Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

Fault-Tolerant, Scalable and Interoperable IoT Platform

José Marques Barga Soares

Mestrado em Engenharia Informática
Engenharia de Software

Dissertação orientada por:

Professor Doutor José Manuel da Silva Cecílio

2020

Acknowledgements

I would like to express my biggest thanks to my professor and advisor, professor José Cecílio for all the help and work that he has done this year, without his help this thesis would not have been possible. His patience and guidance were decisive, especially in a time that was difficult for everyone due to the pandemic. He helped me through the most difficult time of my academic path, and for that I will always be grateful.

I also want to thank Professor António Casimiro for his advices in the beginning of my thesis and for allowing me of the opportunity to work in the AQUAMON project.

I want to thank my parents and my family for always trying to help me in anyway they could, the strength they gave me was very important in the most difficult moments that I had to go through. Additionally, I would like to thank all my friends who always were there to support me, they know who they are but I would like to indicate some who had a greater impact - Ana, Patricia, Tiago, Maria and Rafael.

This work was supported by FCT through funding of the AQUAMON research project, ref. PTDC/CCI-COM/30142/2017 and LASIGE Research Unit, ref. UIDB/00408/2020.

Abstract

Nowadays the growth of Internet usage is quite visible. Everyday the number of devices connected to the Internet increases, everything may be a smart device capable of interacting with the Internet, from smartphones, smartwatches, refrigerators and much more. All of these devices are called *things* in the Internet of Things. Many of them are usually constrained devices due to it's size, usually very small with low capacities such as memory and/or processing power. These kind of devices need to be very efficient in all of their actives. For example, the battery lifetime should be maximized as possible so that the necessity to change each device's battery could be minimized. There are many technologies that allow communication between devices. Besides the technologies, protocols may be involved in the communication between each device in an IoT system. Communication Protocols define the behaviour that is followed by *things* when communicating with each other. For example, in some protocols acknowledgments must be used to ensure data arrival, while in others this feature is not enforced. There are many communication Protocols available in the literature.

The use of communication protocols and communication models bring many benefits to IoT systems, but they may also benefit from using the cloud. One of the biggest struggles in IoT is the fact that things are very constrained devices in terms of resources (CPU and RAM). With the cloud this would no longer be an issue. Plus, the cloud is able of providing device management, scalability, storage and real time transmission.

The characteristics of the communication protocols were studied and an innovative system architecture based on micro-services, Kubernetes and Kafka is proposed in this thesis. This proposal tries to address issues such as *scalability*, *interoperability*, *fault tolerance*, *resiliency*, *availability* and *simple management* of large IoT systems. Supported by Kubernetes, which is an open-source technology that allows micro-services to be extensible, configurable and automatically managed with fault tolerance and Kafka, which is a distributed event log that uses the publish-subscribe pattern, the proposed architecture is able to deal with high number of devices producing and consuming data at the same time.

The proposed *Fault-Tolerant and Interoperable IoT Architecture* is a cluster composed of many components (micro-services) that were implemented using docker containers. The current implementation of the system supports the MQTT, CoAP and REST protocols for data incoming and the same plus websockets for data output.

Since the system is based on micro-services, more protocols may be added in a simple way (just a new micro-service must be added). The system is able to convert any protocol into another protocol, e.g., if a message arrives at the system through MQTT protocol, it can be consumed using the CoAP or REST protocol. When messages are sent to the system the payload is stored in Kafka independently of the protocol, and when clients request it, it is consumed from Kafka and encapsulated by the client protocol to be sent to the client.

In order to evaluate and demonstrate the capabilities of our proposal a set of experiments were made, which allows to collect information about the performance of the Communication Protocols, the system as a whole, Kubernetes and Kafka. From the experiments we were able to conclude that the message size is not so much important, since the system is able to deal with messages from 39 bytes to 2000 bytes. Since we are designing the system for IoT applications, we considered that messages with 2000 Bytes are big messages.

Also, it was recognized that the system is able to recover from crashed nodes and to respond well in terms of average delay and packet loss when low and high throughput are compared. In this situation, there is a significant impact of the RAM usage, but the system still works without problems.

In terms of scalability, the evaluation of the system through its cluster under-layer platform (Kubernetes) allowed us to understand that there is no direct relation between the time spent to

add a node and the number of nodes in the cluster. So the time to add a single node is reduced and constant. However, the same conclusion is not true for the number of instances that are needed at high layer (application layer). Here, time spent to increase the number of instances of a specific application is directly proportional to the number of instances that are already running.

In respect to data redundancy and persistence, the experiments showed that the average delay and packet loss of a message sent from a Producer to a Receiver is approximately the same regardless of the number of Kafka instances being used. Additionally, using a high number of partitions has a negative impact on the system's behaviour.

Keywords: IoT, MQTT, CoAP, REST, Kubernetes, Kafka, Cloud, Scalability, Interoperability, Fault Tolerance, Resiliency, Availability, Simple Management, Publish-Subscribe, Micro-Services, Docker containers.

Resumo alargado em Português

Hoje em dia o crescimento da utilização da internet está a aumentar devido à vasta possibilidade de dispositivos capazes de se conectarem a ela, desde telefones, relógios, frigoríficos ou lâmpadas inteligentes, entre muitos outros dispositivos. Todos estes dispositivos são denominados de "coisas" na *Internet das Coisas ("Internet of Things - IoT" em inglês)*. Muitos deles apresentam pouco poder de processamento e/ou memória, devido ao fato da grande maioria ser de pequenas dimensões. Normalmente estes dispositivos são usados para capturar determinados dados, tipicamente com tamanhos reduzidos, e transmiti-los para outras entidades, como é o caso de sensores de temperatura e/ou humidade que transmitem a sua informação para uma central onde são apresentados.

Para que os sistemas IoT tenham um bom desempenho estes devem ser o mais eficientes possível, por exemplo, o tempo de vida da bateria dos dispositivos usados deve ser maximizada de forma a que estas tenham que ser substituídas o menor número de vezes possível (por exemplo, uma vez por ano).

Existem vários aspetos que devem ser tidos em consideração na maximização da eficiência dos sistemas IoT, como é o caso do tipo de **tecnologia** usada, o **modelo** ou o **protocolo de comunicação**. Muitos dos sistemas IoT são suportados por modelos de computação na nuvem. Neste caso a comunicação com a nuvem também deve ser tida em consideração.

Nesta tese, numa primeira fase, estes aspetos foram estudados de forma a poder ter uma boa base de conhecimento sobre como maximizar a eficiência deste tipo de sistemas.

As tecnologias estudas foram as seguintes: *LoRa*, *Sigfox*, *NB-IoT*, *Bluetooth*, *ZigBee*, *Wi-Fi*, *LTE*, *GSM*.

Em termos de modelos de comunicação, nesta tese foram abordados os modelos *publish-subscribe* e *request-response*. O modelo *publish-subscribe* é um modelo onde dispositivos publicam dados (*Publishers*) para um componente central chamado *Broker*, cuja responsabilidade é receber os dados e reencaminhá-los para o destinatário correto (*Subscribers*). Os dados que fluem nesta comunicação são identificados por meio de *Tópicos*. Por exemplo, os dados relativos a temperatura e humidade podem ser divididos em dois tópicos chamados "temperatura" e "humidade". Os *Publishers* publicam dados num tópico (no *Broker*) e o *Broker* trata de reencaminhar esses dados a todos os *Subscribers* que tenham, previamente, subscrito esse tópico. No caso do modelo *request-response* não existem tópicos, nem *Publishers*, *Subscribers* ou *Brokers*. Em vez disso existem *Clientes*, *Servidores* e *Recursos*. Os dados transmitidos são separados em recursos que representam a informação enviada, por exemplo "temperatura" ou "humidade". Os *Recursos* são pré-programados nos servidores de modo a que estes saibam como proceder para responder aos clientes. A interação com os servidores deste tipo baseia-se no conjunto de primitivas CRUD do HTML (*PUT*, *POST*, *GET* ou *DELETE*). Este conjunto de primitivas permite enviar dados a um determinado *Recurso* criando-o caso não exista, atualizar um *Recurso* existente, pedir ao servidor para receber os dados de um *Recurso* ou apagar dados de um *Recurso*.

