleware and solutions available to build a strong IoT application.

2.1.2.1 Challenges middlewares attempts to resolve

Interoperability and programming abstractions are two of the biggest problems in IoT, every middleware work analyzed mentioned this, the orchestrated collaboration between different devices, protocols, and technologies.

Device discovery and management, all devices present in the network need to be interconnected and aware of neighboring devices as well as the services they provide. This solution must be scalable because the number of devices interconnected are never the same and can increase rapidly in some cases, the system must be prepared for this to happen. Middlewares typically mitigate this with discovery Application Programming Interface (API) that allow the listing of devices. Load balancing is also a main concern in this feature because a middleware should manage devices battery levels if needed as well as error reporting to users.

Scalability, as mentioned above, the number of devices can escalate quickly and all of those nodes are expected to communicate with each other, it's the middleware's job to manage this by adapting infrastructure when required.

Big data and analytics, sensors in an IoT environment, normally, collect a large amount of data that needs to be normalized, processed, and sometimes display to the user. Sometimes due to intermittent connectivity to the processing units of the system that information might be incomplete, by using ML and big data algorithms, a middleware should take this into account.

Security and privacy, typically IoT systems are present to measure some of society metrics, whether it's on a more personal level or in a more general way. As such middleware should address these concerns mechanisms like user authentication and data anonymization. *Cloud service*, the cloud is one of the most important parts of IoT systems because Most data is stored and analyzed in a cloud context, a middleware should let users leverage and make the most out of this cloud environment.

Context detection, the data collected from these sensors comes with various types of meaning and it needs to be extracted the context to make better decisions when it comes to process it and build better systems.

Already there are some middleware technologies available in the market that try to mitigate some of the challenges mentioned, some examples are *OpenIoT*, *Oracle's Fusion Middleware* and *Hydra*. Most middlewares fall into one of the following categories based on their design.

Event based, in this method, the interaction between components is made through events generated by producers and processed by consumers. In general, this architecture is similar to the publish/subscribe system.

Service oriented, this type of middleware is based on SOA, this means that there are multiple components in the system that can be accessed by interfaces, the resources available through these interfaces are observed as services providers. This class of middlewares publishes their functionalities and details in an accessible repository making it discoverable to consumers

Database oriented, in this approach the network of sensors is viewed as a virtual relational database that can be queried, the extraction of information is made easy but this type of middleware has difficulties scaling due to their centralized model.

Semantic, this middleware gives priority to the interoperation of devices that communicate using various types of data formats. For this, it is needed an interface that has many adapters capable of supporting many formats and ontologies, as many as the different types of devices the system contains.

Application specific, this type of middleware exists because, sometimes, is needed a fine-tuned piece

of software to meet the requirements at hand. This might mean that the middleware is tightly coupled with the middleware and is not trying to generalize a solution.

This study concludes that although some middleware technologies have grown and gained maturity although there is still lots of work to do to create a more generalized approach that fits more requirements and solves most of the existing challenges.

Ibrahim and A Rashid (2019) developed work about a lightweight middleware to accelerate the development of IoT platforms by developers who are not experts on pervasive computing. To do so, the authors propose a future-proof lightweight middleware that can be hosted easily in many data centers and provides an efficient way of sharing resources. The referred work based their general architecture in the layered approach having the physical, gateway, middleware, and application layers, in this order from the bottom up. This choice was supported by the requirements at hand.

Actor-based architecture is the middleware structure utilized. This allows a distributed solution in which the middleware can be deployed in every layer of the system depending on the needs at a certain moment. The technology utilized for this work was *NodeJS* due to its simplicity. The system is based on a *RESTful* service for the client, and the communication between the server and client is made through Hypertext Transfer Protocol (HTTP) using a JSON data format. This middleware exposed GET, PUT, POST, and DELETE requests.

To measure performance and results were made a series of requests assessing the response time and response size. To make the desired requests it was used *POSTMAN*. The GET requests averaged a response time of 275 milliseconds and a response size of 582 bytes, the POST requests averaged 762 milliseconds of response time and 578 bytes of response size.

This study concluded that the proposed middleware has advantages over other middlewares due to the use of JSON instead of Extensible Markup Language (XML) in the security side of things. The authors also note that in a *SOA* based architecture, where multiple nodes are deployed, this solution has the upper hand when compared to more mainstream commercial options. These are usually developed in *Java*, the same middleware in *NodeJS* occupies 74MB of memory when compared to 180MB of a similar *Java* system.

There are market solutions that try to address the challenges and have developed useful features to the integration between the cloud and IoT devices. Ngu et al. (2016) made a survey analyzing the different characteristics of this type of software.

Hydra, is a service-based IoT middleware funded by the European Union to develop a platform for embedded systems. A web service is provided as a way to agglomerate all physical devices into applications and then be able to control it no matter their network capabilities or technologies. This software offers service discoverability by using ontologies that semantically describe the features of the IoT devices. These services are offered by a low-level SDK. The uses of this middleware framework are mostly in agriculture, smart homes, and healthcare.

Global Sensor Networks, is also a service-based middleware that attempts to deliver integration, sharing, and deployment of heterogeneous IoT devices. This middleware is based on the virtualization of sensors to specify XML descriptors for deployment. This implementation follows a container architecture

to allow lifecycle management of sensors, including persistency, security, and event processing to name a few. The virtual sensors can be accessed by an API or other *RESTful* service.

Google fit, is a cloud-based middleware for the fitness and health ecosystem. By providing an API allows users to control their data and build apps on top of this software. The main communication protocol with sensors is *Bluetooth* or the implementation of a specific method by the developer. This middleware only provides fitness and health functionalities not allowing it to escalate and be used in a more generalized way.

Xively is considered a cloud-based middleware. It provides a web application to connect sensors with the cloud and allow users to extract the sensor data at any time. In this software is assured scalability in every action. Database services are in this software package. For ease of integration in companies, this middleware provides interfaces to use in Customer Relationship Management, Enterprise Resource Planning, and Business Intelligence software. Not everything is as seamless as it appears, integrating this service in an enterprise environment requires good programming skills.

Calvin is an open-source middleware developed by *Ericsson*. It combines an actor-oriented model and flow-based computing into a hybrid framework to build IoT applications. The actor model follows an asynchronous atomic callback pattern to achieve a high-performance interaction. This makes *Calvin* a lightweight middleware with minimal latency and good resource management.

NodeRED is another open-source option design by *IBM*. As the name suggests, it's based around *NodeJS* leading to a lightweight fingerprint and, therefore enables it to run in the system's edge. The strength of this system is the Graphical User Interface allowing users to drag-and-drop components creating IoT applications. To integrate this service on devices, some libraries provide these features, including authentication via password.

In conclusion, there are many middleware options in the market, commercial, and open-source that provide useful functionalities. All are different from each other, providing different types of communication interfaces and architecture implementations to cover as many requirements as needed. Nonetheless, generally, is difficult to integrate them in heterogeneous environments.

2.1.3 Communication

When it comes to devices communicating with the cloud, there are already two main patterns established. MQTT and Constrained Application Protocol (CoAP) are lightweight communication protocols that allow data transfer from devices at the edge to centralized servers. MQTT uses a publish/subscribe system, and CoAP allows a request/response system. In the next sections is a detailed analysis of both.

2.1.3.1 MQTT

According to the official website MQTT.org, MQTT, is a machine-to-machine communication protocol based around *TCP/IP* protocol. This protocol was developed by *IBM*. Publish/subscribe is the base architecture for MQTT, a one-to-many connection to efficiently distribute messages. It allows bidirectional communication, either cloud to device or device to the cloud. All these communications are secure by

using a *TLS* encryption of messages and by requiring authentication by the client via a safe protocol such as *OAuth* for example. MQTT clients are small allowing them to be easily supported by microcontrollers consuming little resources and maintaining small headers. The size of a packet is as small as two bytes, causing little stress to the network, even in unreliable connections. The persistence sessions between the broker and the client reduces time by not requiring a re-connection every time a message is sent. In Figure 6, there is an overview of this protocol.

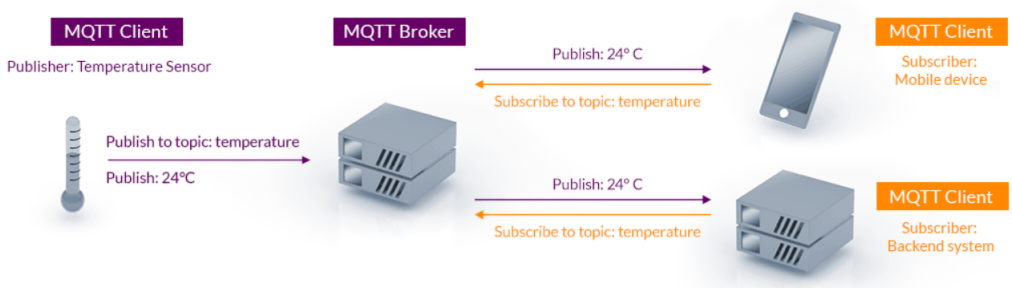


Figure 6: Architecture overview of MQTT. Obtained from MQTT.org.

Soni and Makwana (2017) analyzed the different features of MQTT, the following were highlighted:

- Publish/subscribe: Clients subscribe a topic and receive messages when a publisher sends data to that same topic. Clients can also publish data on a topic, the publisher can access hose messages;
- Topics: A hierarchy structure is created with topics, and subscribers read messages published by producers there;
- Quality of Service levels: QoS is the agreed-upon security in the distribution of data between parts in the system. QoS0 represents a message that is sent at most one time and it's not assured that a subscriber gets it. QoS1 means that a message is sent at least one time, and the subscriber might get several replicated messages. In QoS2 mode, a message is sent precisely one time and its reception is assured by having a 4-way handshake;
- Retained messages: After a broker distributes a message to subscribers in a topic, saves it. When a new subscriber enters the topic, they get all previous messages;
- Clean sessions and reliable connection: When a subscriber enters the broker's system its created a permanent clean session. If the client leaves and then joins again, all previous non-received messages are delivered;
- Wills: This is a message sent from the client to the broker informing him that if a sudden disconnection occurs they must publish that message in the topic to inform other clients.

MQTT offers flexibility when it comes to the message format. Any data format is supported. The developer has to choose a suitable pattern to send information.

Atmoko et al. (2017) compared this protocol with a simple request/response HTTP protocol in a IoT experiment. This experiment consisted in sending temperature and humidity values to a mobile app and

a web app. The values received were also registered in a database. The subscribing app communicated with the broker via *web-socket*. To make the temperature and humidity sensor connected to the broker, the author used a *ESP-8266* board.

The experiment consisted in reading values from the sensor for sixty seconds and record those values in the database, this process was repeated six times for each protocol.

The results achieved were very clear, MQTT sent, on average during one minute, 6520 data objects to be save in the database compared to 934 transmitted by HTTP.

In conclusion, this study shows that is clear that a traditional protocol is very inefficient when compared to MQTT.

2.1.3.2 CoAP

According to Bormann, Constrained Application Protocol is rapidly becoming one of the standards when it comes to Machine-to-Machine communication in an IoT environment. This protocol was developed by the *Constrained RESTful Environments* workgroup at the *Internet Engineering Task Force*. CoAP is modeled after REST and makes available to its user's resource methods such as *GET*, *POST*, *PUT* and *DELETE*. The goal of this protocol is to work on low resources microcontrollers with 10KiB of RAM and 100 KiB of code space. Since CoAP is so similar to HTTP because of their shared REST model, the integration of this protocol in applications or web clients has a high level of simplicity. One important aspect is that it doesn't enforce a data model. The data frame is agnostic to the payload it carries. One less highlighted feature of CoAP is the support for resource/observe architecture (similar to publish/subscribe). This is accomplished by, instead of having topics exists a *URI*, a Universal Resource Identifier. The subscriber subscribes to a resource with a specific *URI* and the publisher puts data in that same *URI* resource for subscribers to access.

CoAP relies on User Datagram Protocol (UDP) as a transport protocol and *DTLS*, Datagram Transport Layer Security, to maintain security. By using, UDP the communication network might seem unreliable, but, CoAP implements a *"confirmable"*, *"non-confirmable"* system. When a message sent is *"confirmable"*, the receiver must respond with an *ACK* as soon as the packet is received. If a message is *"non-confirmable"*, there are no guarantees that a receiver got the message (Bormann et al., 2012).

Thangavel et al. (2014) compared the performance between CoAP and MQTT. This study focused on two metrics delay, and total data transferred per message in bytes. The setup consisted of one laptop running both the CoAP and MQTT servers, a single board computer running a common middleware that acted as the publisher, and a netbook to emulate a loss of packets in the network. All messages were routed through the netbook before reaching the broker to perform the packet loss. *Wireshark* was utilized to record the number of transfered bytes. The *QoS* level of MQTT was *QoS1*.

The results achieved demonstrate a 100% ratio of messages delivered in one broker and one server environment with packet loss. We can conclude that the retransmission process is optimized on both protocols. Now looking at the delay variable, MQTT performed better when the loss of packets was low, CoAP was the clear winner when the loss of packets increased with a lower delay between messages. This

can be explained because MQTT relies on Transmission Control Protocol (TCP), a much heavier protocol comparing to UDP used by CoAP.

The total transferred data per message increased both in MQTT and CoAP environments. The increase was linear in both protocols until the packet loss reached 25%, and MQTT's total transferred size jumped considerably from 800 bytes to over 1200 bytes compared to CoAP's jump from 500 bytes to 600 bytes. The base size was different between protocols, CoAP was much lower, starting at 300 bytes, and MQTT starting at just under 600 bytes.

In conclusion, CoAP is a better protocol in a less reliable network with lower bandwidth resulting in lower transferred bytes and lower delay. If the network is fast and reliable MQTT shows lower delay and, because the network shows higher levels of bandwidth, the overhead caused by TCP is not relevant.

2.1.3.3 AMQP

One alternative to both MQTT and CoAP is Advanced Message Queuing Protocol (AMQP). This protocol supports publish/subscribe and request/response architectures. According to AMQP, this protocol is characterized as a lightweight machine-to-machine protocol developed by *JPMorgan Chase*. In its key features are security, interoperability, reliability, and open-source.

Naik (2017) compared AMQP with MQTT and CoAP and concluded the following aspects. AMQP has a higher message overhead and higher power consumption than both previous protocols. Also has higher latency than MQTT and CoAP but achieves a higher level of reliability than CoAP although lower than MQTT and inversely shows lower levels of interoperability than CoAP. Security and provisioning are where AMQP shines having higher levels than MQTT and CoAP. When it comes to industry usage AMQP is more utilized than CoAP but shows lower levels of standardization.

2.1.4 Deployment

To allow access to the developed system the deployment of all components in the cloud is necessary. There are many options when it comes to hosting and all options can be classified as either Public Cloud or Private Cloud (Bamiah and Brohi, 2011).

According to Charan et al. (2011), public clouds are the most common way of deploying apps. Servers are made available by companies for users to access them in a pay-as-you-go manner. Companies such as Amazon or Google let users choose the type of computing power they need for their systems and allow a much easier way of scaling and sharing resources.

In Bamiah and Brohi (2011) is explained that a private cloud is an infrastructure dedicated to only one organization. The resources available are not shared between groups and the physical servers can be on-site or off-site. Normally, this is an expensive way of deployment because the organization needs to lease or buy the servers.

Organizations with more resources normally have a hybrid deployment strategy having private servers to host the critical infrastructure and all surrounding platforms are hosted in a public cloud. This architecture allows a highly available platform being able to host replicas of all systems across multiple servers

(Charan et al., 2011).

2.1.5 Secure IoT and Cloud interactions

Nowadays security is a priority in every software system and IoT systems that is no exception. This means that the data that flows from the sensors to the cloud needs some form of prior authentication between the two parties to make sure that mainly the sensors are trustworthy.

El-hajj et al. (2019) analyzed and collected in a survey various forms of authentication identifying different forms:

- Token-based;
- Public and private key encryption;
- Credentials;

Token-based authentication consists on relying on a piece of data called token generated by a server respecting a certain protocol such as *OAuth* or *OpenID*.

Public and private key encryption involves the use of hashed asymmetric keys in order to make sure that only the end recipient of a message can decrypt it with the matching key. In this survey is also mentioned that symmetric keys can be used as well.

The credentials-based method makes use of the combination of username and password to authenticate the parties involved when data is shared often using protocols like *TLS* or *DTLS*.