Em termos de protocolos de comunicação, nesta tese são abordados os protocolos *MQTT*, *CoAP*, *LoRaWAN*, *AMQP*, *REST* e *OPC-UA*. Dada a amplitude do trabalho, a parte experimental teve de ser limitada, concentrando a sua avaliação nos protocolos MQTT, CoAP e REST.

O protocolo MQTT é um dos protocolos de acesso público mais usados em IoT. Este protocolo usa o modelo *publish-subscribe* para estabelecer comunicação o que significa que os dispositivos não precisam de saber os endereços uns dos outros porque os dados são enviados para o broker, sendo o único endereço necessário para estabelecer a comunicação.

Em relação à comunicação, este protocolo oferece garantias de entrega da informação, uma vez que usa o protocolo TCP na camada de transporte. Quanto à qualidade de serviço, o protocolo oferece três níveis: "No máximo uma vez", "Pelo menos uma vez" e "Exatamente uma vez". No

primeiro nível ("No máximo uma vez") as mensagens não chegam ao destino mais que uma vez e caso alguma se perca nada é feito. No segundo nível as mensagens que não cheguem ao destino são reenviadas, podendo haver mensagens duplicadas. Para isto o protocolo MQTT faz uso de "acknowledgments" que são mensagens cujo único propósito é confirmar a receção de chegada de uma mensagem à entidade que a enviou. No último nível, as mensagens são entregues uma única vez, sem perdas e sem duplicados, para isso o protocolo usa quatro etapas de confirmação.

No que diz respeito ao protocolo CoAP, este foi desenhado especificamente para ser usado em ambientes onde os dispositivos têm poucos recursos, o que o torna também muito útil em sistemas IoT, porém, ao contrário de MQTT, este segue o modelo *resquest-response*. É também um protocolo de acesso público, mas neste caso pode usar UDP ou TCP na camada de transporte. Este protocolo define quatro tipos de mensagens: "Confirmable", "Non-Confirmable", "Reset" ou "Acknowledgment". O primeiro tipo indica que as mensagens exigem confirmação, ou seja, espera-se que uma mensagem do tipo "Acknowledgement" seja devolvida após a sua chegada ao destino. As mensagens do tipo "Non-Confirmable" não têm essa necessidade de confirmação. Contudo, se forem usados juntamente com o protocolo UDP, não existe garantia de entrega das mesmas. As mensagens do tipo "Reset" são usadas pelo servidor e reservadas para casos específicos como por exemplo mensagens que se perderam e é necessário repor os contadores de mensagens.

REST é um estilo de arquitetura que, de forma semelhante ao CoAP, segue o modelo *request-response*. Porém o REST não foi desenhado para ser usado especificamente em sistemas onde existe acesso a poucos recursos. É um estilo comum em aplicações de rede. Sendo assim, existe também neste caso um servidor para o qual clientes enviam pedidos. Esses pedidos podem ser qualquer um dos anteriormente mencionados "PUT", "POST", "GET" e "Delete" que pedem ao servidor para *enviar*, *alterar/atualizar*, *receber* ou *apagar* recursos.

Além das tecnologias, modelos e protocolos falados, também a *nuvem* é capaz de oferecer muitos benefícios a sistemas IoT. Um dos maiores problemas em IoT é a falta de recursos (processamento e memória) que surge do fato dos dispositivos, tipicamente, oferecerem poucos recursos. Neste aspeto, o uso da nuvem ajuda através da possibilidade de se usarem instâncias com recursos fornecidos pelo provedor de nuvem (como por exemplo a Google Cloud ou Microsoft Azure). Estes recursos têm por norma um custo consideravelmente barato, dependendo da quantidade e tempo que esses recursos são usados. Atualmente, os fornecedores de serviços na nuvem fornecem também recursos para a gestão de dispositivos, escalabilidade, armazenamento e transmissão de dados em tempo real.

Após o estudo destas metodologias e ferramentas disponíveis, foi desenvolvida uma nova arquitetura para sistemas IoT. Esta arquitetura é constituída por um cluster composto por vários micro-serviços implementados com o uso de contentores docker por forma a facilitar a adição, remoção ou atualização de micro-serviços.

A *Arquitetura de IoT interoperável e tolerante a faltas* (*Fault-Tolerant and Interoperable IoT Architecture*) por nós proposta tenta obter **escalabilidade**, **interoperabilidade**, **tolerância a faltas**, **resiliência**, **disponibilidade** e **facilidade de gestão** de sistemas IoT de grande dimensão através do uso de duas principais ferramentas de livre acesso: *Kubernetes* e *Kafka*.

O Kubernetes é um cluster que permite gerir, configurar ou escalar micro-serviços de forma automática e tolerante a faltas. Neste tipo de clusters, existem um ou mais nós master e nós escravos. Os nós master são os nós que gerem e controlam os nós escravos. Os nós escravos são os que executam os micro-serviços. No Kubernetes os micro-serviços são implementados em contentores docker que são colocados no componente mais básico do Kubernetes chamado *"Pod"*. Os Pods são criados em nós escravos existentes no cluster.

Em conjunto com o Kubernetes está também em funcionamento o Kafka que serve como um registo de eventos distribuído. O Kafka é um sistema que segue o modelo publish-subscribe onde *"Producers"* usam tópicos para enviar mensagens para o Kafka e *"Receivers"* consomem dados desses tópicos. Estes tópicos têm ainda dois novos conceitos adicionados pelo Kafka: *fator de replicação* e *número de partições*. O fator de replicação indica o número de réplicas (de Kafka) pelos quais se pretende persistir os dados recebidos num tópico. O número de partições serve para indicar em quantas partições deve o tópico estar repartido. Estas partições servem para permitir processamento paralelo.

A implementação desta arquitetura permite o envio de dados usando os protocolos MQTT, CoAP e REST e a receção desses dados através de qualquer um destes protocolos e ainda Web-Sockets. O sistema é capaz de traduzir estes protocolos, por exemplo, dados enviados usando MQTT podem ser recebidos através de CoAP. Quando os dados são enviados para o sistema, a

mensagem enviada é guardada no Kafka independentemente do protocolo usado, e quando alguém quer receber dados do sistema, este devolve os dados guardados no Kafka.

Caso seja necessário, a arquitetura foi desenhada para que a adição de novos protocolos seja simples, bastando para isso adicionar novos micro-serviços ao sistema.

De forma a demonstrar as capacidades da arquitetura proposta e desenvolvida nesta tese, foi realizado um conjunto de experiências que avaliam a performance do sistema.

A primeira experiência realizada verifica qual a latência de uma mensagem, de tamanho pequeno, desde o momento em que é enviada para o sistema até ao momento que chega ao seu destino. Esta experiência foi repetida para todos os protocolos com diferentes taxas de envio (1, 10 e 100 mensagens por segundo).

Dos resultados obtidos concluiu-se que a latência de envio de pacotes MQTT é de, aproximadamente, 1.1s, 1.8s e 5.3s para as taxas de envio testadas. Em relação ao CoAP, a latência é de, aproximadamente, 2.8s, 3.3s e 2.8s, enquanto no REST obtiveram-se valores 3.2s, 3.1s e 2.97s.

Em termos de perda de informação (pacotes perdidos), o teste realizado com MQTT foi o único que não apresentou perdas para qualquer uma das taxas de envio usadas. No CoAP, para ritmos de entrada de 1, 10 e 100 mensagens por segundo houve uma perda de 0,92%, 18,52% e 17,58%, respetivamente. No Caso de REST, obtiveram-se valores de 0,52%, 32,48% e 28,33% de perdas para os mesmos ritmos de entrada de dados.

Com base nos valores apresentados, é possível concluir que, em relação à perda de pacotes, quando a taxa de envio é alta, todos os protocolos têm dificuldades em responder. Em relação à latência de envio dos dados, parece existir uma tendência para ser tanto maior quanto a taxa de envio, excepto no caso de REST que não foi identificada uma relação directa.

De forma a avaliar qual o impacto do tamanho de uma mensagem no desempenho do sistema, realizou-se um conjunto de experiencias onde se enviaram mensagens com tamanho grande e com tamanho variável usando MQTT. Os resultados destes testes foram comparados com os resultados obtidos na experiência anterior.