In Dammak et al. (2019) is detailed an implementation of a light weight token-based authentication in which is developed a solution to maintain trust between user devices and IoT gateways. The breakdown of the process is as follows:

1. Registration Authority (RA) distributes to each of the gateways and smart devices the gateway contains a 160 bits token (with this token trust can be maintained between the smart devices and its nodes);
2. User registers with the RA and receives a 128 bit token;
3. User token is shared with the gateway nodes and smart devices;
4. User logs in with the token and identity can be verified by both the gateways and smart devices.

This study goes on and concludes that this protocol provides a high level of anonymity, Perfect Forward Secrecy and resilience.

Bhawiyuga et al. (2017) investigated how to integrate token based authentication in a MQTT broker to ensure every message came from trusted devices bearing the authentication token. In this study, the author claims that if the communications broker ensures the message comes from a trusted device, the cloud is safe from processing data coming from undesired sources.

The flow of the broker authentication is as follows:

- Request token using a username and a password;

- Authentication server asserts credentials;
- Authentication server sends token to device if credentials are valid;
- Device connects to broker using the token;
- Device ready to publish or receive messages;

This simple flow ensures the security of every message traded with the broker.

2.2 Crowdsensing

There is a considerable amount of research done analyzing different approaches on how to count people indoors and outdoors. Some studies utilize invasive technologies such as cameras and others force people to carry some kind of specific device that can be tracked. There are also less invasive methods that have been analyzed by the community but, normally, they produce less accurate results when compared to the more invasive ones. According to Yuan et al. (2011), the use of invasive technologies like cameras are not only over invasive but also require a lot of computational power to process and retrieve useful information, this author also refers that crowd estimation using wireless sensor networks can produce better results than cameras in environments that are influenced negatively by light. As ethics is technology is getting more important than ever this work will only contemplate non-invasive techniques to sense the environment in a smart room.

2.2.1 WiFi Probing

Large numbers of studies were made around the topic of *WiFi Probing* as this represents a passive and non-invasive way of having the perception of how many devices are in an area and then trying to translate that on knowing how many people that represents. The number of smartphones in the world, according to Murphy (2019), has already surpassed the number of population in the world and, as seen in Figure 7, the number is increasing every year. With this data, it can be assumed that the majority of people carry around their smartphone with the *WiFi* active. Although this work doesn't rely only on smartphones but also other devices such as computers or wearables, for example, this demonstrates the extent of *WiFi* enable devices a person carries daily.

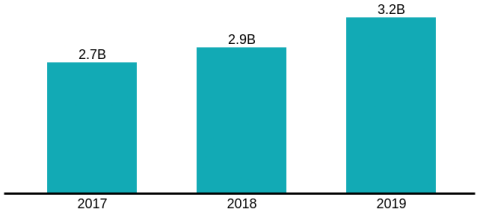


Figure 7: Smartphone number growth between 2017 and 2019. Obtained from Gu (2019).

Regarding how *WiFi* probing works, it is built based on the *IEEE 802.11* standard. A probe request is made when the device has the *WiFi* interface active allowing it to send out *request* frames containing information about itself to nearby access points, to inquire availability and information on how to connect to them if there is an access point available for connection when it receives the *request* frame generates a *response* frame with data about the Access Point (AP) that is sent to the device Zhou (2017), as seen in Figure 8, this method is called active scanning. It's also important to mention that inside the *request* frame is the device's MAC address allowing the AP to identify it. There is also passive scanning that is performed by the AP sending out *beacons* to notify devices of its presence, for this work this method is not relevant.

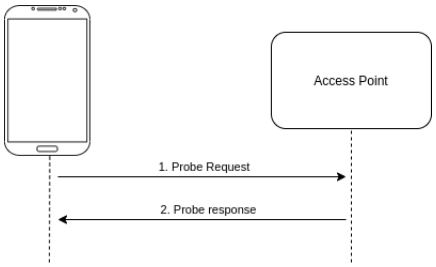


Figure 8: WiFi Probing schema.

The RSSI is also a relevant metric when probing the surrounding (Fernandes et al., 2018), helping to decide on whether the device is inside the area of importance to the system.

Schauer et al. (2014) performed important work on combining different approaches to crowdsensing pedestrian flow using probing requests. It relied on the MAC addresses received on probing requests belonging to only one person and measuring movement by capturing the same MAC address in different nodes of the system. The author developed more refined strategies on top of the initial one making use of time and RSSI to improve his solution coming to a hybrid method of using MAC addresses, the time when they were captured by the nodes with a RSSI value above a certain threshold. This threshold is very hard to find due to the overlapping zones.

As nowadays security and privacy of data are very important smart devices manufacturers begun to implement MAC address randomization in their software making it more difficult to identify a device (Oliveira et al., 2019). Martin et al. (2017) conducted a study on MAC randomization and concluded that *"MAC address randomization policies are neither universally implemented nor effective at eliminating privacy concerns"*. This randomization could harm some studies, such as Schauer et al. (2014), but the fact is that it's not widely spread or implemented.

The accuracy and range of *WiFi* probing outdoors were measured to an extent by Fernandes et al. (2018), who performed a study with ESP8266 ESP-12E NodeMCU Amica board, low power and reduced cost Arduino board. Fernandes et al. (2018) concluded that on an open area, a probe request is detected as far as 27 meters from the sensor and within a circular area of 2290 square meters. On the accuracy side, the author conducted an observation of the real world while the developed system was counting probes and deciding whether they were unique or repeated, the results of this research are in Figure 9.

The figure above makes us conclude that according to Fernandes et al. (2018), we can have 92.8%

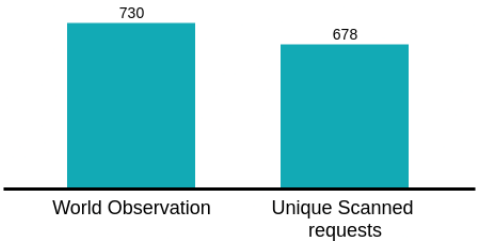


Figure 9: Unique requests vs Observed people. Obtained from Fernandes et al. (2018).

accuracy when deciding if a probe is unique or is just a repeated request or a different device.

Most of the research analyzed focuses on outdoor spaces, Mehmood et al. (2019) studies *crowdsensing* inside buses allowing us to extract more relevant information on how a system based around probe requests would perform indoors. This study aims to design a system that estimates the occupancy of a bus by using a combination of time, probe requests, and their RSSI. The solution proposed utilizes more than one sensor that is capable of detecting probes. To solve this challenge the following algorithm was developed:

- All sensors receive a probe request in a given time interval;
- All sensors receive an RSSI above a minimal defined threshold;
- The first and the last probe have a defined number of seconds apart;
- The request is detected at least a given minimum amount of times;

After several real-life tests, this study concluded that the use of only one sensor leads to results five times greater than what was happening in the real world. The best results are when used either three or four sensors concurrently. In particular, using four sensors resulted in only a 15% overestimation. This overestimation proved a real challenge due to external influence in comparison to Fernandes et al. (2018) underestimation outdoors.

2.2.2 Bluetooth

Bluetooth is also a non-intrusive way of crowdsensing. This method relies on smartphones having the Bluetooth function enabled and set in discoverable mode. The second part of the last sentence is the more difficult one. According to Weppner and Lukowicz (2013), the probability of detecting people through their smartphone is about 10% meaning the accuracy of this method is low.

Another Bluetooth crowdsensing method is using Bluetooth Low Energy (BLE). It is a more invasive technique because users are forced to carry small identifying items with them, making them visible to the system. This is accomplished by using a Beacon like in Fernandes et al. (2018), these beacons are low cost and long-lasting pieces incorporated in clothes as an example.

These methods, both smartphone Bluetooth detection and BLE have proven to be, in one case low accuracy in normal conditions and, in the latter too invasive for the proof of concept idealized. This means that no further research will be performed.

2.3 Machine Learning

ML is soon becoming an essential part of IoT. The massive data that the sensors gather and store are used to develop accurate ML models. ML can be described as a subset of Artificial Intelligence that, without being explicitly programmed, makes some predictions based on data analysis and experience. ML models are programmed to make small evolutions every time new data is inserted in them, and by doing small increments improve the results. In the realm of ML learning paradigms, there are three main types: *Supervised*, *Unsupervised*, and *Reinforcement* (Padala et al., 2019; Bansal and Sharma, 2020).

The goal of Supervised Learning is to create a mapping function in order to input a value and predict an output. To create these types of models is necessary a training dataset containing input and output values. The values present in the dataset can be categorized or continuous. After a dataset is built it's time to train the model to develop an accurate mapping function. When this process ends, the model is ready to receive never-before-seen input values and predict their output (Bansal and Sharma, 2020). Artificial Neural Network (ANN), are probably the most used and talked about algorithm when it comes to ML. This model is helpful for pattern recognition, classification, clustering and predictions. Currently, ANN, are becoming easier to use due to the rapid development of libraries such as *Keras* and *Tensor Flow*. ANN's are compose of independent layers, input, hidden and output, that contain nodes. The number of nodes can be arbitrary in each layer. These nodes are connected to other nodes in the next layer and the propagation of information between nodes is finalized with an output (Abiodun et al., 2018). Other very effective algorithms may be used like, for example, Long Short-Term Memory Networks to make predictions on time series problems (Oliveira et al., 2021).

Unsupervised Learning is a technique used for clustering and analyzes unlabeled datasets to discover the patterns in its features. These models don't have an expected output when are trained, instead, the model apprehends the traits of the dataset. This method can be unpredictable and, when not set up correctly, deliver bad results (Bansal and Sharma, 2020). In the real world, this approach has various scenarios in which it is applicable, for example, to detect abnormal levels of *pH* in Wastewater Treatment Plants water (Gigante et al., 2021).

Reinforcement Learning is a method taken from psychology where the action an agent takes is compensated in a good or a bad way. The goal of these ML models is to maximize the reward given a set of constraints and rules. The process of getting the maximum reward may take a while and, to avoid agents from being stuck negatively, normally is introduced an exploratory variable. This means that in a small percentage of actions, instead of taking the maximum reward route, it explores other options randomly (Bansal and Sharma, 2020; Padala et al., 2019; Abdualgalil and Abraham, 2020). In Figure 10 is a diagram representing this process.

2.3.1 Evaluation Metrics

To evalute the pergformance of a ML algorithm, there are a set of metrics that should be taken into account. According to Abdualgalil and Abraham (2020) some of these metrics are:

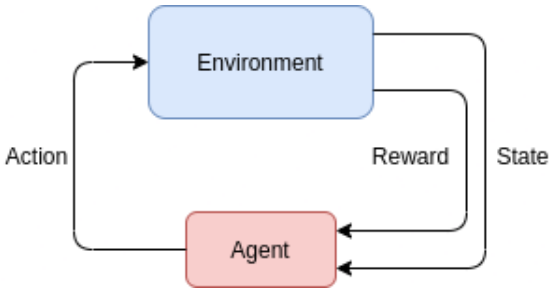


Figure 10: Flow of Reinforcement Learning algorithms.

- Classification Accuracy - is the ration between correct assumptions per total number of inputs.

$$Accuracy = \frac{Number\ of\ Correct\ Assumptions}{Number\ of\ Total\ Predictions} \tag{1}$$

- Measures of evaluation - are accuracy, precision and recall, and are determined by the confusion matrix shown in 1.

		Predicted Class: P	Predicted Class: N
Actual Class	P	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

Table 1: Confusion Matrix

  - Accuracy, performs an evaluation to the system determining the percentage of right evaluations.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2}$$

  - Precision, verifies the proportion of all positives that are real positives according to the total number of positives

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

  - Recall, calculates the proportion of actual positives measured correctly.

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

  - F-measure, combines the recall values with the precision. If the F-measure is high the system is well engineered and techniques are appropriate.

$$F\text{-}measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{5}$$

- Mean Absolute Error - is the average between the predicted values and the actual values, this alerts us to the difference between the prediction and actual output.

$$\text{Mean Absolute Error} = \frac{1}{N}\sum_{j=1}^{N}|y_j - \hat{y}_j| \tag{6}$$

- Mean Squared Error - is equivalent to Mean Absolute Error but it's a squared average. The estimation of gradient is easier with this method.

$$\text{Mean Squared Error} = \frac{1}{N}\sum_{j=1}^{N}(y_j - \hat{y}_j)^2 \tag{7}$$

- Mean Relative Error - gives us the relative estimate between the predicted values and the actual values.

$$\text{Mean Relative Error} = \frac{1}{N}\sum_{j=1}^{N}\left|\frac{\hat{y}_j - y_j}{\hat{y}_j}\right| \tag{8}$$

Chapter 3

System Architecture and Implementation

This chapter aims to explain the architecture and its implementation starting with a more general approach and afterwards diving down into each of the components as well as the techniques and communication patterns used to develop them.

3.1 Overall architectural design

After analyzing different strategies to design an architecture for this project's IoT system all conclusions pointed to a microservices architecture. This type of architecture allows more flexibility when it comes to the development of the platform itself. The various components can be designed and implemented separately allowing a more defined division of responsibilities in the system. When it comes to non-functional requirements this approach has the disadvantage that it needs more effort to deploy various systems instead of just a monolithic one, this con is eclipsed by the fact scalability is easier allowing horizontal scaling compared to vertical if it was a monolithic system (De Lauretis, 2019).

The need for the system to be *Cloud-based* comes from the fact that, in this scenario, is needed to maintain a state. If this state is managed by the edge devices, the CPU power needed to maintain the state consistent across all devices was more than we could afford. Also, in a scenario where the state was managed by devices, they needed to communicate to share information, putting strain on the network. Using the *Cloud* as a central entity this network strain is similar, if not less, when sharing the data produced by sensors. In addition to that using the cloud computing capabilities we gain a lot of CPU power to effortlessly consume and process data.

In Figure 11 is a high-level overview of the system's components.

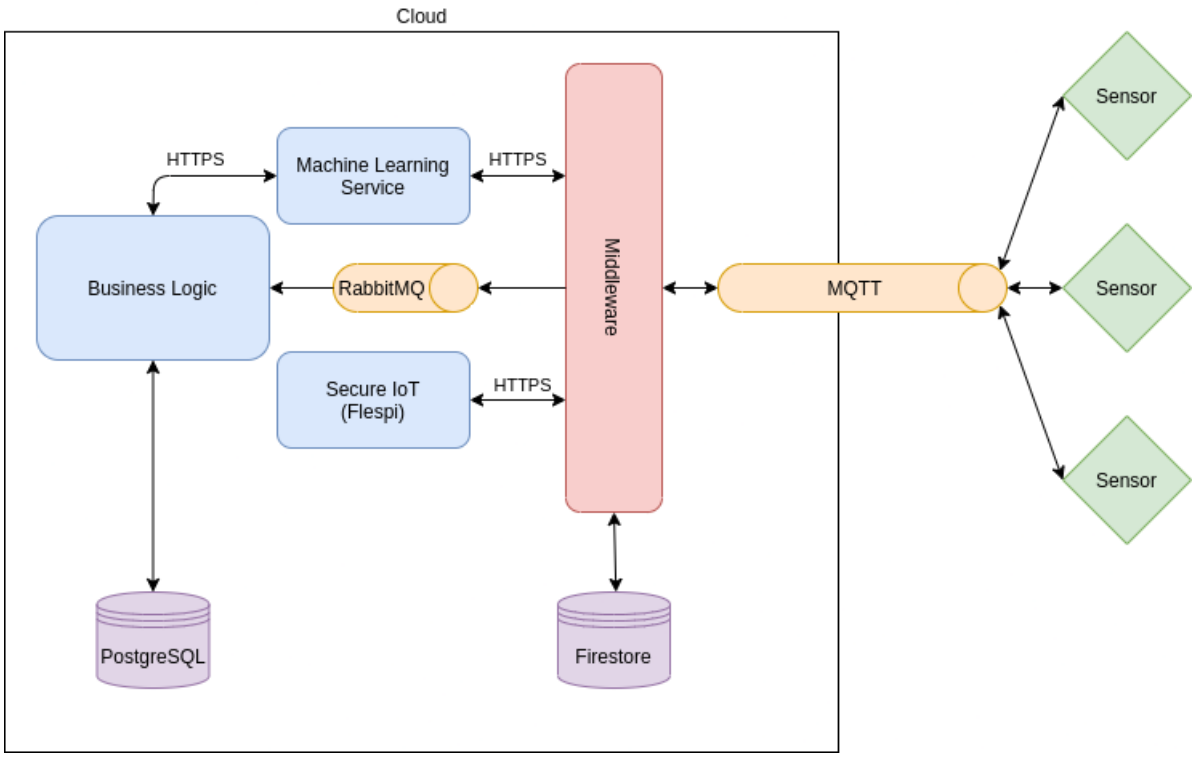


Figure 11: High-level architecture design.