Tendo em consideração que a arquitetura do sistema foi concebida para utilizações em contexto de IoT, mensagens com 39 Bytes são consideradas mensagens pequenas ou regulares e mensagens com 2000 Bytes são consideradas mensagens grandes. Nos testes onde mensagens de tamanho variável foram consideradas, estas tiveram um tamanho compreendido entre 100 e 1000 Bytes.

Com estes testes pôde-se verificar que não existe um impacto muito significativo provocado pelo tamanho das mensagens em termos de latência. Contudo, deparamo-nos com um impacto expressivo, em termos de pacotes perdidos, quando se usam tamanhos variáveis das mensagens.

Em termos de capacidade de recuperação a falhas de nós, foram realizadas experiências onde os nós foram forçados a falhar e verificou-se a capacidade de recuperação do cluster. Neste caso verificou-se que o sistema foi capaz de se adaptar, removendo os pods do nó que falhou, distribuindo-os por outros nós, e retomar o funcionamento normal do sistema.

De seguida, e de forma a avaliar a influência da quantidade de dados em processamento no sistema, foram realizados dois testes de carga, onde os níveis de uso do CPU e memória RAM foram monitorizados. No primeiro teste usou-se um único produtor MQTT para enviar dados, enquanto que no segundo usaram-se 10 produtores. A quantidade de dados recebidos e em processamento aumenta em dez vezes de um caso para outro. Como resultado, concluiu-se que o nível de CPU utilizado não varia muito entre cada teste, mas os níveis de uso de RAM aumentam significativamente.

Sendo a arquitetura proposta escalável em termos de instâncias dos micro-serviços ou em termos do número de nós de hardware que dispõe, realizaram-se também experiências onde se verificou qual o tempo necessário para que o sistema adicione um novo nó (hardware) ao cluster consoante o número de nós existentes. Foram realizados testes com diferentes dimensões do cluster (1, 3, 5, 8 e 10 nós) onde se acrescentava um nó adicional. Deste teste concluiu-se que, independentemente da dimensão do cluster, a adição de um novo nó demora entre 30 e 40 segundos.

Por outro lado, a arquitetura proposta também permite a escalabilidade horizontal, permitindo aumentar o número de instâncias de um micro-serviço. Para avaliar esta componente, configurou-se o sistema de forma a ter um master e quatro nós escravos. Com base nesta configuração, fizeram-se testes com 10, 20, 30, 40, 50 e 60 instâncias de um micro-serviço, e de cada vez que se escalava o micro-serviço verificou-se o tempo que este demorou a passar de estado "Não Pronto" para "Pronto". Desta experiência concluiu-se que o tempo necessário para escalar o micro-serviço aumenta de forma diretamente proporcional ao número de instâncias a escalar. Quantas mais instâncias são precisas escalar maior o tempo necessário para o fazer.

Tendo em consideração a performance, eficiência e o processamento em paralelo que a arquitetura dispõe, realizou-se também um conjunto de testes com diferentes fatores de replicação (1, 3, 7, 10 e 15) que permitiram demonstrar a vantagem da replicação da informação neste tipo de sistemas. Nestes testes verificou-se que para o fator de replicação 1 existe uma latência ligeiramente superior aos restantes. Esta diferença deve-se ao fato de não ser possível existir processamento paralelo visto que os dados não são replicados para outros Kafkas dentro do sistema. Já nos testes com fatores de replicação maior, a diferença em termos de tempo de processamento é notória.

Por fim, foi explorada a possibilidade da segmentação dos dados em partições. Neste caso foram realizados testes com número de partições 1, 50 e 100. Do teste concluíu-se que para 1 e 50 partições a latência foi de, aproximadamente, 1.5s enquanto que o teste com 100 partições mostrou ter, aproximadamente, 27.2s. Esta questão não foi completamente explorada, sendo apontada como trabalho futuro.

Em suma, nesta tese propõe-se uma arquitetura de IoT eficiente, interoperável e tolerante a faltas. A avaliação feita permite concluir que a arquitetura traz vantagens em situações de necessidade de integração de diferentes sistemas IoT num único sistema, onde milhares de dispositivos podem ser conectados para enviar ou receber dados de forma eficiente.

Palavras-chave: IoT, LoRa, Sigfox, NB-IoT, Bluetooth, ZigBee, Wi-Fi, LTE, GSM, MQTT, CoAP, LoRaWAN, AMQP, REST, OPC-UA, Kubernetes, Kafka, Computação em Nuvem, Escalabilidade, Interoperabilidade, Tolerância a faltas, Resiliência, Disponibilidade, facilidade de gestão, Publish-Subscribe, Request-Response, Micro-serviços, Contentores Docker.

List of Figures

List of Tables

Abbreviations

3GPP Third Generation Partnership Project. 18, 20

6LowPAN IPv6 over Low-power Wireless Personal Area Network. 28

AES Advanced Encryption Standard. 25

AMQP Advanced Message Queuing Protocol. 14, 22

API Application Programming Interface. 31

AWS Amazon Web Services. 39

Azure Microsoft Azure. 39

CEP Complex Event Processing. 39

CEPT European Conference of Postal and Telecommunications. 20

CoAP Constrained Application Protocol. 14, 22

COM Component Object Model. 36

CoRE Constrained Restful Environments. 26

CRUD Create, Read, Update and Delete. 34

CSS Chirp Spread Spectrum. 17

DCOM Distributed Component Object Model. 36

DNS Domain Name Space. 45, 60

DOS Denial of Service. 25, 28, 37

DTLS Datagram Transport Layer Security. 28

DUP Duplicated. 24

E-UTRAN Evolved UMTS Terrestrial Radio Access Network. 20

EPC Evolved Packet Core. 20

FIFO First In First Out. 31

FRM Frame. 30

FSK Frequency Shifting Keying. 17

GCP Google Cloud Platform. 39

GSM Global System for Mobile Communications. 16

HetNet Heterogeneous Network. 20

HTTP Hypertext Transfer Protocol. 34

HTTPS Hyper Text Transfer Protocol Secure. 34

HW Hardware. 41

I/O Input/Output. 79

IANA Internet Assigned Numbers Authority. 25

id identification. 36

IETF Internet Engineering Task Force. 26

IoT Internet of Things. 14

JMS Java Message Service. 31

JPMC JPMorgan Chase. 31

LAN Local Area Network. 20

LooCI Loosely-coupled Component Infrastructure. 22

LoRaWAN Long Range Wide Area Network. 14, 22

LPWAN Low Power Wide Area Networks. 17

LTE Long Term Evolution. 14, 18, 20

LTE Cat-M1 Long Term Evolution Category M1. 20

LTE-A Long Term Evolution Advanced. 20

MAC Medium Access Control. 17.

MIC Message Integrity Code. 30

MIMO Multiple-Input Multiple-Output. 20

MQTT Message Queuing Telemetry Transport. 14, 22

NAT Network Address Translation. 34

NBIoT Narrow Band IoT. 14

NesC Network Embedded Systems C. 22

OPC Open Platform Communications. 35

OPC AE OPC Alarms and Events. 35

OPC DA OPC Data Access. 35

OPC HDA OPC Historical Data Access. 35

OPC-UA OPC - Unified Architecture. 14, 22

OS Operating System. 41

PKIs Public Key Infrastructures. 37

REST Representational State Transfer. 14, 22, 34

RUNES Reconfigurable Ubiquitous Networked Embedded Systems. 22

SDK Software Development Kit. 35

SMS Short Message Service. 20

TCP Transmission Control Protocol. 25, 26, 31

TinyOS Tiny Operating System. 22

TLS Transport Layer Security. 34

TTL Time To Live. 32

UDP User Datagram Protocol. 26

UNB Ultra-Narrow Band. 17

URI's Uniform Resource Identifiers. 34

VM Virtual Machine. 41

VMs Virtual Machines. 41

WLAN Wireless Local Area Network. 20

WPAN Wireless Personal Area Networks. 18

XML Extensible Markup Language. 38

ZigBee Zonal Intercommunication Global-standard. 14

Contents

1: Introduction

Nowadays the growth of the Internet usage is quite visible. The number of connected devices is increasing every day. Everything can be a device connected to the network, from smartphones, smartwatches, refrigerators or even televisions. Therefore, there is a net full of things connected to it, building the Internet of Things (IoT).