In the previous image it's possible to see the breakdown of services and how they communicate. The *Sensor*'s are the gatherers of environmental data and send them to the cloud with minimal processing. The *Middleware* is the receiver of said data and processes it until it's in a stage that can be easily utilized by other services, all the data that comes to this layer is saved in a *Google Cloud Firestore* instance. The *Secure IoT* provides the authentication service of the sensors managed by the *Middleware*, it uses a *MongoDB* database to manage all the information it receives. The *Business Logic* service receives clean processed data from the *Middleware* and maintains the service that is provided with the help of a *PostgreSQL* database to store all business related data. Finally, the *Machine Learning Service* communicates with the *Business Logic* to improve the results that are sent to the user. In the following sections is a more detailed explanation of each component.

Go was the chosen development language, a compiled open-source programming language with simple syntax and efficiency to develop systems (Google). The developed system utilizes other technologies, mainly in the message broker realm. *Flespi* is the MQTT broker, and *RabbitMQ* is another message broker to communicate between services in the cloud.

3.2 Physical Layer

The Physical Layer is, as the name suggests, the part of the system that is in the physical environment from where we want to extract data. It is the "*Things*" in IoT. In the next subsection is a breakdown of what type of sensors were utilized to perform what jobs to meet the requirements listed in the previous

chapters.

3.2.1 Devices

To perform the sensing of the crowd in indoor spaces the has been chosen the *AZDelivery NodeMCU Lolin V3 Module ESP8266 ESP-12F* (shown in Figure 12). This board allows the detection of probe requests from the 14 interfaces present *WiFi*, containing the MAC address associated with the device it makes them. This is a low cost *Arduino* board (4€/per board). It contains 4MB of flash memory and 64kB of SRAM, this device is also low power.

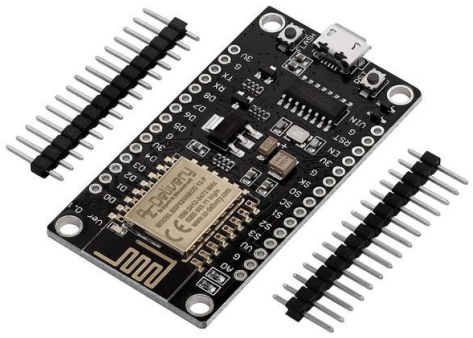


Figure 12: *AZDelivery NodeMCU Lolin V3 Module ESP8266 ESP-12F*.

The development and setup of these devices were made through the Arduino IDE, an editor that contains all the drivers and interfaces necessary for development in C++.

3.2.2 Probe Request Detection

Using a device like the one described in the previous section and with the help of open-source code (Kalanda; Brunofmf), the system can detect probe requests coming from surrounding devices in the different *WiFi* channels. The developed code can identify the RSSI, the channel of *WiFi*, the MAC address, and, if available the *SSID*. In Figure 13 is a screenshot of captured requests by a sensor.

```
RSSI: -32 Ch: 4 Peer MAC: c0:ee:fb:92:26:a6 SSID:
RSSI: -33 Ch: 4 Peer MAC: c0:ee:fb:92:26:a6 SSID:
RSSI: -29 Ch: 4 Peer MAC: c0:ee:fb:92:26:a6 SSID:
RSSI: -32 Ch: 5 Peer MAC: c0:ee:fb:92:26:a6 SSID:
RSSI: -28 Ch: 5 Peer MAC: c0:ee:fb:92:26:a6 SSID:
RSSI: -29 Ch: 5 Peer MAC: c0:ee:fb:92:26:a6 SSID:
RSSI: -30 Ch: 5 Peer MAC: c0:ee:fb:92:26:a6 SSID:
RSSI: -30 Ch: 5 Peer MAC: c0:ee:fb:92:26:a6 SSID:
RSSI: -9 Ch: 1 Peer MAC: d0:ab:d5:77:3b:61 SSID:
RSSI: -6 Ch: 1 Peer MAC: d0:ab:d5:77:3b:61 SSID:
RSSI: -6 Ch: 1 Peer MAC: d0:ab:d5:77:3b:61 SSID:
RSSI: -15 Ch: 1 Peer MAC: d0:ab:d5:77:3b:61 SSID:
RSSI: -50 Ch: 1 Peer MAC: d0:ab:d5:77:3b:61 SSID:
```

Figure 13: Probe Requests captured by a *NodeMCU Lolin V3 Module ESP8266 ESP-12F* sensor.

In listing 1 is a snippet of key code utilized to capture probe requests by the sensor module. It is relevant to refer that this code is open-source produced by Brunofmf.

```
WiFiEventHandler probeRequestCaptureDataHandler;

void setup(){
...
probeRequestCaptureDataHandler = WiFi.onSoftAPModeProbeRequestReceived(&
    onProbeRequestCaptureData);
..
}

void onProbeRequestCaptureData(const WiFiEventSoftAPModeProbeRequestReceived& evt) {
  if(currIndex < ARRAY_SIZE){
    if(newSighting(evt)){
      probeArray[currIndex].mac = macToString(evt.mac);
      probeArray[currIndex].rssi = evt.rssi;
      probeArray[currIndex++].previousMillisDetected = millis();
    }
  } else{
    Serial.println(F("*** Array Limit Achieved!! Send and clear it to process more
    probe requests! ***"));
  }
}
```

Listing 1: Probe Request code. Obtained from Brunofmf.

The code shown in Listing 1 is only the handler to a probe request, before this can be executed the sensor is configured as a soft AP, meaning that if a device opens the *WiFi* list, a new network created by the sensor is displayed. This configuration triggers devices to send probe requests to enquire about the state of this network allowing the sensor to log those requests.

Because one of the goals of this thesis is to prove that low CPU power, energy efficient sensor modules don't need to process data and leave that job to the cloud that effortlessly does this job, all the data gathered is sent to the cloud services almost untouched.

Before the data is sent through the proper channel the only processing needed to be done was the serialization to a JSON format. This serialization can be done in two ways, one more simple when the data to send does not exceeds 80 bytes and other more intricate due to the need to fragment the data into smaller segments. In listing 2, are the serialization formats. One thing to note is that in the fragmented JSON option, the content of the *Data* field, is a fragmented version of the simpler JSON option.

```
\\Simpler option
{
    DeviceID: 1,
    ProbeData: {
        MacAddress: "0:0:0:0:0",
        RSSI: "100",
```

```
        PrevDetected: 123
    }
}

\\Fragmented option
{
    Id: 1,
    Data: "data fragment",
    End: 0
}
```

Listing 2: Serialized JSON options.

The need of fragmentation at first was not evident when testing the system in environments with very few devices probing for available networks (0 to 3 devices). But when testing in a real world scenario with more than 3 devices present in a room the data was not being sent to the cloud. This problem came down to the library that was being used to publish the data (Industries) not allowing anything above 128 bytes to be published.

3.3 Middleware

By definition a *middleware* is "a type of software which manages and facilitates interactions between applications across computing platforms" (Sunyaev, 2020). In this thesis particulate case, it manages the interaction between sensors and a cloud service. This facilitation is needed due to the heterogeneous qualities of sensors and, in services that require multiple types of edge devices interactions, the data exchanged between them and the cloud has different formats. When it comes to the implementation, as was seen in the *State of the art* chapter, that is very difficult because in IoT systems the problem is almost certain to be specific to the context.

In this thesis system, the *Middleware* is the point of entry of data into the cloud. This means that is the first chance of applying heavy processing and algorithms, taking full use of the power available in the cloud.

3.3.1 Architecture

After analyzing the problem, its constraints, and types of middleware architectures the conclusion was that an Event-Driven approach to this middleware has beneficial properties but, SOA, also has its strengths in the particulate case of managing the sensors.

The Event-Driven approach seems the obvious choice to the systems *Middleware* architecture. This choice is backed by the communication pattern used between the sensors and the cloud. Data is produced when a probe request is detected by a sensor (event) and then sent to the cloud. The *Middleware* acts as a consumer of data in the scenario. Another reason why an Event-Driven architecture suits this problem

is how we send processed data to other services. In this case, as we want the data to be processed and showed in real-time, the *Middleware* needs to act as a producer of processed data for the *Business Logic* to consume.

A SOA enters the debate because the system needs to manage its sensors and authenticate data received by them. Traditionally, a SOA provides a set of interfaces to communicate synchronously with other services and, in this scenario, that is one of the requirements to manage (add) sensors to the system as well as checking if the data received belongs to a sensor inside the known network.

With the information described previously the choice of architecture evolved to a somewhat hybrid being predominantly Event-Driven to manage all received data and all outgoing data except on the management and authentication aspect that is Service-Oriented. According to the research performed regarding the types of *Middleware*, calling this approach Application Specific is also valid. In Figure 14 is a breakdown of modules inside the *Middleware*.

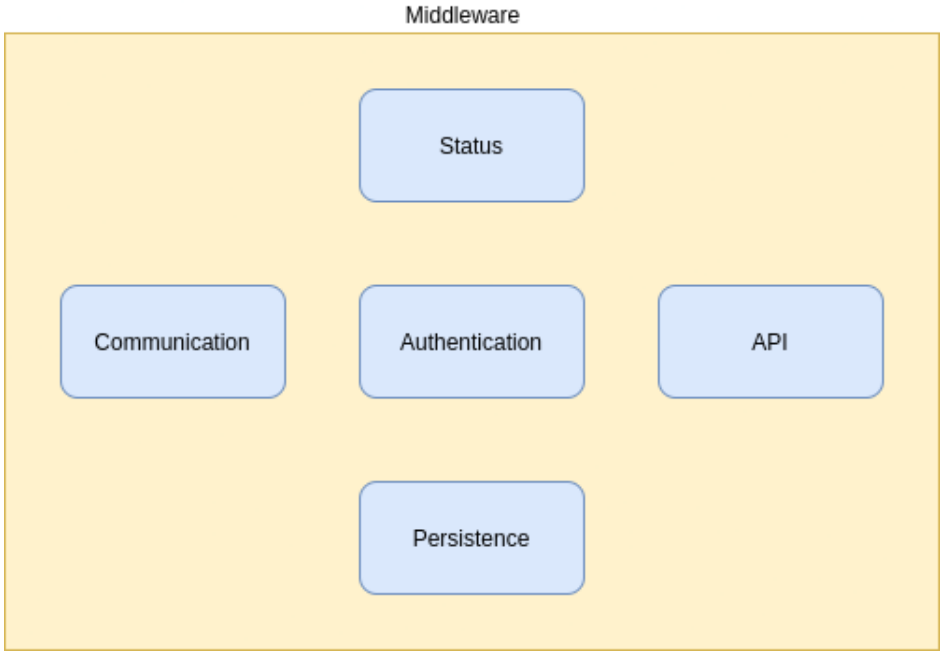


Figure 14: Middleware service architecture modules.

3.3.2 Module breakdown

3.3.2.1 Communication

The Communication module is in charge of maintaining all asynchronous communication between cloud and devices and, between different services inside the cloud.

To receive data produced by sensors, the system creates a lightweight thread that subscribes to the topic where messages are sent. After the reception of said data, there is a need to understand if the message is complete or if it's just a fragment in the callback function (Listing 3). If it's a fragment, the data is saved in a temporary structure until the whole message can be put together to be processed.

Subsequently, a timestamp of the current server time is added to the data. This added field is needed because in the current context the sensors utilized are unable to keep track of the current time without an add-on.

```
var f MQTT.MessageHandler = func(client MQTT.Client, msg MQTT.Message) {
  topic := msg.Topic()
  payload := msg.Payload()
  var result map[string]interface{}

  _ = json.Unmarshal(payload, &result)
  if result["Id"] != nil {
    handleFragmentArrival(payload)
  } else if result["DeviceID"] != nil {
    createProbeData(payload, topic)
  }
}
```

Listing 3: Callback function for MQTT message handling.

Another aspect where cloud to device communication is present is the management of devices, particulate when adding new sensors. After receiving a request to add a device, a credential token is sent to the sensor through the communication broker. In this scenario, the cloud is the producer and, the sensor is the subscriber. In Listing 4 is the Publish function.

```
func PublishToken(topic string, message string, client MQTT.Client) {
  token := client.Publish(topic, 0, false, message)

  if token.Wait() && token.Error() != nil {
    log.Fatal(token.Error())
  }
}
```

Listing 4: Function that publishes token.

Finally, after a message is received and processed, the information generated needs to be sent to the *Business Logic* side of the service. The cloud acts as a producer and publishes the desired message in a queue to accomplish that. From this queue, any service that has subscribed can consume messages. In Listing 5 is the function that publishes these messages.

```
func (mq MQClient) PublishToQueue(queueName string, payload []byte) {
  ch, err := mq.conn.Channel()
  if err != nil {
    log.Fatal(err)
  }
  err = ch.Publish(
    "",
    queueName,
```

```
    false,
    false,
    amqp.Publishing{
      ContentType: "application/json",
      Body: payload,
    },
  )

  if err != nil {
    fmt.Println(err)
  }
}
```

Listing 5: Function that publishes messages for Business Logic to consume.

3.3.2.2 Persistence

In the context of SaaS every piece of knowledge that can be extracted from the data is valuable and in this scenario of IoT as a Service and a crowdsensing environment that is no different. To save the data generated by the sensors was chosen a *NoSQL* Firestore database instance. This allows the system to save data in JSON format making querying simple. In Lisiting 6 is an example of a JSON saved in the database.

```
{
    MacAdress: "00:00:00:00:00",
    PrevDetected: 12345,
    Rssi: "10",
    Timestamp: march 8th 2021 15:30:30 UTC
}
```

Listing 6: Example of JSON saved in *Firestore*.

In Figure 15 is snapshot of how the data is organized in the *Firestore* database. The first column represents the collection in which the data is going to be inserted, in this scenario the name of the sensor, the second column is the unique identifier of the record and, in the third column, is the detailed data information.

Figure 15: *Firestore* data snapshot.

Although this data is not used in this service, is useful to to extract data and help build datasets to enhance the accuracy of the crowdsensing algorithm using Machine Learning.

3.3.2.3 API

The need to manage devices in a IoT service is a priority. Users may want to extend their system's capabilities, and to perform this action, sometimes adding new devices is key. The designed system enables the users to add new rooms to the service, and with that comes new sensors. To ensure all sensors are known to the service, a *Restfull HTTP API* was developed with a single endpoint to make the addition of sensors. The endpoint consists of a *POST* request where the device *ID* is in the body of the request. This event triggers a chain of events, ending with a published token in a topic consumed by the sensor.

To ensure the ML service has access to the number of probes made in a certain interval of time there is another endpoint. A GET endpoint that returns the number of probe requests made in the last minute. This is needed because the *Middleware* is the only service with access to *Firestore*.

3.3.2.4 Authentication

In these types of systems is needed to assure that the data received comes from trustworthy sources. In the authentication module, every time the system received a message from sensors is verified the existence of an access token and sent a request to the *Secure IoT* Service to make sure that the data comes from a registered device in the system.

3.3.2.5 Status

The Status module is in charge of applying a detection algorithm that decides if a probe is coming from a device associated to a person inside the room. Oliveira et al. (2019) researched the algorithm utilized in this thesis. This method relies on three structures to monitor the state of a room. One saves Potential Arrivals another, Arrivals, and, finally, one handles Potential Departures. The number of people in a room is dictated by the size of the Arrival structure. In Figure 16 is an activity diagram of the detection of presences in rooms.

The diagram displayed in Figure 16 only shows the detection of a presence. The second part of this algorithm shows us how to detect an exit and when to clean potential arrivals for the structure not to become cluttered with old data. This job is in charge of independent threads running while the system is active. Figures 17, 18 and 19 show the cleanup algorithms of these managing structures.