Most of these "things" are constrained devices because they were developed without high processing, memory or energy capacity. A good example of devices that have these characteristics are smart sensors, such as small temperature or humidity smart sensors. In this context, everything that can be done to decrease the cost of devices or provide better efficiency is very valued.

A good way to improve the efficiency of these devices is by using an appropriate communication protocol that takes into account the features present in those devices. There are already many protocols proposed in the literature, such as Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), Long Range Wide Area Network (LoRaWAN), Advanced Message Queuing Protocol (AMQP), Representational State Transfer (REST), OPC - Unified Architecture (OPC-UA) or Zonal Intercommunication Global-standard (ZigBee), which are appropriated for different application contexts.

Besides the communication protocol, also the communication technology used can improve the capability of sensor networks (things network), which is used for specific application scenarios. There are many communication technologies that allow devices to communicate with each other, such as cellular technologies like Long Term Evolution (LTE) or 5G, satellite or others like Narrow Band IoT (NBIoT). Even with a good protocol and using the best technology available, the IoT system may still fail, and if so, data may be lost forever. To prevent such cases, it would be great if these kind of systems were fault tolerant and able to scale in order to address high number of devices. There are frameworks that allow for an automatic management of system failures already in the literature such as *Kubernetes* and *Kafka*. A full architecture that comprises things, network and those frameworks will impose a new generation of IoT systems where scale and failure handling improves the IoT system.

1.1 Objectives

Depending on the goals of each application, protocols might not satisfy all the needed requirements. For example, the MQTT protocol is a publish-subscribe protocol, a pattern that is dominant in IoT, allowing the communication to have better performance when there are a lot of consumers wanting for the same information. On the other hand, there's, for example, the LoRaWAN protocol that was designed for long-range communications, taking into account the energy spent in the communication. Hence, each protocol might be better suited for different kinds of situations.

Typically, the IoT protocols are proposed individually, taking into consideration their own requirements. As a result, each was implemented separately, having created several brokers and servers that work independently. This hinders the creation of a full interoperable system because for each protocol there is a need for understanding the broker and/or server implementations to deal with information coming from devices.

Moreover, there is a big heterogeneity in the IoT devices [92], and each of these devices may work with a specific communication protocol, making deployments complex when different types of devices are included in the same setup.

Hence, we propose the implementation of a scalable, fault-tolerance architecture that is able to deal with different IoT protocols. It can be used, for instance, to evaluate the protocols behaviour in individual or mixed way, where a device can send, for example, a CoAP or MQTT packet to the system and the consumer can receive it through other protocols like REST or Websocket. This

architecture brings the possibility to have a set of protocols working on a single IoT network or system.

Besides the implementation of the solution, we also intended to perform the tests on the field and use different hardware such as Raspberry Pi's and Arduinos, which allows to obtain real measurements and test, for instance, LoRaWAN in a real environment.

Unfortunately, due to the new Corona Virus, which forced us to stay at home for a large amount of time, the real environmental testing was not done. In order to perform tests of our system we decided to change the initial plan and considered an approach to use the cloud to deploy the system and use our own machines to send data to the system. After some considerations it was evident that the use of the Kubernetes with the cloud (in our case the Google Cloud Platform was considered) would be expensive and these tests would be easier to make on a bare metal environment using a local machine, the LASIGE Navigators cluster (called *LASIGE Quinta*) or simple virtual instances on Google Cloud Platform.

1.2 Structure

The rest of the document is organized as follow: Chapter 2 presents some background of the area and the State of the Art related with high availability and scalable solutions that can be used in the IoT context. The most commonly used Communication Technologies are presented along with their main features and characteristics. Wireless sensor networks and IoT architectures may follow different models, which are also reviewed in this chapter. After that, a review of some communication protocols used in this kind of communications are presented, such as MQTT, CoAP or LoRaWAN, presenting their main features and patterns that are followed. After this, we discuss how IoT may benefit with the use of the Cloud infrastructure and some technologies that allow our system to be scalable, elastic and fault tolerant such as Kubernetes and Kafka. The main characteristics of these technologies are also presented.

In Chapter 3 a requirement analysis is presented. Some use cases and requirements are shown allowing the understanding of what are the main points that an IoT system should be focused on.

In chapter 4, the definition of the system and its architecture is described, explaining the main features and how it can be used in order to have data flowing from device to device without ever being lost or having the network overloaded.

In chapter 5 we define a set of experiments that allow better understanding the behaviour and performance of the proposed architecture, for example if nodes are restored after a fail or how much time does it take to add a new node to the system. The results of these experiments are presented in Chapter 6.

Finally, Chapter 7 concluded the thesis with a reflection of the entire work explaining its achievements, limitations and improvements that can be done in the future.

2: State of the Art

The deployment of sensors for IoT applications is a challenging task because many issues must be handled. For example, data is flowing at high rates and many times this data is quantitatively big. Also, there is a possibility to store the data and only process it later, but this way the opportunity to have a quick reaction to more urgent scenarios is lost. Typically, IoT applications require reliable and low power wireless communication links between nodes and gateways. Examples of such applications are smart homes and health care systems.

A sensor, or *"Things"* in these kind of environment represents a physical object that can be uniquely identified, capable of communicating with other sensors and/or other entities (with the use of an address). It may have the ability to measure specific conditions such as temperature level or proximity to other objects [85]. In order to understand the magnitude of the task that is building an IoT system, in the next sections we reviewed the existing communication technologies, patterns and protocols that are known in the literature and may be used to construct such systems.

Additionally, we know that these kind of systems must be scalable, resilient and fault tolerant. In this context we also revised the benefits of using the cloud and other frameworks that allow systems to fulfill all these requirements. Kubernetes and Kafka frameworks were revised and presented in section 2.4.

2.1 Communication Technologies

The number of IoT devices is growing day by day in different contexts, such as smart cities, smart homes, healthcare among other applications. These devices must communicate wirelessly with each other very often with some requirements, such as long range, low data rate, low energy consumption and low cost [94].

There are several technologies that allow wireless communication between devices, but some are most suited for long range communications and others are more suited for short range communications. There are also some technologies who take into consideration the need for low power consumption, while others don't take this into account. Examples of technologies that were built to provide communication in close range are *Bluetooth* and *ZigBee*. Using one of these technologies for long range communication wouldn't be feasible unless a mesh network was used to extend the reach of the communication, but in that case there would be a need to have intermediary devices (*hops*), meaning that more money would be spent in infrastructure. An example of a long range communication technology is the *Global System for Mobile Communications (GSM)*. This type of technology is not the best suited for sensor networks because it doesn't have into consideration the power consumption of the devices. An example of a technology that allows long range communication and concerns about the devices power consumption is *LoRaWAN*. Because of this, LoRaWAN may be the best one out of these technologies to use in IoT systems [5, 106].

In the rest of this section the most used communication technologies in the context of IoT applications are introduced. The following wireless communication technologies are revised: *LoRa* [13], *Sigfox* [39], *NB-IoT* [94], *Bluetooth* [125], *ZigBee* [83], *Wi-Fi* [82], *LTE* [68], *GSM* [107].

LPWAN Technologies

This technology is being used as a complement, and even replacement, of the traditional communication technologies, such as the Bluetooth or Wi-Fi technologies, allowing the nodes to communicate with low cost, having a radio chipset that costs at maximum 2€, operating cost of 1€ for each device a year [94]. Low Power Wide Area Networks (LPWAN) ensures **low power consumption** with nodes working with a battery with lifetime over 10 years. Besides all of this, LPWAN enables communication over long distances, from 10km to 40km and 1km to 5km distance range, in rural and urban areas, respectively. There are many LPWAN technologies, such as the ***LoRaWAN***, ***Sigfox***, ***LTE-M*** and ***NB-IoT***.

According to Ikpehai et al. [71] and Mekki et al. [94], the most used LPWAN technologies are the LoRa, Sigfox and NB-IoT. LoRaWAN (with LoRA in the Physical layer) is the most deployed LPWAN technology in IoT use cases due to its signal propagation method (Chirp Spread Spectrum [108]) and provides a better coverage, reliability and data rate [94].

- ***LoRa***

  LoRa is an LPWAN technology that uses Chirp Spread Spectrum (CSS) modulation [45]. It was designed to have low power characteristics and increased communication range when compared with, for example, the Frequency Shifting Keying (FSK), which is a often used modulation technique for achieving low power communication [5]. LoRa implements a Medium Access Control (MAC) layer protocol where protocols like LoRaWAN can be integrated giving them access to the LoRa architecture [44].

  With LoRa technology the range utterly depends on the environment and obstructions on it, i.e. for an urban environment the range may be between 2 km to 5 km and for a rural environment this range may go up to around 45 km [5, 44]. Despite this, when combined with LoRaWAN it offers a better *link budget* then most of the standardized communication technologies [5].

  LoRa operates in specific frequency bands that are relative to the region of usage, for example, in Europe it operates in frequency band 867-869MHZ but in North America the frequency band is 902-928MHz [5, 13].

- ***Sigfox***

  Sigfox is also an LPWAN technology, similar to LoRa, working as a network operator but also being a company (while LoRa belongs to *Semtech Corporation*) [71]. It allows communication over long distances with an approximated speed of 100-600 bps (bits per second) [39]. This technology was created in France (Toulouse) and it has been commercialized worldwide due to its partnerships with other network operators [94].

  To make this possible, Sigfox uses a technology called *Ultra-Narrow Band (UNB)* [7] which allows for a bidirectional connection between nodes and a *base-station*. This *base-station* is responsible to send data received from the nodes to the Sigfox cloud, which forwards the information to the client cloud where the data is visualized and manipulated as the client wishes. The Sigfox system architecture is represented in figure 2.1.

  Based on *UNB* technology, due to its high *wave-length* and *low bandwidth*, Sigfox is able to send messages in a reduced bandwidth (less then 200Hz), with the ability to reach more than 30 km distance in rural areas. UNB also provides noise-cancelling features, which improves the communication [39].

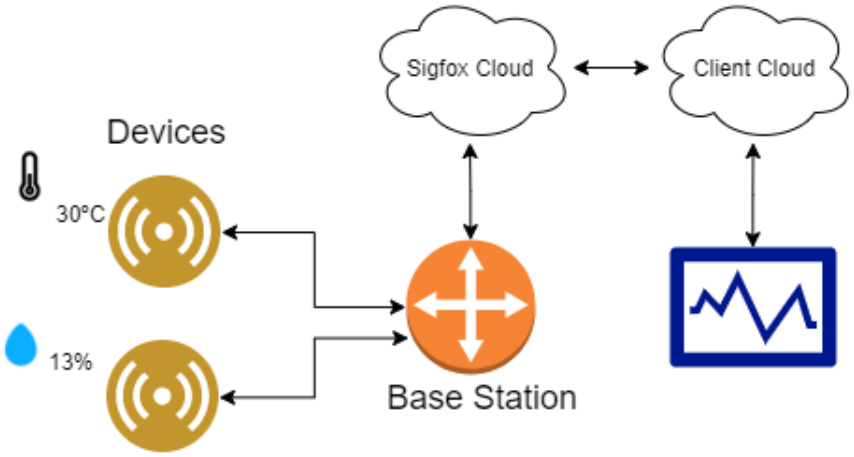


Figure 2.1: Sigfox Network [39]

- ***NB-IoT***

  Another LPWAN technology is the Narrowband Internet of Things [94]. It was standardized by Third Generation Partnership Project (3GPP), and publicly released in June 2016. In December 2016, both Vodafone and Huawei integrated the technology on a Spanish Vodafone network and sent the first message following the standard to a device placed on a water meter [94]. Since this technology rises from another one named LTE it inherited some features and sets of specifications that are very well suited for smart metering use cases. In terms of energy this technology is comparable to LoRa, and even sometimes outperforms it. Battery lifetime may reach more than 10 years [11]. One of the differences between NB-IoT and LTE is that NB-IoT is more stationary, data is transmitted on small chunks and applications are prepared for that. Another difference is that NB-IoT responds well when having a big number of devices, and those devices may even be installed on troublesome places like basements or high points (top of buildings) [93].

Short Range Technologies

- ***Bluetooth***

  Bluetooth technology was created by a group of the biggest companies in 1998 (Ericsson, Nokia, IBM, Toshiba and Intel) as a solution for wireless data communication interface that allowed exchange data in a seamless way [41, 55, 125]. Devices with Bluetooth capabilities can be used as bridges between networks, or nodes of an ad-hoc network, transporting small amounts of data and over a short range. These devices are able to detect each other and connect to one another using radio waves with frequency 2.4 GHz. This kind of applications are known as Wireless Personal Area Networks (WPAN) [55].

Bluetooth devices may work in two different modes, the ***Slave*** mode and ***Master*** mode. The Slave nodes may be active or asleep and are connected by one or more Master nodes. Nodes may act as both the Slave or the Master at the same time in order to connect multiple *Piconets*, creating the *Scatternet*. ***Piconets*** are groups of up to eight active nodes working together, with one Master node, seven Slave active nodes and zero or more sleeping slave nodes. When Piconets are connected together they form a larger Bluetooth Network called the ***Scatternet*** as visible in figure 2.2. Masters on the network send inquiries to know which nodes are available on the area. If there is a recently awaken Slave node in the area, it will respond to the Master providing its address. After this, the Master may send a request to establish a connection, which must be accepted by the Slave. Both devices then start synchronizing in a specific frequency defined by the Master, which is different for every Piconet, allowing data to be exchanged. When the connected devices are not needed in the network, they have the option to enter power or bandwidth saving modes [55].

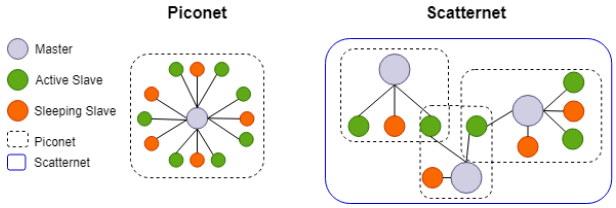


Figure 2.2: Overview of the *Piconet* and *Scatternet* [55]

- ***ZigBee***

  ZigBee is a Wireless Personal Area Networking (WPAN) open standard created by the *ZigBee Alliance*. it was designed specifically to meet sensors and control devices needs, such as low *latency*, *cost*, *energy consumption* and low *bandwidth* [83].

  It is based on the IEEE 802.15.4 Medium Access Control (MAC) and Physical Layers providing technologies that enable networks to be scalable, self-organizing and self-healing while trading data. It is characterized for having a very low rate, which consequently consumes less power and increases the battery lifetime. Although its range is not very wide, it can be used with a mesh network, which provides high reliability and larger range [3].

  ZigBee has two major low power consumption modes, the *TX/RX* and *Sleep* mode. On the *TX/RX* mode, sensors send (TX - transmit) and receive (RX - receive) the data that is collected, while on the *Sleep* mode devices are placed in stand by. There are two different ways to transmit data called *Traffic Types*. They are:

  - ***Periodic***: The application must provide the value of the data rate. Devices will activate on the given time/rate, check for data, send it and then deactivate again [3].
  - ***Intermittent***: Traffic may be stimulated by an application or an external stimulator defines the data rate. For example, a light switch or smoke detectors. This way, data is not send periodically, instead, different amount of time may pass between each send (intermittent sending) [83].

- ***Wi-Fi***

  The Wi-Fi, Wireless-Fidelity, protocol (also referred to as Wireless Local Area Network (WLAN) or IEEE 802.11) allows us to have a Local Area Network (LAN) on which devices are connected and are capable of communicating with each other wirelessly. Devices such as laptops or smartphones connect to a router or access points (hotspots) which generates Wi-Fi signals [122]. IEEE approved the 802.11 standard (Wi-Fi) in 1997 specifying the MAC and Physical layers for transmission with 2.4 GHz band and signal rate up to 2 Mbps [55, 82]. Many amendments were made from then on, and in 2009 an amendment called the *IEEE 802.11ad* started with the goal of addressing the 60GHz band technology with higher bandwidth of 2160 MHz [82, 102] and data rate of 6.7Gbps with a cost of distance (this data rate is only possible at 3.3m of the access point) [112]. IEEE 802.11 is based in a cellular architecture where devices, also called ***Stations (STA)***, are connected with a base station called ***Access Points (AP)*** (also called medium) which is connected to a LAN, when there is one. STA's may be mobile, portable or stationary and must support *STA services*. *STA Services* are the *authentication*, *deauthentication*, *privacy* and *data delivery*. AP's provide *distribution services* which may be the *association*, *reassociation*, *disassociation*, *distribution* and *integration*. Since the architecture is not centralized, it is very flexible and helps preventing bottlenecks, allowing for small networks with few devices and large networks with several access points and servers. Also allowing for these networks to be permanent or semi-permanent, lasting for several years or for a short duration time [100].