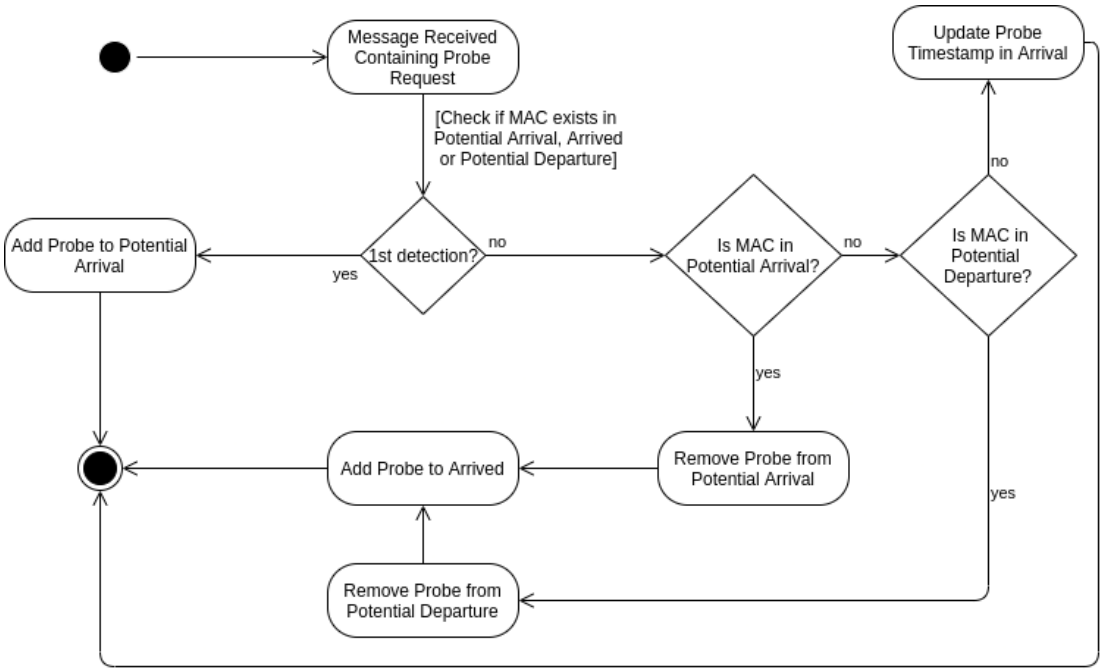


Figure 16: Algorithm to detect a device presence.

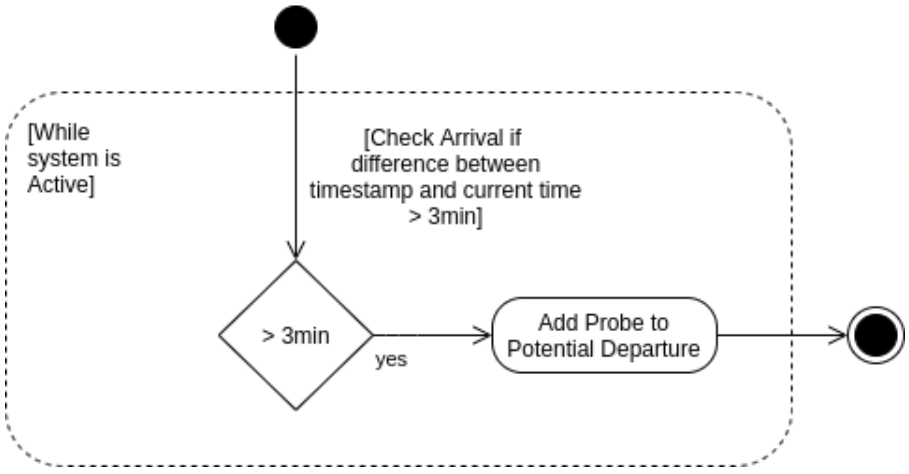


Figure 17: Algorithm to detect a presence potential exit.

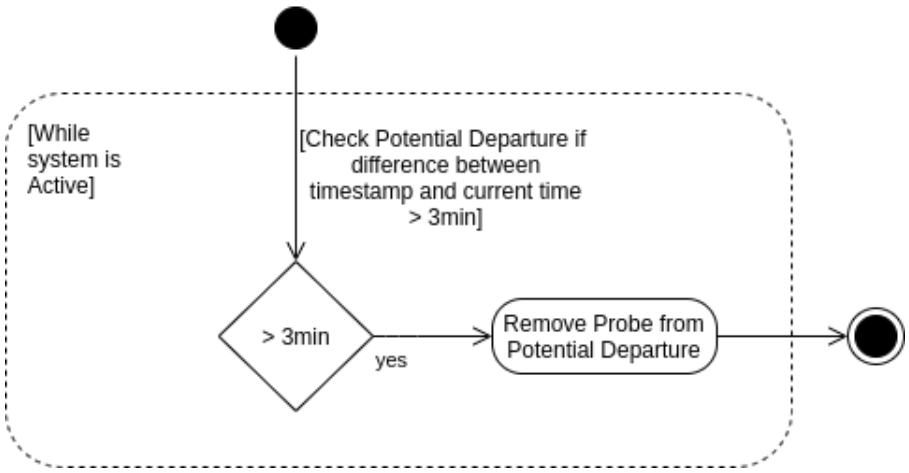


Figure 18: Algorithm to confirm a presence exit.

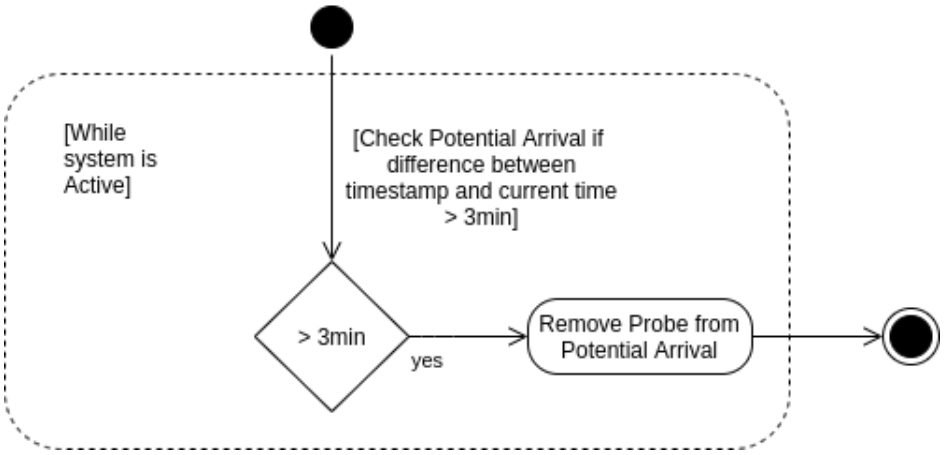


Figure 19: Algorithm to detect a misread presence.

The Status module was also in charge of generating messages to send to the *Business Logic* when existed a:

- Confirmed Presence (probe request from the same MAC is detected more than one time inside an interval of 3 minutes or, MAC in Potential Departure);
- Potential Exit (probe request from the same MAC is in the Arrival structure but isn't detected in 3 minutes)

In Listing 7 is an example of message informing the *Business Logic* of a change in a room state.

```
{
    DeviceID: "123",
    MacAdress: "00:00:00:00:00",
    Active: true, //false in case of exit
    Timestamp: march 8th 2021 15:30:30 UTC
}
```

Listing 7: Example of JSON message sent to *Business Logic*.

Since data is generated from sensors and at the same time cleanups are occurring, there is one concern, two threads trying to access the same structure, one to insert or update data and another to check and move/remove data. To mitigate this problem was designed a *Mutex* system (Mishra, 2018) ensuring integrity when accessing the data structure that contains the state, guaranteeing that only one thread can change it. In Listing 8 is a *Mutex* utilization example.

```
func (state RoomState) CleanPotDeparture() {
  state.InUse.Lock()
  for _, val := range state.PotDeparture {
    if time.Since(val.Timestamp) > time.Minute*3 {
      delete(state.PotDeparture, val.MacAddress)
      State.mq.PublishToQueue(os.Getenv("queue"), val.ProbeDataToMsg(false))
    }
  }
  state.InUse.Unlock()
}
```

Listing 8: Example of *Mutex* usage when cleaning the Potential Departure structure.

This piece of the middleware is the most critical for the service to work because it's in here that the algorithm detects the presence of people. This importance leads to the need for tests in this module. Several unit tests were developed, achieving over 95% of code coverage in every file inside this module. In Lisiting 9 is an example of the type of existent unit tests.

```
func TestCleanUpPotDeparture(t *testing.T){
  status.InitializeRoomState(test.CreateMQClient())
  status.State.PotDeparture[data.MacAddress] = data
```

```
  status.State.PotDeparture[data1.MacAddress] = data1
  status.State.CleanPotDeparture()

  _, ok := status.State.PotDeparture["123"]
  assert.Equal(t, false, ok)
  _, ok = status.State.PotDeparture["1234"]
  assert.Equal(t, true, ok)
}
```

Listing 9: Example of test of function *CleanUpPotDeparture*.

3.4 Business Logic

To transform a system into a *as a Service* software there needs to be a Business Logic component that aggregates all business related concepts and provides a service. In this dissertation, the Business Logic will provide a set of services, allowing the user to see real-time information, historic room state data and manage its rooms and sensors.

3.4.1 Architecture

After reviewing the requirements that the *Business Logic* had to fulfill, there was an obvious choice of architecture, SOA. This concept allows us to create a set of endpoints, enabling the user to send requests to obtain information about the service.

Another key point of the design architecture is the onion/hexagon architecture (Khalil et al., 2016). This type of design facilitates the implementation of key concepts such as the Dependency Inversion Principle, inner layers provide interfaces and outer layers utilized them, the coupling points inwards and, the implementation doesn't depend on the infrastructure. In Figure 20 is a breakdown of this service.

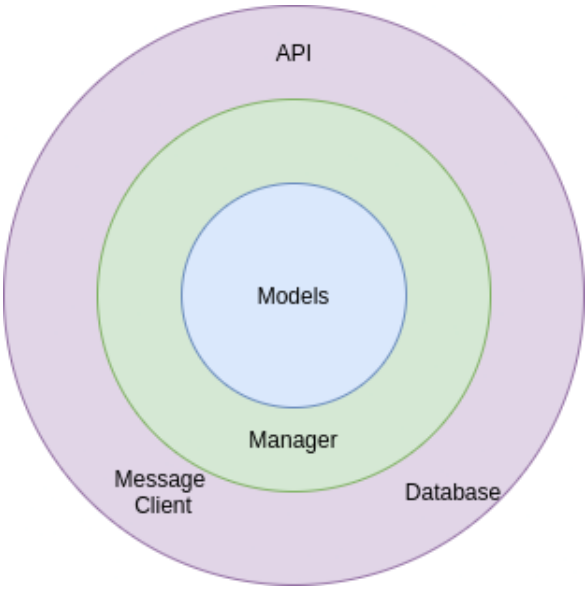


Figure 20: Business Logic service architecture

As this system provides a service, there is the need to save data. This need is mitigated using a relational database since all entities present in the system have a close relationship. There are three models in the system:

- Sensor - A sensor is composed of an *ID*, a name and a *RoomID* identifying the room of the deployment;
- Room - A room is composed of a name and an *ID*;
- Presence - A Presence is composed of an *ID*, a MAC, a last detected time, an active marker and a *RoomID* where the presence was detected.

PostgreSQL is the chosen relational database engine, is an open-source project widely spread and utilized by the community. This engine provides data types for ease of use, as well as, data integrity, concurrency, and extensibility, among other relevant features, making it a safe choice to store all of our service data. In Figure 21 is the database diagram.

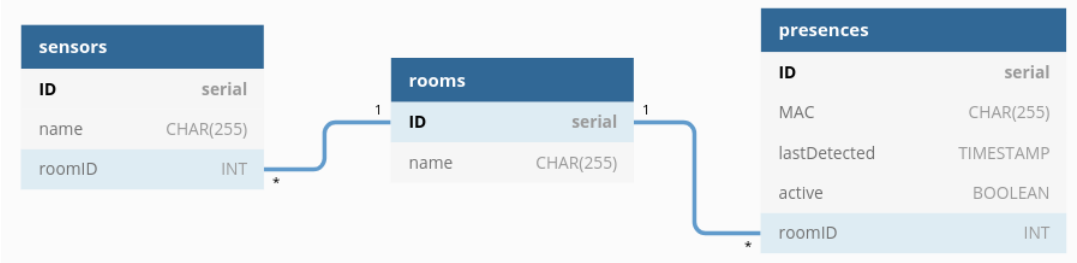


Figure 21: Business Logic database diagram

3.4.2 Module Breakdown

In this subsection is an explanation of all the modules individually of the service.

3.4.2.1 Models

The *Models* module houses the definition of the data structures representative of our domain objects and respectively associated methods. In this system there are three models specified in the previous subsection: *Room*, *Presence*, *Sensor*.

A *Room* is a place where resides a deployed system. A *Presence* is a device detection that can translate into a human presence. A *Sensor* is the physical device that is present inside the *Rooms* that detects the *Presences*.

3.4.2.2 Manager

The *Manager* module is responsible for being the bridge between our domain module (*Models*) and the infrastructure/communication modules. This is accomplished with the definition of contracts that outer layers need to follow to work with the system. In Listing 10 is the interface that the persistence package needs to implement and in Listing 11 is the interface the Message Queue subscriber needs to implement.

```
type Store interface {
  SaveNewRoom(room model.Room) error
  GetRoomByName(roomName string) model.Room
  SaveNewSensor(sensor model.Sensor, roomID int) error
  SaveNewPresence(presence model.Presence, roomID int) error
  GetRoomPresences(roomID int) []model.Presence
  GetRoomBySensorName(sensorName string) model.Room
  GetRooms() []model.Room
  GetActivePresencesByRoom(roomName string) []model.Presence
  GetAllPresencesByRoom(roomName string) []model.Presence
  UpdatePresenceState(MAC string, state bool)
  PresenceExists(MAC string, roomID int) bool
  UpdatePresenceLastDetected(MAC string, timestamp time.Time, roomID int) error
}
```

Listing 10: Interface contract to accomplish persistence.

```
type Queue interface {
  SubscribeToQueue(queueName string, channel chan []byte)
}
```

Listing 11: Interface contract allow Message Queues clients.

This module manages all the message arrivals deciding to save them appropriately. It can receive one type of message, informing the system of a new/updated or exited presence. Before a message arrival

this module calls for the start of the subscriber, a thread that subscribes to a *Queue* that receives data coming from the *Middleware*.

3.4.2.3 Database

This module maintains the connection to the database implementing all the *SQL* functions for inserting, updating, or deleting data. These *SQL* definitions were all custom made for this software due to the lack of simple, well-implemented open-source *Object Relation Mapping* tools. The most famous one in *Go* is *GORM* (Go-Gorm), but it didn't provide the flexibility and simplicity needed for the system and, as a result, it was decided to implement a custom solution.

3.4.2.4 Message Client

The *Message Client* is a simple module that creates the *Message Queue Client* and subscribes to the designated *Queue*. After the initialization of the client, sends the received messages to a channel for the *Manager* to handle.

3.4.2.5 API

This module provides endpoints from which the User Interface can make requests and obtain data to show users and, to manage rooms and sensors. This is accomplished by a *REST HTTP* interface that provides access to available resources in the service. This API only provides basic needs for the service to demonstrate it's potential and doesn't implement all RESTful (REST) actions. The resources, actions and JSON specification are described in the next paragraphs.

The **Room** resource is in charge of creating and listing all rooms available. It's actions are:

- GET /rooms
- POST /rooms
- GET /rooms/:id/status
- GET /rooms/:id/historical

For the *POST* action, the entry data is:

- name - string

For the *GET* action, the outgoing data format is:

- id - number
- name - string

The **Sensor** resource handles the registration of new sensors in the system. The only action is:

- POST /sensors

The incoming data must obey the following specification:

- roomName - string
- name - string

3.4.3 User Interface

To show the use of this service it was developed a simple *Single Page Application Web-app* using the *React-JS* framework (ReactJs). This interface provides the current state of rooms, the possibility of adding rooms and sensors and, historical data on the state of a room. The purpose of this prototype is to show the potential and not a fully fledged service capable of authenticating users, having a back-office and a very complex interface. The choice of a *SPA* was easy due to the request/response environment provided by the service (Solovei et al., 2018). Every action of the API is available in the home page and with some modals. In the next set of figures are screenshots of the *UI*.

In Figure 22 is the component that shows the current occupation.

Figure 22: See current occupation

In Figure 23 is the chart that shows the historical data, the "Delta" represents the data time, this day, this week, or this month.

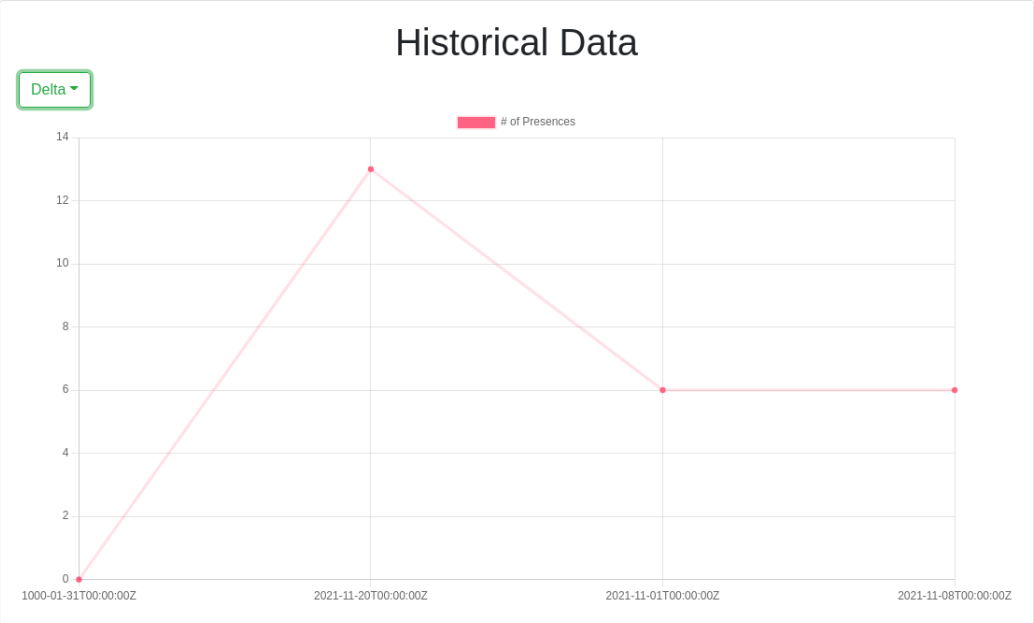


Figure 23: Historical presences data last month

In Figure 24 is the modal component containing a form to add a new sensor to the system.

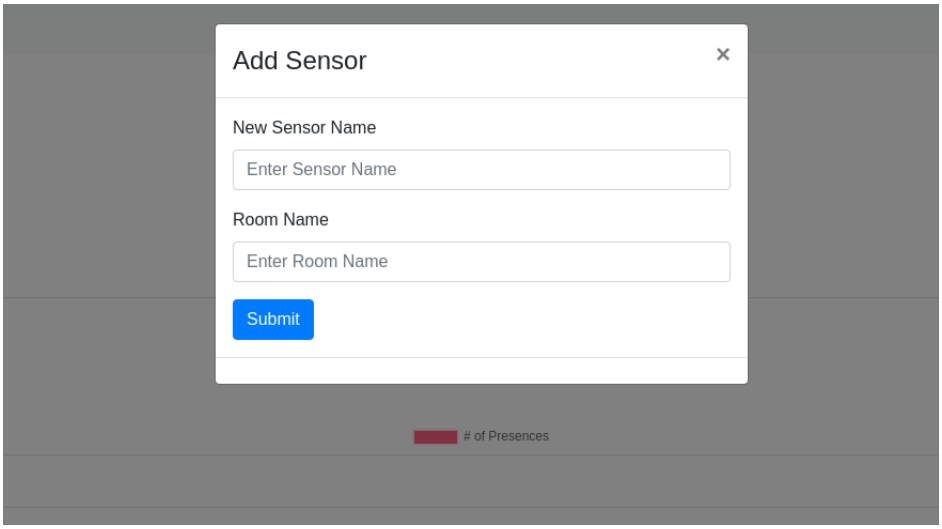


Figure 24: Modal to add a Sensor to the system.

In Figure 25 is the modal component containing a form to add a new room to the system.

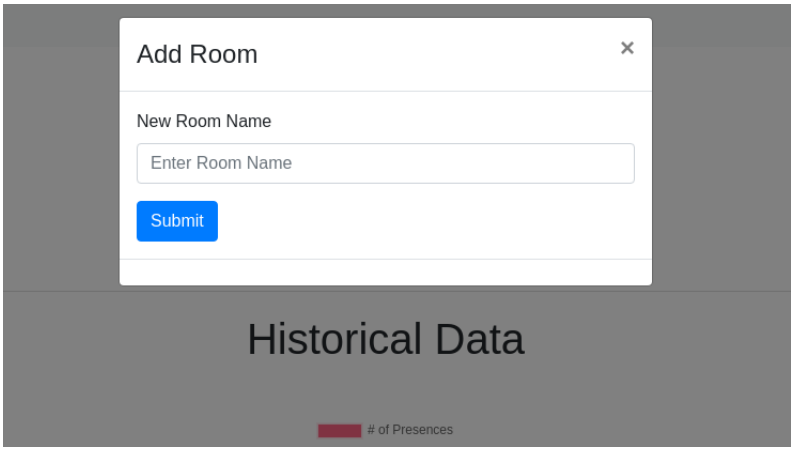


Figure 25: Modal to add a Room to the system.

To enrich the users experience and information other charts like the one on Figure 23 could be applied ML forecast algorithms. This would take advantage of previous historic data and try to predict the future room states (Fernandes et al., 2020b, 2019).

3.4.4 Machine Learning Service interactions

The *Machine Learning Service* gives its estimation on how many people are inside a room given a certain number of probe requests in an interval of time.

This Service exposes an API with one endpoint, a *GET* endpoint. When a request is made, it responds with an estimation made by a ML model (further details on its implementation in the Results chapter).

In Figure 26 is the diagram representing how the interaction takes place.

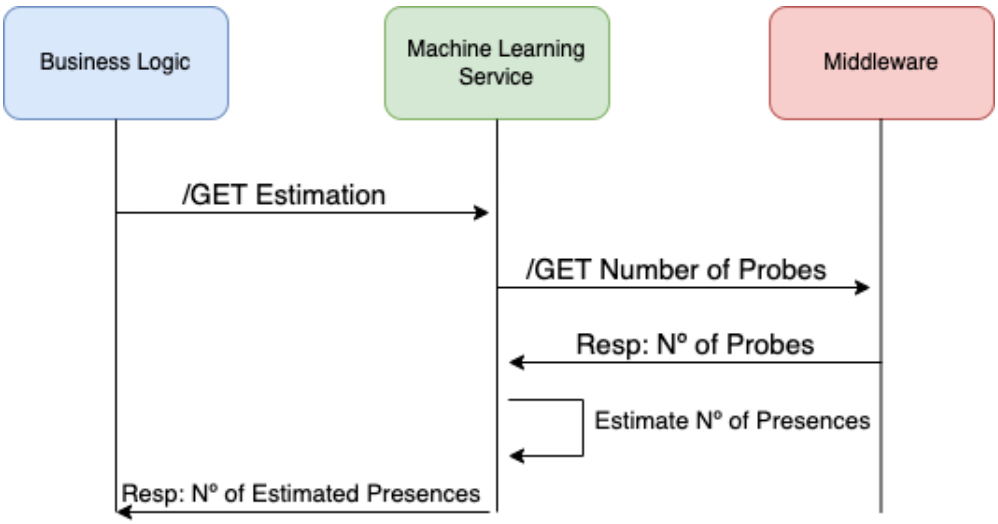


Figure 26: Estimation request flow diagram.

After the response arrives, the *Business Logic* can calculate if the current estimation is too different to the ML estimation and adjust its value to try to be more accurate.

3.5 Communication Patterns

This section will address the decisions behind the communication patterns and protocols in the different aspects of the system. The first one is on Sensor to Cloud communication and, the second is Service to Service communication within the Cloud.

3.5.1 Sensor to Cloud communication

Sensor to Cloud communication is an important factor when designing a cloud-based system. This characteristic allows us to send unprocessed data to the server application for filtering and processing. The IoT field already has a few agreed protocols, such as, AMQP, *CoAP* and MQTT.

As reviewed in the *State of the Art* chapter AMQP and *CoAP* offer similar features both, give the ability to communicate through a request/response interface as well as a publish/subscribe system. When we compare both, AMQP is slower at sending messages. MQTT also is a valid communication standard enabling the construction of a publish/subscribe system. This feature is the only supported feature but allows various types of *QoS* levels (0,1,2).

Because the system at hand has a seemingly low level of different types of messages traded, *CoAP* or AMQP seemed like protocols that would have harmed the construction of the service adding unnecessary overhead. On the other hand, MQTT offered a simple way of rapidly setting up a message trading environment mainly sensor to cloud as the system requires so, the choice fell on MQTT to be the standard communication protocol present in the system. The Event-driven architecture present in the *Middleware* component also greatly influenced this choice, given that in this type of architecture, message queues are the main form of module communication.

To set up an MQTT protocol in the service, there is the need for a broker to manage channels, as well as the messages sent through them. Ideally, this broker is hosted on the cloud for ease of use. Other important features that the chosen broker should have are:

- Authentication process - to guarantee that the intervening parties are trusted;
- *TLS/SSL* - to maintain a secure connection;
- Multiple *QoS* - to have the desired degree of certainty that the sent message reaches the destination;
- Online interface - to allow for an easier experience when setting up the service and debugging;
- Free - as this is an academic project, there isn't any funding to spend on resources.

In Table 2 is an overview of the research conducted online of different available brokers and their features, these aren't the only options but it shows the different existent types.

After reviewing the different options to intermediate the system's communication, *Flespi* appeared like a good option and, after experimenting with its interface, it was clear that it had the necessary features for usage in the system.

Broker	Authentication	TLS/SSL	QoS 0,1,2	Cloud Interface	Free
Mosquitto	Yes	Not standard	All	No	Yes
Hive	Yes	Yes	All	Yes	Yes (100 devices max)
Flespi	Yes	Yes	All	Yes	Yes
Azure	Yes	Yes	All	Yes	No
AWS	Yes	Yes	All	Yes	No

Table 2: Different available MQTT brokers.

One clever way *Flespi* stands out is the authentication system. It uses authentication tokens to guarantee the devices in the system are trusted. This is accomplished by having a *Platform API* that allows the user of the service to perform requests whenever it needs to add a device or revalidate credentials.

For this case study, every sensor publishes messages that contain information about the environment in the same channel. The fact that only one channel exists makes channel management and message reading straightforward in the *Middleware*. Regarding the authentication tokens, the *Middleware* publishes it in a private channel from which only the recipient subscribes, avoiding the need for other devices to read it, maintaining a level of secrecy. In Figure 27 is a diagram representing the described channels and message flow.

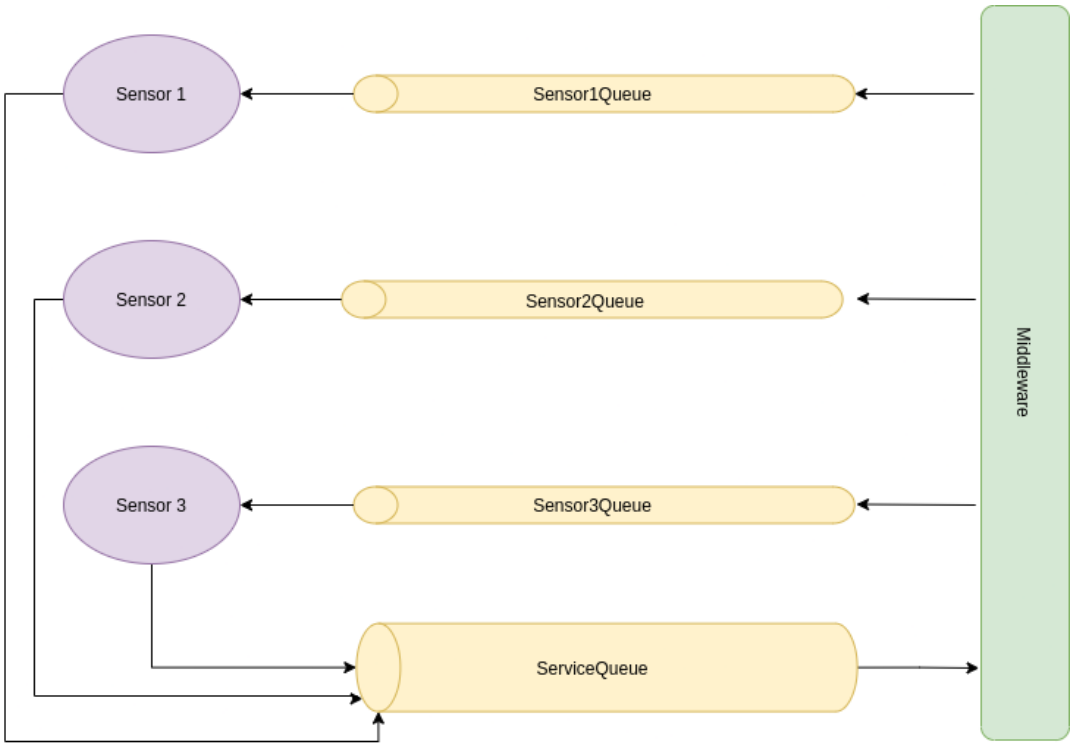


Figure 27: Channels and message flow diagram.

3.5.2 Secure IoT communication

To ensure that all messages come from trusted devices, the system needs to implement some form of communication. After reviewing the options in the State of the Art section,token-based authentication on the MQTT broker appeared to be a simple but effective solution to the problem.

The chosen broker (*Flespi*), already implements an API to fetch tokens from the authentication server present in their service. This made the implementation of a secure system less difficult.

When a device is added to the system, is made a *GET* request to the *Flespi* server requesting a new token. This token is subsequently used in every messaged that is published. In Figure 28 is the authentication token flow.

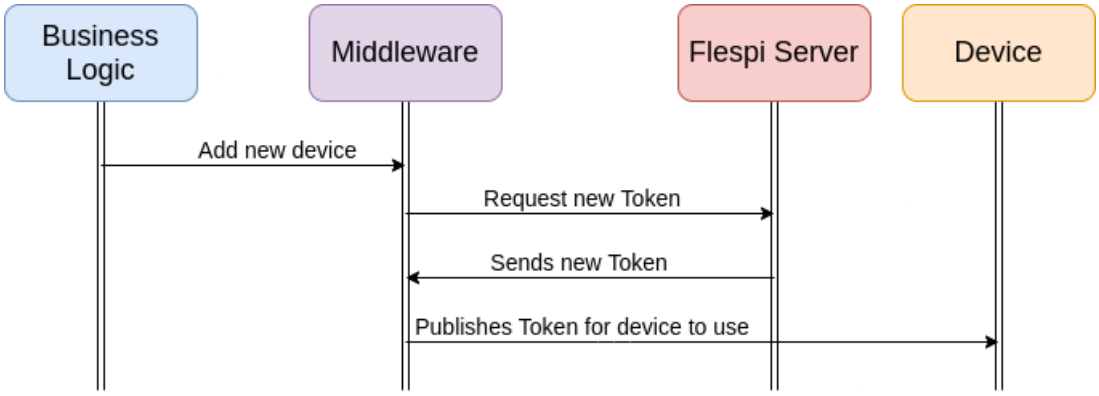


Figure 28: Token request flow.

3.5.3 Service to Service communication

Service to Service communication, in this system, consists of sending messages from one Service to another. As the design of the different systems are mainly event driven, it makes sense to use message queues to trade the majority of messages. Message queues are nothing more that publish/subscribe systems optimized to run in the cloud and offer different types of features such as elasticity and scalability (El Rheddane et al., 2014).

To build a message queuing communication system based in the cloud two platforms choices stand out. Kafka, a distributed open-source message streaming platform and AMQP based RabbitMQ, also an open-source project that acts as a message broker.

Kafka supports a point-to-point architecture as well as pub-sub. Its internals is composed of Producers, Topics, Partitions, and Consumers, offering performance over reliability and high throughput as well as low latency (John and Liu, 2017), in Figure 29 is an overview of its architecture.

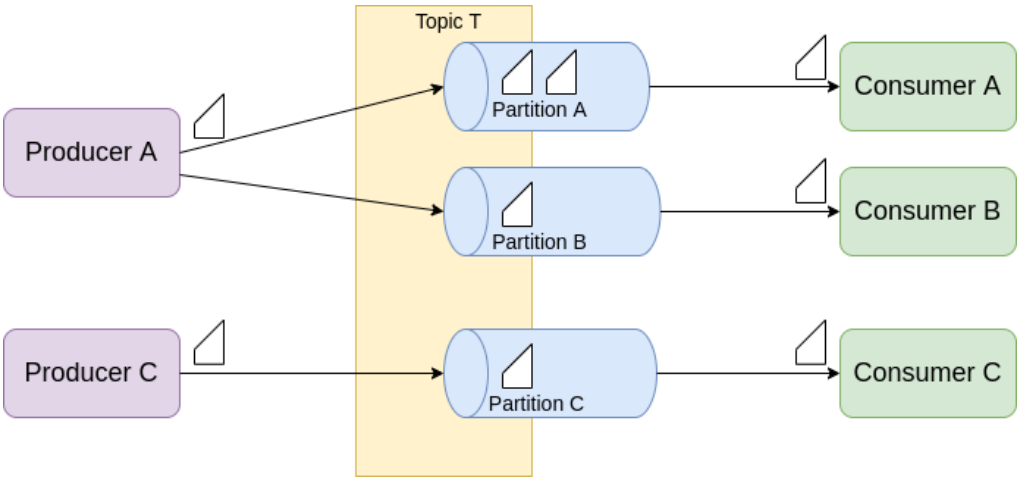


Figure 29: *Kafka* architecture diagram. Obtained from John and Liu (2017).