Cellular Technologies

- ***LTE***

  The LTE standard was first released in 2009 by the 3GPP. LTE is an IP-based network that brought higher data rates and lower latency than those of the *3G*. This technology allowed cellular operators to use a wider spectrum. To achieve this goals the *air interface* and the *network architecture* evolved to the Evolved UMTS Terrestrial Radio Access Network (E-UTRAN) [50] and Evolved Packet Core (EPC) [62], respectively. To test LTE technology, the first deployment was made in a limited scale, in Scandinavia. Nowadays deployments are being done all around the world in a large scale [42, 86]. Along the years many releases have been published by 3GPP with improved performances of the LTE. In the 10th and 13th Release the Long Term Evolution Advanced (LTE-A) and Long Term Evolution Category M1 (LTE Cat-M1) (LTE-M) were announced, respectively. The former uses a high-reorder Multiple-Input Multiple-Output (MIMO), carrier aggregation and deployment strategies that are based on Heterogeneous Network (HetNet), which is an heterogeneous network topology that uses *macrocells*, *picocells*, *metrocells*, *relays* and *femtocells*. The latter was specially designed for an IoT environment, being an LPWA technology that supports lower device complexity, extended coverage, bigger battery lifetime at lower costs [42, 86].

- ***Global System for Mobile Communications***

  GSM technology is related to mobile communication specifically, being the most used standard in cell/mobile phones. With this technology people from all around the world are able to make wireless calls. It was created by the European Conference of Postal and Telecommunications (CEPT) in 1982, being only commercially from 1991 on, providing standard features such as phone call encryption, data networking or Short Message Service (SMS) [52].

2.2 Communication Models

The *Publish-Subscribe* and *Request-Response* models are commonly used communication approaches in IoT architectures [92, 104]. The following subsection presents a description and discussion about each pattern in order to have a better understanding about how they work.

2.2.1 Publish-Subscribe Pattern

The Publish-Subscribe Pattern has three main components, the *Publisher(s)*, the *Broker(s)* and the *Subscriber(s)* that work together to establish communication (figure 2.3). This communication works around *Topics* that describe data is going to be exchanged using a string format. This string may contain multiple levels, divide by slashes, for example: topic 1: "room41/device10", topic 2: "room41/device10/temperature" [104]. Every time data is published or subscribed, it must be done on a given Topic, which is registered by the Broker on the moment of the publishing or subscribing if it doesn't exist already.

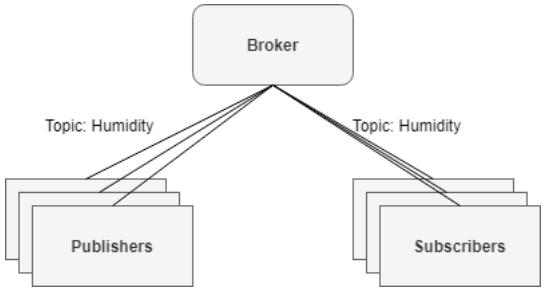


Figure 2.3: Pub-Sub Pattern [104]

The Publisher is responsible for the publishing of data (information for example, a temperature value) to the Broker (on a specific topic). In turn, the Broker is responsible to forward the data received from that publisher and topic to all Subscribers that had subscribed the topic. These exchanged data is the information included in packets with a specific format defined by communication protocols. They are called messages.

Subscribers are responsible for subscribing to a specific topic, receiving messages from the Broker every time that a publisher sends messages to the topic. Both Publisher and Subscriber act as Clients and neither of them needs to know about each other's existence. The communication is decoupled which means they also don't need to run synchronously [92]. For example, considering a temperature sensor that has a new value to output, it will be published to a Broker on a specific topic (e.g. topic "temperature"). If a client wants to receive data from the topic, it needs to known the name of the topic and subscribe it by sending a subscribe message to the Broker. Upon new data is received at Broker, it will be forwarded to all registered clients.

The Publish-Subscribe pattern is a widely used model in IoT architectures because it is very light weight and efficient in terms of resources and scalability [104].

2.2.2 Request-Response Pattern

On the other hand, the Request-Response pattern doesn't include a Broker. Instead of a Broker, there is a component called Server, which sends data to Clients as visible in figure 2.4. The data exchanged is identified by resources and those resources are made available for clients. For example, if a temperature sensor sends a PUT request with payload "23ºC" to a resource called "Temperature", this payload will be saved by the Server and everytime another client performs a GET command requesting for that resource, the server provides that temperature value to the client. The commands available to request the server are the PUT, GET, POST and DELETE. These commands allow managing the content and the resource itself on the server. A good example of this pattern is a RESTful API, which is widely used and it is supported by the HTTP protocol where Web Browsers act as client in order to put and ask for resources [92].

Unlike to the Publish-Subscribe pattern, this pattern requires a synchronous communication schema, so a direct coupling between the communicating clients is needed [92].

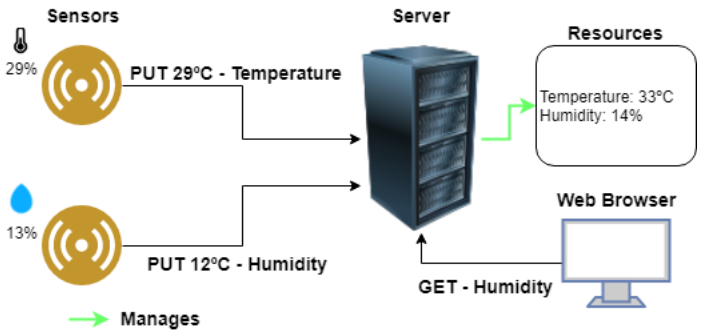


Figure 2.4: Request Response Pattern

2.3 Communication Protocols

There's a lot of research being done around communication protocols for IoT. These protocols were defined to fit embedded devices and try to improve the communication between those devices. Some good examples of these protocols are MQTT, CoAP, LoRaWan, AMQP, REST, OPC-UA and Zigbee.In this chapter we will review and discuss the main aspects of these protocols. Although, there are many other protocols such as Loosely-coupled Component Infrastructure (LooCI) [70], Network Embedded Systems C (NesC) [65] used in the Tiny Operating System (TinyOS) operating system [88], or Reconfigurable Ubiquitous Networked Embedded Systems (RUNES) [47]. These protocols follow the publish-subscribe pattern but are less used in the literature. They were developed for specific contexts and are not generic protocols.

In IoT architectures there are two main communication styles (**publish-subscribe** and **request-reply** patterns [104]) used to exchange data between the devices. Both are widely used, but according to Gowri S. R. et al. [104], the request-reply pattern isn't appropriated to be used in constrained devices because it is not designed to be scaled and its portability is weak.

In these subsections we will discuss the communication protocols that are being researched and used by the community (meaning scientists, researchers, students and even companies). We concentrate our discussion around the MQTT, CoAP, LoRaWAN, AMQP, REST and OPC-UA protocols. The majority of these protocols fit in the Application layer of the TCP/IP model as shown in Figure 2.5.