RabbitMQ, as previously stated, is based on AMQP. It aims to be the standard when it comes to asynchronously message queuing by offering high standards of scalability, reliability, and manageability (John and Liu, 2017). In Figure 30 is a breakdown of *RabbitMQ*'s architecture.

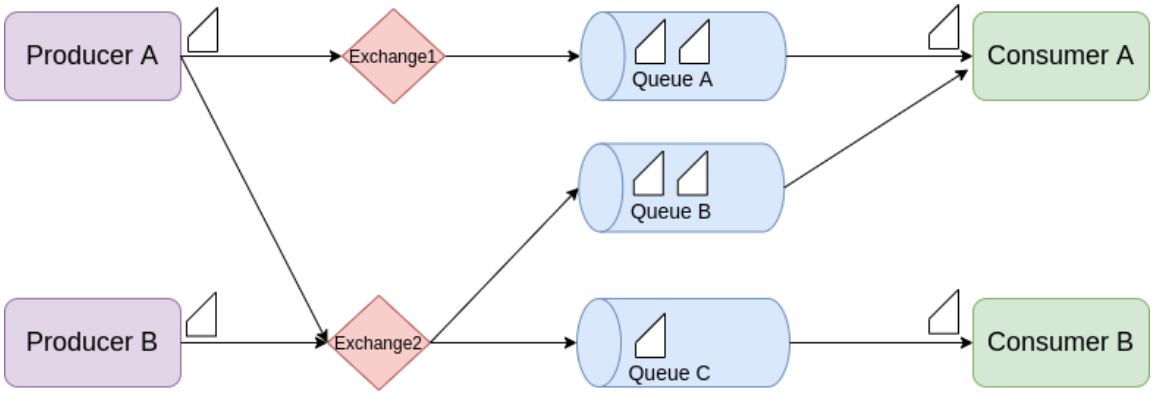


Figure 30: *RabbitMQ* architecture diagram. Obtained from John and Liu (2017).

Given the gathered information, as the system needs to be easily set up reliable, *RabbitMQ* was the choice to build the communication system. A *Docker Compose* file was assembled to setup a local image of a *RabbitMQ* broker to serve our system.

At this moment, there is only one replica of each Service (*Middleware* and *Business Logic*) so, there only needs to be one Exchange and one Queue. If this system needs to grow in the future and, to manage the addition of each service replicas, the broker configuration will necessitate extension to allow these changes. In Figure 31 is the message flow between the *Middleware* and *Business Logic*.

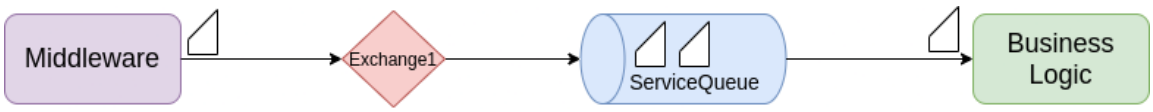


Figure 31: *RabbitMQ* message flow diagram in the system.

Chapter 4

Results and Discussion

This chapter aims to show the obtained results and in what circumstances they were produced, as well as, a discussion the most important metrics of the system and how they were optimized. Another major focus of this chapter is to evaluate the performance of the chosen algorithm and the improvements made to it.

4.1 Introduction

To evaluate the results of the developed system, the best way to do so is to go out and test it in a real-world environment. All the results gathered in these experiments were done in group study rooms where there was people traffic. Firstly, it will be exposed the results that lead to the chosen parameters of the RSSI and *Cleanup Delta* in the cloud (to ascertain if people were passing by or left the room). Then an overview of results and performance of the human detection algorithm already with the best parameters and with the optimization of a *ML* model helping improve the performance.

4.2 Finding the best Cleanup Delta and RSSI

In the presence detection algorithm, the two main variables that are tunable to fit the circumstances in which the system is are the *Cleanup Delta* and RSSI.

4.2.1 RSSI

The RSSI represents the range of capture of the sensor. This value is always between 0 and 100. If it's close to 0, the detected device is closer to the sensor. If it's close to 100, it means that the device is further away from the sensor. This value helps us to determine if the detected device is inside the room we are analyzing.

To determine the best RSSI were performed three experiments inside a room. One experiment had the RSSI threshold unlimited, which means that every WiFi probe detection is considered to be inside the

room. Other with the threshold being 90, meaning that every probe detection with the RSSI over this number will be discarded and, a final one, where the threshold is 80. In the following Figures (32, 33, 34) are charts with the results of the experiment, every two minutes after the system is active, was noted the system estimation and, the actual number of presences in the room. It's also relevant to note that, as the *Cleanup Delta* experiments haven't occurred, the delta was 2 minutes.

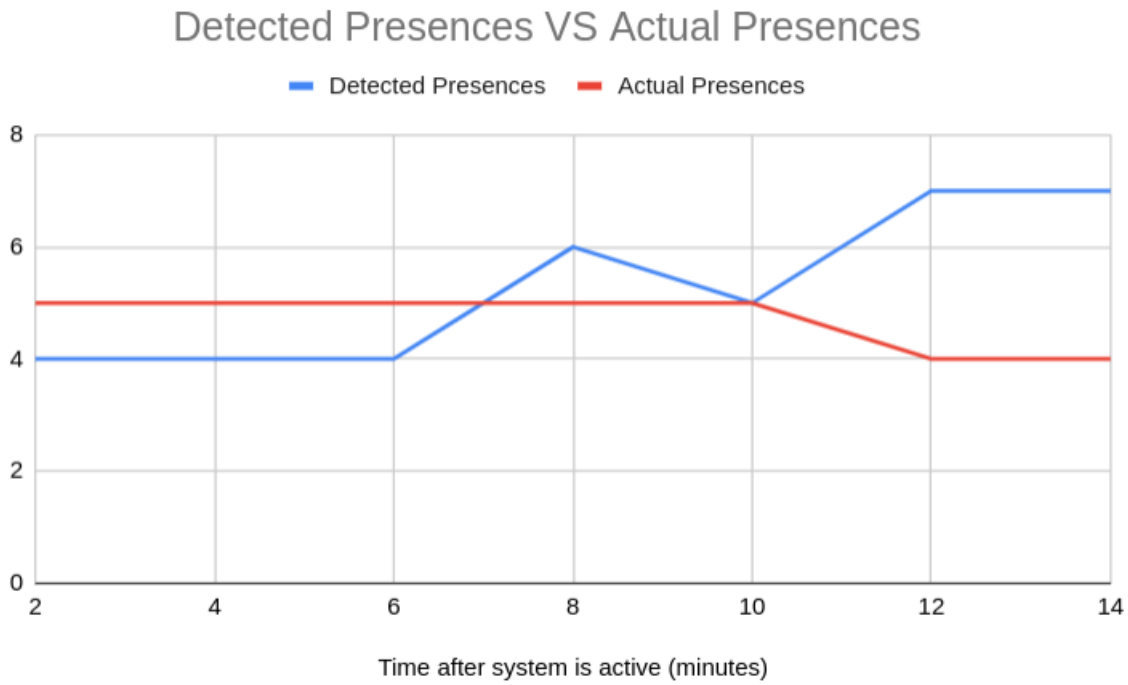


Figure 32: Experiment results with RSSI unlimited.

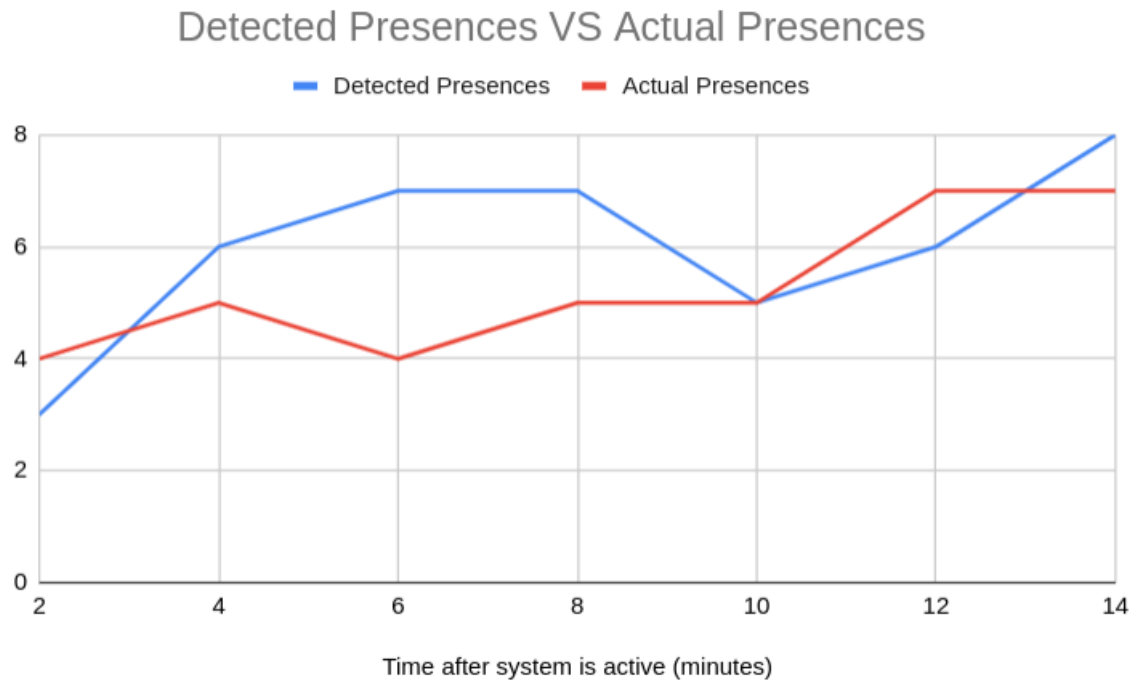


Figure 33: Experiment results with RSSI limit equal to 90.

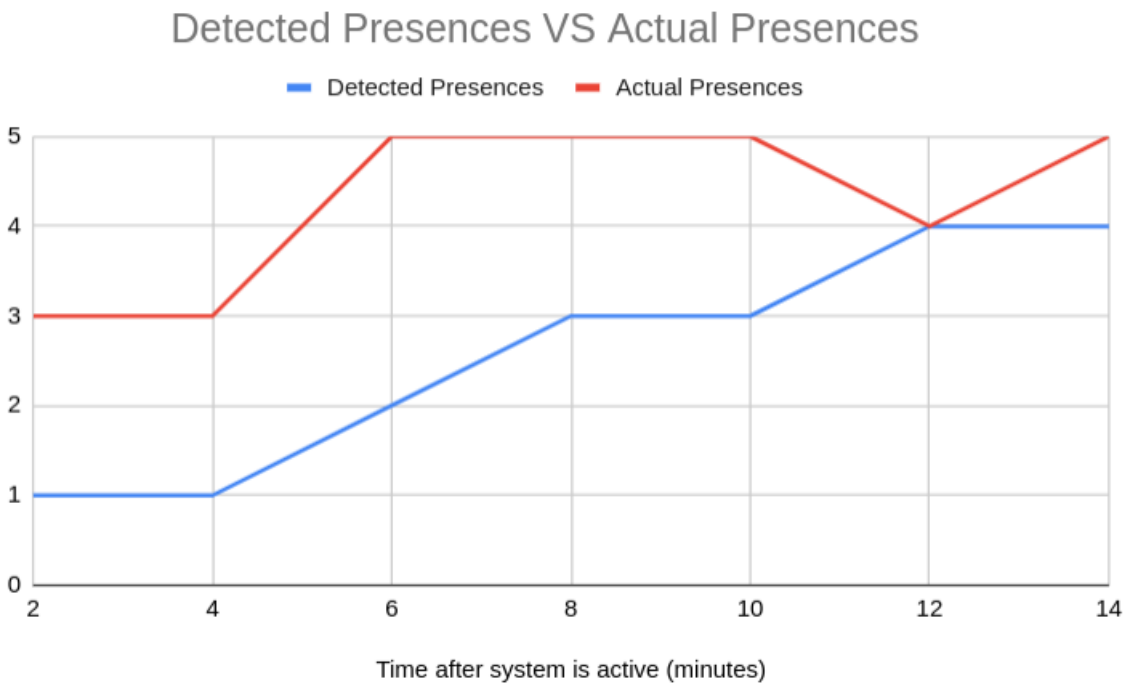


Figure 34: Experiment results with RSSI limit equal to 8 0.

To assess the performance of each RSSI value was utilized the Mean Relative Error. For the experiment without RSSI limit, the *MRE* was 0.33. This result might be due to capturing probe requests from devices outside the room and assuming they are inside, it's possible to see that, in the later stages of this experiment, the algorithm overestimated the number of people inside the room. In the second experiment, with the RSSI being 90, the *MRE* was 0.26. In Figure 33 is possible to see fluctuations above and below

the actual truth. In the last experiment, with the RSSI equal to 80, the *MRE* was 0.42. This result is explainable because the RSSI value over-restricted the area of relevant probe request capture. In Figure 34 can be seen that, almost always, the system was underestimating the number of presences.

With the results of the experiments, it's safe to conclude that the best RSSI value threshold is 90.

4.2.2 Cleanup Delta

The *Cleanup Delta* is the time interval where the system is moving presences from structure to structure to infer if they are inside the room, if they left or, if they never entered.

To get the best *Cleanup Delta* were performed four experiments with different delta times. The deltas were 1 minute, 2 minutes, 3 minutes, and 4 minutes. Given that every 25 seconds, new information arrives from the sensors, at first glance, before the experiment, these values appeared to be in the region of the optimal delta. In Figures 35, 36, 36 and 38 are charts that show the experiment results, every two minutes after the system is active was noted the system estimation and the actual number of presences in the room to reach these values. Also, in this experiment the RSSI threshold was set to 90, the best researched RSSI threshold.

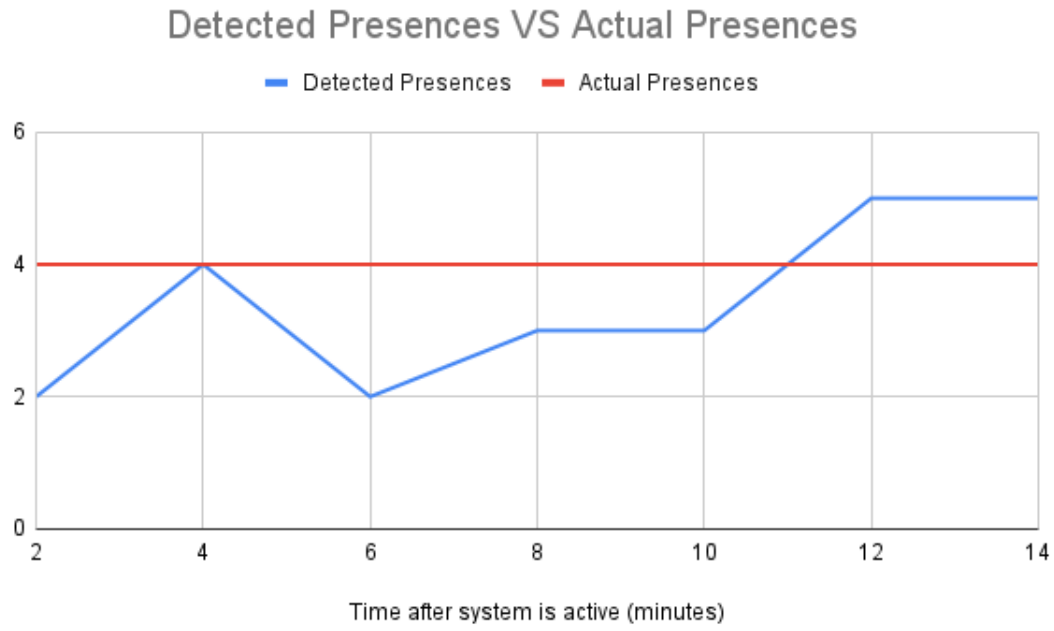


Figure 35: Experiment results with delta equal to 1 minute.