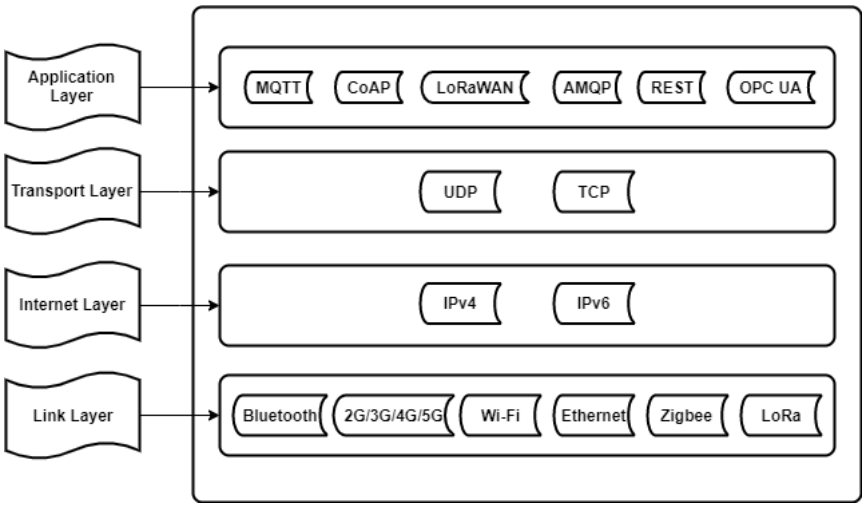


Figure 2.5: Communication Protocols on the network layers - TCP/IP model [84]

2.3.1 MQTT

MQTT is a communication protocol standardized by OASIS [38] that uses the publish-subscribe pattern to establish communication. It provides one-to-many communication (one sending, multiple receiving or vice-versa), being one of the most used protocols in IoT applications. MQTT is open-source, lightweight, data-centric and designed to be used in constrained environments like in IoT [92, 104]. Devices involved in the communication do not need to know the address of other nodes (consumers) because data is forwarded by a Broker and everyone just needs to know the Broker address [92].

MQTT runs over TCP/IP or over any other network protocol that provides ordered, lossless, bidirectional connections. The protocol's first version was MQTT v3.1.1 being also standardized as ISO/IEC 20922:2016. Nowadays MQTT v5.0 is available bringing better scalability, improved error reporting, *Session Expiry* or *Message Expiry* [38, 101].

Functionality

MQTT follows the Publish-Subscribe communication model and describes how the communication should occur between end-devices. It is supported by a Broker, which its responsibility is to forward the published messages, on a specific topic, to the subscribers. These end-devices are any device that intends to send or receive messages (Publisher or Subscriber).

There are many types of messages available to be sent, such as the Connect, Publish or Subscribe Messages, for creating a connection to the broker, publish data to the broker or subscribe data from the broker. The MQTT protocol defines three QoS levels. These levels are used to assure that the communication is being done under certain level of assurance.

MQTT Message Format

This protocol works by exchanging messages between nodes, these messages are series of MQTT Control Packets that are divided in three parts, the **Fixed Header** with 2bytes of size, the **Variable Header** and the **Payload** as shown in Figure 2.6 [38].

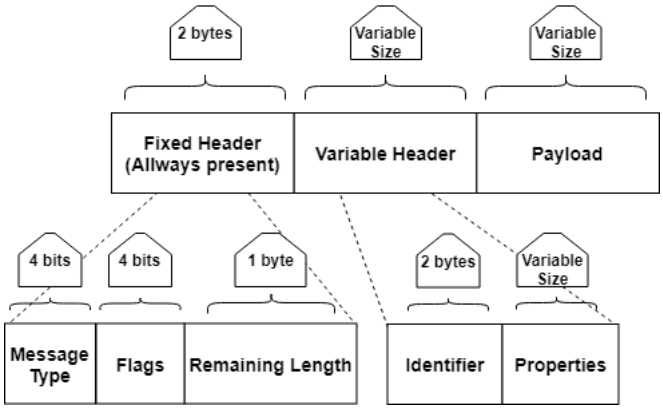


Figure 2.6: MQTT Control Packet Format [38]

The **Fixed header** is present in all MQTT Control Packets. It has 1 byte (8 bits) reserved for the *type of message* and for *flags* (4 bits each), which represents the *MQTT Control Packet Type*. The final byte represents the *Remaining Length* containing the number of bytes remaining within the current packet, which includes the *Variable Header* and the *Payload*.

Variable header is only present in some MQTT Control Packets, they are: ”CONNECT”, ”CONNACK”, ”PUBLISH”, ”PUBACK”, ”PUBREC”, ”PUBREL”, ”PUBCOMP”, ”SUBSCRIBE”, ”UNSUBSCRIBE”, ”SUBACK”, ”UNSUBACK”, ”PINGREQ”. This header’s content depend on the packet type [38]. For most of the packet types, the Variable Header contains a two Byte Integer packet *identifier field* that once set for an MQTT Control Packet, it only becomes available again for reuse when the sender has processed its acknowledgement. This header finishes its fields with a set of *Properties* which are composed by a field with Property Length corresponding to the length of the properties and the actual Properties, which includes an Identifier that defines the message’s usage, data type and value.

The **Payload** field is where the actual content of the message is. It is not included in all MQTT Control Packets.

QoS

In this protocol the concern is only with the delivery of an application message from a single sender to a single receiver, this way when a server is sending messages to many clients, each of them are dealt with independently [38]. The MQTT protocol offers three different levels of QoS:

- QoS 0 - “At most once”

  Best effort, sends only once. Some messages may never reach the destination. The receiver sends no response and the sender doesn’t resend messages, so the message either gets there at the first try or it doesn’t reach the destination at all [38].

- QoS 1 - “At least once”

  The sender re-sends messages if they don’t get to the destination, some messages may be duplicated but there is assurance that the message is received at least once. The sender waits for a PUBACK, from the receiver, after sending the PUBLISH packet “publish1”. If it is receiving other PUBACKS the sender compares each message Packet Identifier to know if any of them are from the message “publish1”. If the sender waits a reasonable amount of time without getting the PUBACK, it re-sends the “publish1” message [38].

- QoS 2 - “Exactly once”

  Messages are assured to arrive exactly once at the cost of some performance by the usage of a four-part handshake. The sender assigns an unused Packet Identifier to a PUBLISH packet before sending it, and then waits for a PUBREC, meanwhile re-sends the PUBLISH packet in a reasonable amount of time. After receiving the PUBREC packet, sends a PUBREL packet and waits for a PUBCOMP packet (with Duplicated (DUP) flag). After receiving the PUBCOMP the Packet Identifier becomes available again. An illustration of the protocol using this QoS level is shown in figure 2.7 [38].

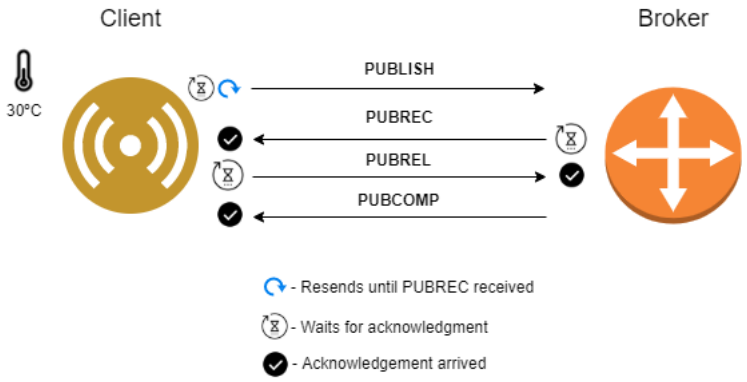


Figure 2.7: MQTT QoS level 2 [119]

Security

MQTT protocol specification lets to developers the choice to have **privacy**, **authentication** and/or **authorization** implemented. Although there is a strong recommendation to use the Transmission Control Protocol (TCP) port 8883 (Internet Assigned Numbers Authority (IANA) [48] with service name: "*secure-mqtt*") [38].

To understand how to better security and how to protect the communication with MQTT, there are some threats that were identified and are very important to fight [38]:

- *Devices can be compromised:* Developers must be aware that the devices in the communication might be hacked, therefore being compromised and possibly work in malicious ways, for example sending packets with malicious content.
- *Possible access of the data in the Clients and Servers:* The same way that devices may be compromised, so their data may also be compromised, meaning that even if a device is not compromised, it can still work with maliciously manipulated data that had previously been deformed by an attack.
- *Protocol behaviors could have side effects:* This protocol has specified QoS levels which send additional packets in the communication to guarantee its level of assurance that the message was received, an example of this is the Acknowledgment message. This kind of behaviour is specific for each protocol, so attackers can possibly use *timing attacks* [111] to find vulnerabilities on the communication.
- *Denial of Service (DOS):* One of the most common attacks on the web are the DOS attacks, which stands for Denial of Service, in this kind of attacks, attackers use malicious software to request operations on the system, in this case, the Broker, in the order of millions or more requests, which will consume a lot of effort from the system to respond to them, and actually resulting on some failures and real requests to not be dealt with, denying the services that the system provides.
- *The communication can be intercepted, modified, re-routed or deleted:* Even if the devices and their data are protected, between the time that the data leaves the device to get to the Broker, and from the Broker to the devices the communication might be intercepted, meaning that a sent message might be caught by a malicious program and be changed, so the communication channels also must be secured.
- *Malicious MQTT Control Packet may be sent to the net:* Attackers may also inject spoofed MQTT Control Packets into the network, meaning that this messages have false IP addresses that are read by the system as true IP addresses misleading the Brokers to accept the malicious messages when they shouldn't.