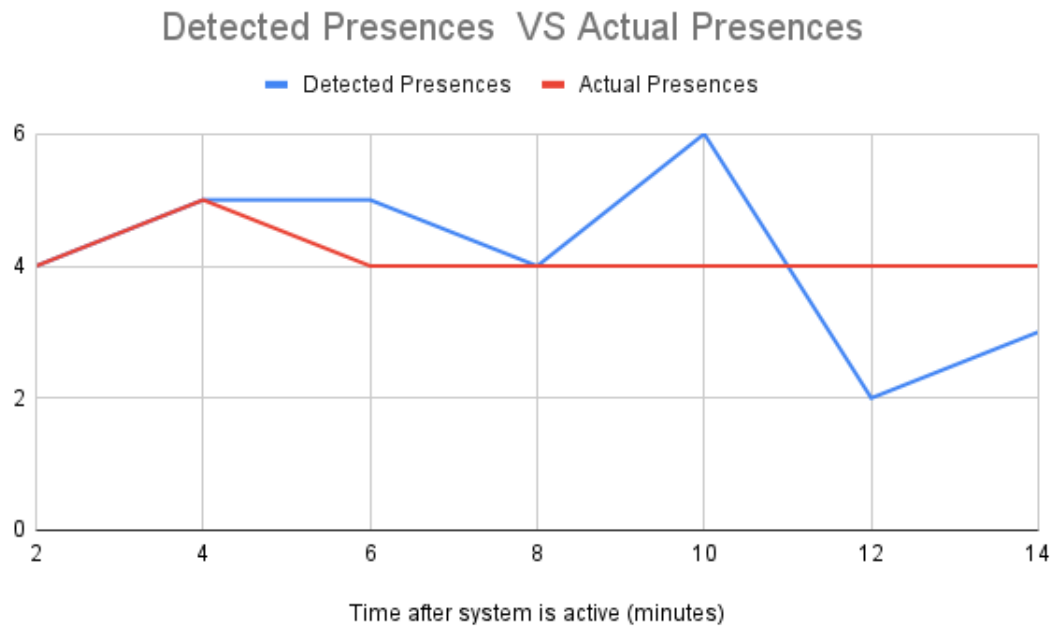


Figure 36: Experiment results with delta equal to 2 minutes.

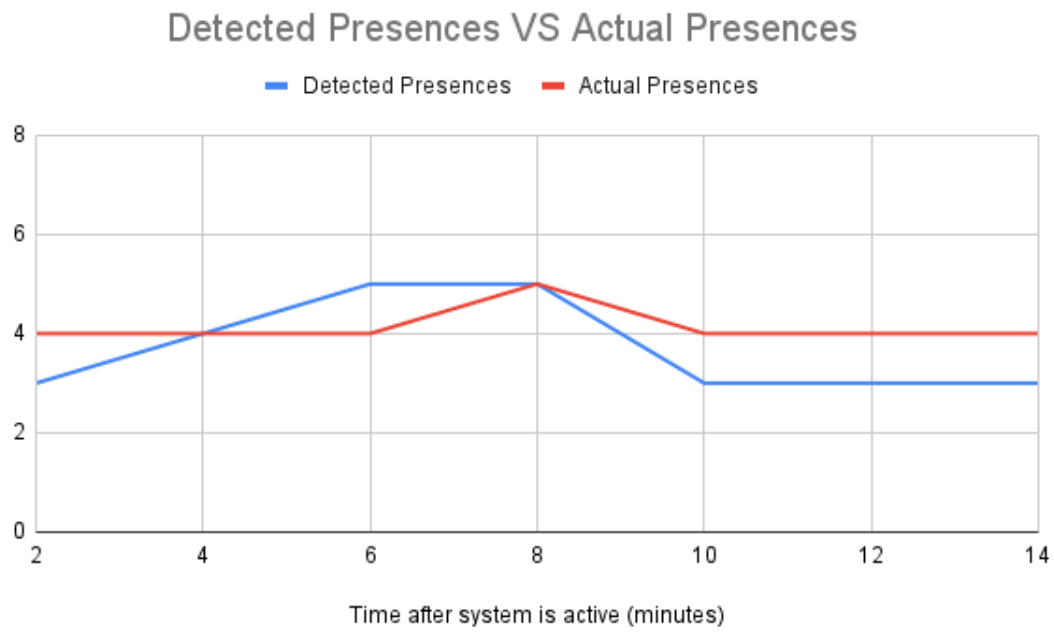


Figure 37: Experiment results with delta equal to 3 minutes.

To determine the best delta time to utilize was used the Mean Relative Error formula, the experiment with the lowest error should be the best performer in the real world.

With the delta being equal to 1 minute, the relative error is *0.28*, this is not a bad performance but, it is lower than expected when compared to the results of other experiments. In the second experiment, with the delta being equal to 2 minute, there is a significant improvement on performance dropping the

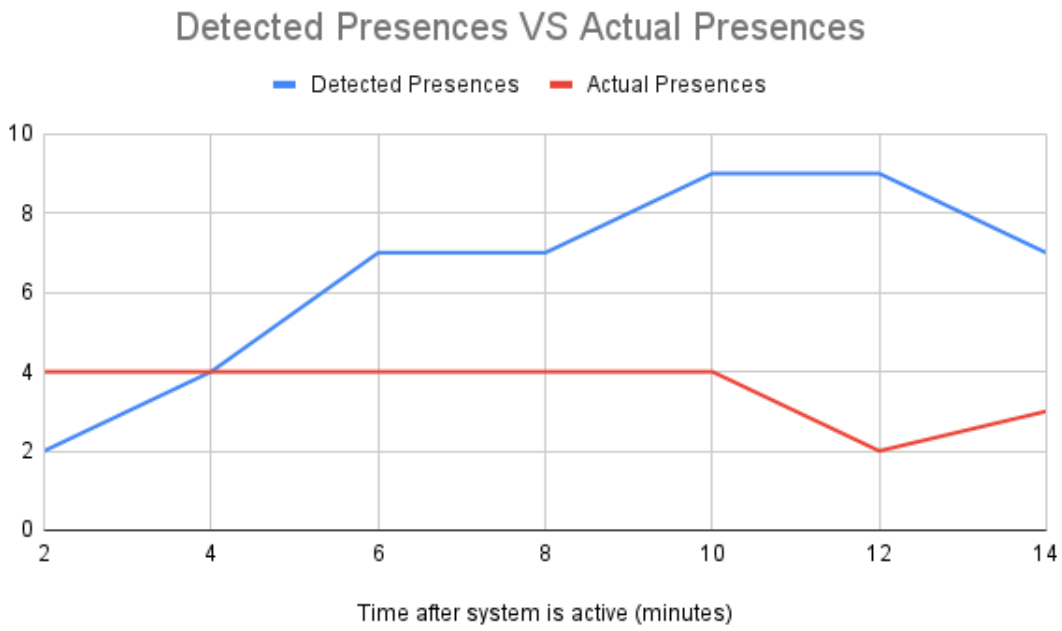


Figure 38: Experiment results with delta equal to 4 minutes.

relative error to *0.21*. In the third experiment, is reached peak performance, with the delta being equal to 3 minute the relative error is only *0.17*. When the delta was set to 4 minutes, the performance of the system was evidently bad, with a *MRE* over 1.

4.3 Overall system's performance

In this section will be evaluated the overall performance with and without a ML improvements.

4.3.1 Performance without Machine Learning Improvements

To assess the performance of the system was performed a 1 hour and 15 minutes experiment in a study room where people enter, leave, or pass by. It's a high movement environment with the possibility of several devices per person probing for available networks.

As evaluated before, the RSSI set was 90 and, the *Cleanup Delta* was set to 3 minutes. It's also relevant to mention that every 25 seconds, new data arrives from the sensor.

On the particular day of the experiment, the environment was very busy. The room was almost always close to full capacity, with that being 20 people maximum. Like in the previous experiments, the inferred presences and the real presences were toked every two minutes while the system was active.

In Figure 39 are the results of the experiment.

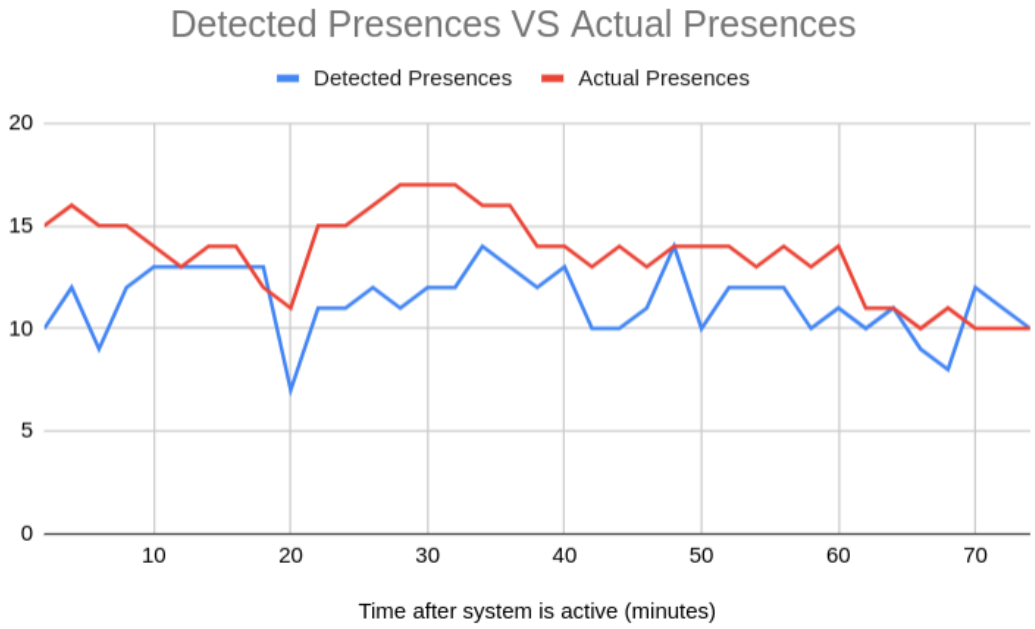


Figure 39: Overall performance experiment.

In this experiment, the *Mean Relative Error* was *0.179*. Overall, the performance was good although it can be seen that, in almost every moment, the system was under-reading the number of presences. That can be explained by the fact the environment was a study room. Many people were focused on reading and writing on paper and not using their smartphones. These days smartphones have mechanisms to save battery when inactive, so they don't probe for networks. This exponential back-off performed by devices when inactive might be the reason why the system is under reading.

4.3.2 Performance with Machine Learning

To maximize the collected data usage was built a *dataset*. This *dataset* contains the captured time of the data, the number of probe requests made in that time, the inferred presences, and the actual presences.

The dataset construction was from the data gathered in the experiment displayed in Figure 39. In Table 3 is a sample of the *dataset*.

Time	Inferred Presences	Real Presences	Probe Requests Number
03:15:00 PM	10	15	46
03:17:00 PM	12	16	28
03:20:00 PM	9	15	51
03:22:00 PM	12	15	63
03:24:00 PM	13	14	47
03:26:00 PM	13	13	42
03:28:00 PM	13	14	38

Table 3: *Dataset* sample.

The previous *dataset* only has 39 entries. Seemingly lack of entries is due to the time needed to perform experiments and data gathering. All data were collected by hand, having no way of automating this.

The ML model applied was a simple one to demonstrate the potentialities of taking advantage of every piece of data collected, the Linear Regression. Since, theoretically, the number of probe requests is correlated to some people, it seems like a good model to implement.

One important column of the *dataset* is the *Time* in which the probe requests took place. If there were more entries, this column was important to correlate the time of day with the busiest hours but, since the sample is low, for the model training it was discarded.

On the following Figures 41 and 40 are some data visualization charts to see if, to the naked eye, can be made conclusion.

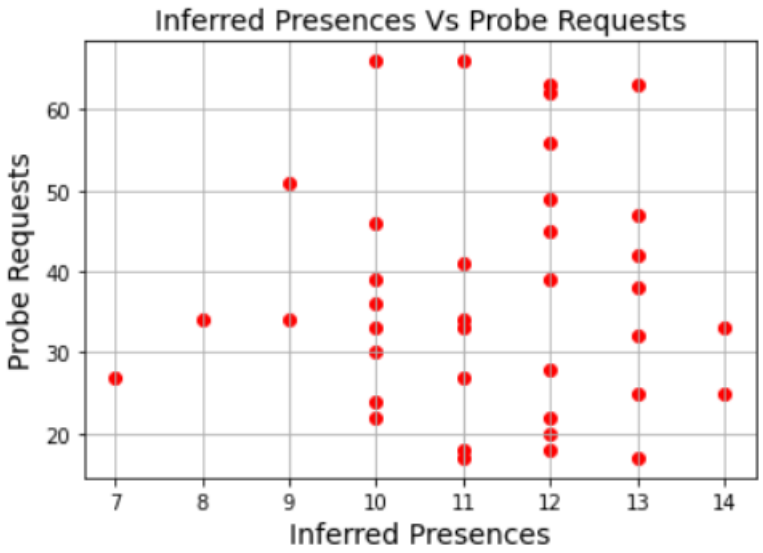


Figure 40: Scatter plot on Inferred Presences and Probe Requests number.

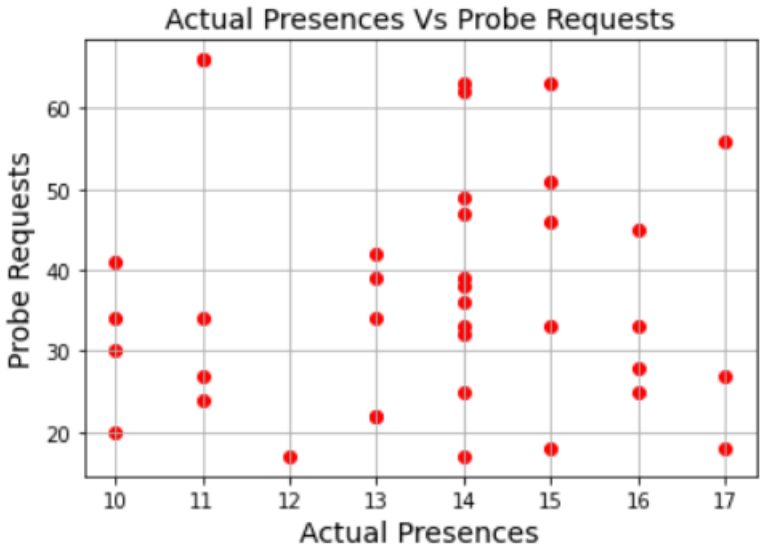


Figure 41: Scatter plot on Actual Presences and Probe Requests number.

From the previous Figures, is hard to make correlations between the number of presences and the number of probe requests. This proves the volatility and indeterministic way that different brands of devices operate.

Nonetheless, Linear Regression was applied. The *dataset* shuffled and split *70/30*, 70% training data and 30% test data. Since the data values don't differ in the order of magnitude, they are normalized.

To apply the regression was used the *sklearn* library *Linear Model* which provides the necessary tools to train and test the model. After all the process, the results were satisfactory. In Table 4 are the results.

Inferred Presences	Actual Presences	Predicted Presences
11	13	13
11	14	13
14	16	15
12	17	14
11	15	13
10	13	13
12	13	14
11	15	13
12	14	14
9	15	12
11	11	13
8	11	11

Table 4: Results after Linear Regression applied.

As we can see in the previous Figure, the results with the ML model improved in a significant way. Using the same metric as before, the Mean Relative Error, this time it was only *0.09*, better than the performance without ML of 0.17.

These results with more data, although the collection is very expansive, could be even better using other ML models. The usage of the *Time* column if we had, for example, a week's worth of data could be

very beneficial to try to predict patterns in the environment, for example, at lunch the room being more empty because are eating outside.

4.4 Summary

In summary, this work fulfilled the expectations of the research hypothesis by integrating a *edge* devices with a *Cloud* application using adequate patterns to assert an architecture and communications between the different layers of the system. By taking advantage of *Event-driven architecture*, it was possible develop a *Middleware* component that consumes data produced by sensors and leverages it to extract useful information, in this case the state of rooms. That leads us into the second part of the hypothesis, that is creating a service that informs people of the state of a room. This objective was accomplished by creating a system that takes *WiFi* probes captured by sensors, applies an algorithm to calculate whether this probe belongs to an unique person and maintains a state of how many people are inside a room at any given time. One important aspect of this work is the use of non-intrusive sensors, meaning that the detection of a human presence is done in a passive manner without explicit interaction between the Physical layer of the system a person.

Finally, the goals of this work were accomplished, by doing an extensive research before starting any kind of development, it was possible to evaluate how to approach the integration of IoT and the *Cloud* on *saas* systems. That meant understanding in a deep manner how the different communication patterns are applied and lead to a solid implementation of an end to end system. ML played a key role on optimizing the data usage and improving the results and overall experience of the service.

Chapter 5

Conclusions

This dissertation was based around the integration between the *Cloud* and IoT with a *Middleware* component and how architecture could be designed to provide a service. Another important aspect of this work was understanding Crowdsensing to be able to provide a solid proof of concept that adds value to the research performed.

Finally, this chapter is divided into two sections, one focusing on the main conclusions of this work and its results, and, another that goes into what further work and investigation could be made.

5.1 Final Considerations

The main goal of this dissertation was to understand how to design a minimal intrusive IoT system that could be integrated with a service provided by the *Cloud*. This meant implementing a proof of concept, an end-to-end system that inferred how many people were inside a room at any given time.

The motivation behind this project came from the challenge that currently there is on integrating physical devices, like sensors, for example, with *Cloud* services. There aren't any generic *Middlewares* that are capable to integrate different kinds of devices available to the community.

The implementation of the proof of concept proved to be important in the realm of this work because allowed for two things. The first one tests all the studied concepts regarding communication between sensors and the *Cloud*, different types of architectures, and other approaches. The second one expands the spectrum of this dissertation by adding the *Crowdsensing* field of study.

Regarding the integration of IoT with the *Cloud* and building an end-to-end service the conclusions are clear. The requirements need to be well defined to be clear on the choices that are needed to be made. If the system requires constant data streaming, with an asynchronous generation of events from the Physical layer the design of the system leans towards an Event-driven. Communication patterns like *mqtt* or other types of message, queues are key to the success of these types of systems. When the system needs to act in a request-response environment, a Service-Oriented approach is a way to go. The implementation of REST API's are the main building block for this approach.

In the SaaS side of this work, besides the architectures mentioned before, the internal design patterns

implemented in the *Business Logic* were very important, especially the use of *Dependency Inversion*. That allowed us to have a well-defined internal structure without the domain logic being dependent on other modules or infrastructure. Another very important aspect was the implementation of tests during the development in the *Middleware* layer, this part of the system was very delicate and, there was a need to make sure the behavior was still the same when different iterations of decision algorithms were experimented.

One area that shows room for improvement is the security of the devices and how we authenticate them and their messages in the *Cloud*. There are some approaches, for example, the use of tokens, like it was implemented in this project, but the research is still very much conceptual and difficult to implement in more simple scenarios like the one we had at hand.

Regarding the topic of *Crowdsensing*, the implementation of the proof of concept was successful. The level of accuracy was good enough to have an adequate perspective of what was happening in the room and how many people were inside when the system was running. The result of 82% accuracy without using ML optimizations was lower than expected when compared to the research performed, but at the same time not too low. With the addition of a ML algorithm, even though it was a simple one, the performance increased to 91%, meaning that it was the best accuracy seen when compared to the research. This leads to conclude that the use of ML optimization algorithms is very powerful and should be considered when conducting other related projects.

5.2 Future Work

The research and proof concept presented in this work already presents an insight into the realm of IoT and its integration with the *Cloud* to design service, as well as a better algorithm to do indoors *Crowdsensing*. Nonetheless, there are still improvements that can be made in both aspects.

In terms of delivering a service to consumers, some aspects were purposefully ignored, for example, the implementation of a user authentication system to restrict who could access the information about the state of the room. This research and implementation would be interesting because in an environment like the one utilized, a public study room, this information can be accessed by everyone but, if we switch to a similar environment like a company office, we probably don't want that information publicly available.

Another aspect, already mentioned before, is the security integration between devices and the *Cloud*, which still has a lot of room for improvement. More real-world applicable research is needed regarding this topic.

In terms of optimizing the use of the data gathered, more ML algorithms can be applied. Currently, only the real-time results are being optimized with these models but more could be done in the future. For example, predicting the future states of the rooms. Since we have historical data we could provide a system that forecasts how busy a room might be at any given moment of the future.

With all this being said, real-world deployment of the system would be a great test of the system, seeing how it performs during extended periods, with peak times both in the Physical and the Business layers. This means that the designed deployment needs to take into account non-functional requirements

such as elasticity and scalability.

Bibliography

B. Abdualgalil and S. Abraham. Applications of machine learning algorithms and performance comparison: A review. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, pages 1–6, 2020. doi: 10.1109/ic-ETITE47903.2020.490.

O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018. ISSN 2405-8440. doi: https://doi.org/10.1016/j.heliyon.2018.e00938. URL http://www.sciencedirect.com/science/article/pii/S2405844018332067.

O. Al-Debagy and P. Martinek. A comparative review of microservices and monolithic architectures. 05 2019.

AMQP. Advanced messaging queuing protocol. URL https://www.amqp.org/about/what. (Accessed Jan. 15, 2021).

R. Asir, H. Manohar, W. Anandraj, and K. Sivaranjani. Iot as a service. *International Conference on Innovations in information, Embedded and Communication Systems (ICIIECS)*, 03 2016.

H. F. Atlam, A. Alenezi, A. Alharthi, R. J. Walters, and G. B. Wills. Integration of cloud computing with internet of things: Challenges and open issues. *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2017. doi: 10.1109/ithings-greencom-cpscom-smartdata.2017.105.

R. Atmoko, R. Riantini, and M. Hasin. Iot real time data acquisition using mqtt protocol. *Journal of Physics: Conference Series*, 853:012003, 05 2017. doi: 10.1088/1742-6596/853/1/012003.

M. Bamiah and S. Brohi. Exploring the cloud deployment and service delivery models. *International Journal of Research and Reviews in Information Sciences*, 3:2046–6439, 02 2011.

H. Bansal and K. Sharma. A review study on various algorithms of machine learning. *International Journal of Emerging Technologies and Innovative Research (www.jetir.org)*, 7(4):1065–1070, Apr 2020.

A. Bhawiyuga, M. Data, and A. Warda. Architectural design of token based authentication of mqtt protocol in constrained iot device. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, pages 1–4, 2017. doi: 10.1109/TSSA.2017.8272933.

C. Bormann. Constrained application protocol. URL https://coap.technology/. (Accessed Jun. 25, 2021).

C. Bormann, A. P. Castellani, and Z. Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012. doi: 10.1109/MIC.2012.29.

Brunofmf. brunofmf/crowd-sensing - github. URL https://github.com/brunofmf/Crowd-Sensing. (Accessed Dec. 17, 2020).

P. Chapman, J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. R. Shearer, and R. Wirth. Crisp-dm 1.0: Step-by-step data mining guide. 2000.

N. R. G. Charan, S. T. Rao, and D. P. Srinivas. Deploying an application on the cloud. *International Journal of Advanced Computer Science and Applications*, 2(5), 2011. doi: 10.14569/IJACSA.2011.020520. URL http://dx.doi.org/10.14569/IJACSA.2011.020520.

M. Dammak, O. R. M. Boudia, M. A. Messous, S. M. Senouci, and C. Gransart. Token-based lightweight authentication to secure iot networks. In *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 1–4, 2019. doi: 10.1109/CCNC.2019.8651825.

L. De Lauretis. From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96, 2019. doi: 10.1109/ISSREW.2019.00050.

P. Desai, A. Sheth, and P. Anantharam. Semantic gateway as a service architecture for iot interoperability. 10 2014. doi: 10.1109/MobServ.2015.51.

M. El-hajj, A. Fadlallah, M. Chamoun, and A. Serhrouchni. A survey of internet of things (iot) authentication schemes. *Sensors*, 19(5), 2019. ISSN 1424-8220. doi: 10.3390/s19051141. URL https://www.mdpi.com/1424-8220/19/5/1141.

A. El Hakim. Internet of things (iot) system architecture and technologies, white paper. 03 2018. doi: 10.13140/RG.2.2.17046.19521.

A. El Rheddane, N. De Palma, A. Tchana, and D. Hagimont. Elastic message queues. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 17–23, 2014. doi: 10.1109/CLOUD.2014.13.

B. Fernandes, F. Silva, C. Analide, and J. Neves. Crowd sensing for urban security in smart cities. *Journal of Universal Computer Science*, 24:302–321, 06 2018.

B. Fernandes, F. Silva, H. Alaiz Moreton, P. Novais, C. Analide, and J. Neves. *Traffic Flow Forecasting on Data-Scarce Environments Using ARIMA and LSTM Networks*, pages 273–282. 04 2019. ISBN 978-3-030-16180-4. doi: 10.1007/978-3-030-16181-1_26.

B. Fernandes, J. Neves, and C. Analide. *SafeCity: A Platform for Safer and Smarter Cities*, pages 412–416. 06 2020a. ISBN 978-3-030-49777-4. doi: 10.1007/978-3-030-49778-1_37.

B. Fernandes, F. Silva, H. Alaiz Moreton, P. Novais, C. Analide, and J. Neves. Long short-term memory networks for traffic flow forecasting: Exploring input variables, time frames and multi-step approaches. *Informatica*, pages 1–27, 01 2020b. doi: 10.15388/20-INFOR431.

G. Firestore. Cloud firestore firebase. URL https://firebase.google.com/docs/firestore/. (Accessed Jul. 26, 2021).

D. Gigante, P. Oliveira, B. Fernandes, F. Lopes, and P. Novais. Unsupervised learning approach for ph anomaly detection in wastewater treatment plants. In H. Sanjurjo González, I. Pastor López, P. García Bringas, H. Quintián, and E. Corchado, editors, *Hybrid Artificial Intelligent Systems*, pages 588–599, Cham, 2021. Springer International Publishing.

Go-Gorm. Orm library for golang. URL https://gorm.io/. (Accessed Apr. 19, 2021).

Google. Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. URL https://golang.org/. (Accessed Jun. 26, 2021).

T. Gu. Newzoo's global mobile market report: Insights into the world's 3.2 billion smartphone users, the devices they use and the mobile games they play. Sep 2019.

A. Ibrahim and R. A Rashid. Lightweight iot middleware for rapid application development. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 17, 01 2019. doi: 10.12928/telkomnika.v17i3.11793.

I. Ibrahim. Iterative and incremental development analysis study of vocational career information systems. 11:13–24, 2020.

A. Industries. Adafruit mqtt library. URL https://github.com/adafruit/Adafruit_MQTT_Library. (Accessed Jan. 17, 2021).

V. John and X. Liu. A survey of distributed message broker queues. 04 2017.

A. Kafka. Apache kafka. URL https://kafka.apache.org/. (Accessed Jul. 18, 2021).

Kalanda. Kalanda/esp8266-sniffer - github. URL https://github.com/kalanda/esp8266-sniffer. (Accessed Jan. 5, 2021).

M. E. Khalil, K. Ghani, and W. Khalil. Onion architecture: a new approach for xaas (every-thing-as-a service) based virtual collaborations. In *2016 13th Learning and Technology Conference (L T)*, pages 1–7, 2016. doi: 10.1109/LT.2016.7562859.

C. Lai, F. Boi, A. Buschettu, and R. Caboni. Iot and microservice architecture for multimobility in a smart city. pages 238–242, 08 2019. doi: 10.1109/FiCloud.2019.00040.

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 of Things Journal*, PP:1–1, 03 2017. doi: 10.1109/JIOT.2017.2683200.

J. Martin, T. Mayberry, C. Donahue, L. Foppe, L. Brown, C. Riggins, E. Rye, and D. Brown. A study of mac address randomization in mobile devices and when it fails. *Proceedings on Privacy Enhancing Technologies*, 2017, 03 2017. doi: 10.1515/popets-2017-0054.

U. Mehmood, I. Moser, P. Prakash, and A. Banerjee. Occupancy estimation using wifi: A case study for counting passengers on busses. 03 2019. doi: 10.1109/WF-IoT.2019.8767350.

D. Mishra. *Performance Analysis of Deadlock Prevention and MUTEX Detection Algorithms in Distributed Environment*. 01 2018.

MQTT.org. The standard for iot messaging. URL https://mqtt.org/. (Accessed Jan. 12, 2021).

M. Murphy. Cellphones now outnumber the world's population, Jan 2019. URL https://finance.yahoo.com/news/cellphones-now-outnumber-world-population-210800857.html.

N. Naik. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, 2017. doi: 10.1109/SysEng.2017.8088251.

S. Nastic, S. Sehic, D. Le, H. Truong, and S. Dustdar. Provisioning software-defined iot cloud systems. In *2014 International Conference on Future Internet of Things and Cloud*, pages 288–295, 2014. doi: 10.1109/FiCloud.2014.52.

S. Naveen. Study of iot: Understanding iot architecture, applications, issues and challenges. May 2016.

A. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Sheng. Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, PP:1–1, 10 2016. doi: 10.1109/JIOT.2016.2615180.

L. Oliveira, D. Schneider, J. Souza, and W. Shen. Mobile device detection through wifi probe request analysis. *IEEE Access*, PP:1–1, 06 2019. doi: 10.1109/ACCESS.2019.2925406.

P. Oliveira, B. Fernandes, C. Analide, and P. Novais. *Multi-step Ultraviolet Index Forecasting Using Long Short-Term Memory Networks*, pages 187–197. 01 2021. ISBN 978-3-030-53035-8. doi: 10.1007/978-3-030-53036-5_20.

V. S. Padala, K. Gandhi, and P. Dasari. Machine learning: the new language for applications. *IAES International Journal of Artificial Intelligence*, 8:411–421, 2019.

C. Pahl, P. Jamshidi, and O. Zimmermann. Microservices and containers. 03 2020.

G. D. G. PostgreSQL. Postgresql: The world's most advanced open source relational database. URL https://www.postgresql.org/. (Accessed Aug. 12, 2021).

D. Preuveneers and P. Novais. A survey of software engineering best practices for the development of smart applications in ambient intelligence. *Journal of Ambient Intelligence and Smart Environments*, 4:149–162, 08 2012. doi: 10.3233/AIS-2012-0150.

RabbitMQ. Messaging that just works. URL https://www.rabbitmq.com/. (Accessed Jul. 12, 2021).

ReactJs. React – a javascript library for building user interfaces. URL https://reactjs.org/. (Accessed Aug. 1, 2021).

L. Schauer, M. Werner, and P. Marcus. Estimating crowd densities and pedestrian flows using wi-fi and bluetooth. 01 2014. doi: 10.4108/icst.mobiquitous.2014.257870.

P. Sethi and S. Sarangi. Internet of things: Architectures, protocols, and applications. *Journal of Electrical and Computer Engineering*, 2017:1–25, 01 2017. doi: 10.1155/2017/9324035.

V. Solovei, O. Olshevska, and Y. Bortsova. The difference between developing single page application and traditional web application based on mechatronics robot laboratory onaft application. 03 2018. doi: 10.15673/atbp.v10i1.874.

D. Soni and A. Makwana. A survey on mqtt: A protocol of internet of things(iot). 04 2017.

A. Sunyaev. *Middleware*, pages 125–154. Springer International Publishing, Cham, 2020. ISBN 978-3-030-34957-8. doi: 10.1007/978-3-030-34957-8_5. URL https://doi.org/10.1007/978-3-030-34957-8_5.

D. Thangavel, X. Ma, A. Valera, H. Tan, and C. K. Tan. Performance evaluation of mqtt and coap via a common middleware. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6, 2014. doi: 10.1109/ISSNIP.2014.6827678.

Vijayalakshmi. Analysis on software development approaches. 2(12), Dec 2011.

J. Weppner and P. Lukowicz. Bluetooth based collaborative crowd density estimation with mobile phones. In *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 193–200, 2013. doi: 10.1109/PerCom.2013.6526732.

Y. Yuan, C. Qiu, W. Xi, and J. Zhao. Crowd density estimation using wireless sensor networks. pages 138–145, 12 2011. doi: 10.1109/MSN.2011.31.

X. Zhou. Research on wi-fi probe technology based on esp8266. In *Proceedings of the 2017 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering (ICMMCCE 2017)*, pages 163–167. Atlantis Press, 2017. ISBN 978-94-6252-381-4. doi: https://doi.org/10.2991/icmmcce-17.2017.34. URL https://doi.org/10.2991/icmmcce-17.2017.34.