A good way to protect the communication could be to use an implementation that follows the specific industry security standards such as NIST Cyber Security Framework or PCI-DSS [38].

In terms of cryptography, the most used algorithm is the Advanced Encryption Standard (AES), but in the case where the systems are embedded many of the devices may not support this kind of algorithm (concerning the Hardware), a better solution could be the ChaCha20 [99], which encrypts and decrypts the messages much faster, but it not as widely available as the AES algorithm [38].

Implementation of MQTT

A very widely used implementation of this protocol is the *Mosquitto, through its Broker, Publisher and Consumer* developed in C programming language with its open source available on git hub[1] [89, 57] . The state-of-the-art also comprises other Brokers that provide MQTT based communication, such as *Kafka* [54] or *RabbitMQ* [115].

Kafka was developed to be used in data centers, being relatively heavier than Mosquitto, but bringing some other features such as ordering guarantees and configurable persistence of messages [54].

RabbitMQ is an open source message Broker that was developed with Erlang [8] by the Open Telecom Platform framework [54].It allows communicating through MQTT and other protocols such as AMQP [123].

2.3.2 CoAP

A group of the Internet Engineering Task Force (IETF) named Constrained Restful Environments (CoRE) built an application layer protocol called CoAP [63]. It is based on the REST protocol (using HTTP functionalities). This protocol is very suited for the IoT environment since by default it uses User Datagram Protocol (UDP) instead of TCP. UDP has some known issues, being one of them the reduced reliability when comparing with TCP, because of this, IETF has started a new document for the standard to open the possibility for CoAP to run over TCP, being this only in the beginning [51].

This protocol follows the request-response model. It supports the GET, POST, PUT and DELETE methods. Even though CoAP modifies some of these functionalities to guarantee better efficiency when the devices are constrained, having low processing capacity or bad links on the communication [63]. To improve the communication, CoAP responses are not sent over a previously established connection, instead they are exchanged asynchronously over CoAP messages which are packets that have a specific format defined by CoAP protocol. This improves the performance but may decrease the reliability [51].

Figure 2.8 represents how CoAP works. Since REST architecture is already very widely used with HTTP, by having a Proxy in between to translate this two Protocols, we can have a network of things communicating through the internet using CoAP. *Things* communicate through CoAP using a CoAP Server, that communicates to the rest of the Internet using the REST-CoAP Proxy to make a straightforward conversion between the two protocols [63].

CoAP is very lightweight being ideal for IoT environment with constrained devices. However, it still offer low portability for a wide range of platforms and shows poor scalability [104].

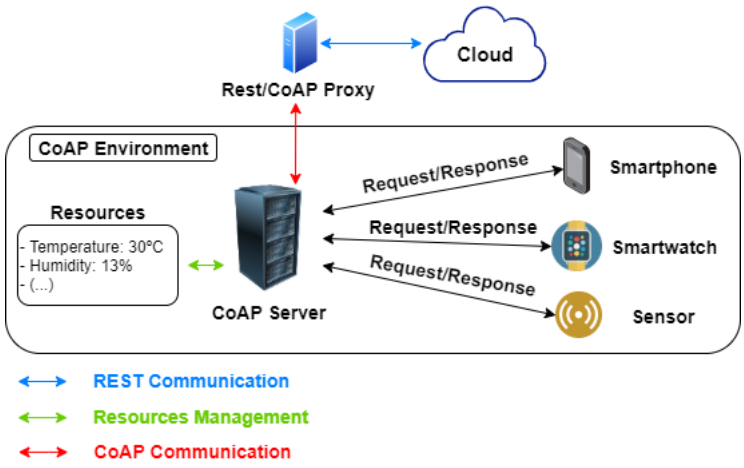


Figure 2.8: CoAP Functionality [63]

[1] https://github.com/eclipse/mosquitto

Functionality

CoAP is divided into two different logic layers, the first layer is the *Request/Response* layer, which deals with the implementation of the REST paradigm, similar to HTTP. What differentiates CoAP from HTTP is the second layer, called *Message* layer. This layer deals with the re-transmissions when packets have been lost, since UDP has no reliable connections as explained in 2.3.2 [51].

Since CoAP is based on HTTP commands, when an end-device wants to send data to a server it uses the *Put* or *Post* command, and when a device wants to get some information from the server it uses the GET command. To ensure that a messages arrives at destination, CoAP defines four types of messages (described below), which are used to perform handshakes assuring the message has arrived or not.

CoAP also has an option to improve the request-response approach by allowing clients to continuing receiving messages from a resource that was requested on the Server (observable option). This is done by adding an *observe* option to the GET request, which is added to the Server observer's list. When the resource state changes, clients in the list are notified. Through this option, CoAP offers a closer behaviour to the publish-subscribe pattern with the clients being notified of changes in the resources they ask for.

CoAP Message's Format

A CoAP message is divided in four fields, as shown in the figure 2.9. The first field is fixed and corresponds to the ***Header***. It occupies four bytes. The Header is composed by the *CoAP* ***Version***, the ***Type*** *of transaction*, the ***Option Count*** (OC), the *request method/respond code* (**Code**) and a message identifier (**Message ID**). The second field is an optional ***Token*** value that is used for correlating requests and responses, this value can occupy up to eight bytes. The next two fields are optional and they are reserved to add ***Options*** and the ***Payload*** [63]. A typical CoAP messages occupy from ten to twenty bytes in total [63].

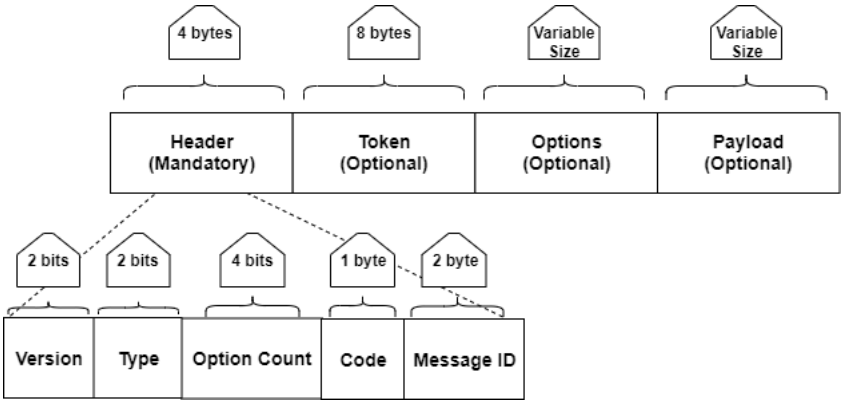


Figure 2.9: CoAP Message Format [63]

Type of messages

At the *message* layer level four different types of messages [51]: *CONFIRMABLE*, *NON-CONFIRMABLE*, *RESET* and *ACKNOWLEDGMENT* are defined.

When a *CONFIRMABLE* message is sent by a client, it waits for an ACKNOWLEDGMENT from the CoAP Server, re-sending the original message with a certain interval until it gets the *ACKNOWLEDGMENT*.

When using the *NON-CONFIRMABLE* type, the client won't be waiting to receive an *ACKNOWLEDGMENT* message from the Server, it just sends the message and hopes it gets there. There is no guarantee that the CoAP Server actually got the message. The *RESET* messages are used by the Server when it notices that messages were missed or other issues occurred [63]. *ACKNOWLEDGMENT* messages are used to acknowledge that some message has arrived (CONFIRMABLE Messages). This doesn't mean that it indicates success or failure of the request made,

but it may contain a response for that request, in this cases the response is called a ***Piggybacked Response*** (Acknowledgment with reply) [113].

Security

In terms of security, CoAP makes use of Datagram Transport Layer Security (DTLS) which is based on a slightly changed TLS protocol to achieve better compatibility with constrained devices. The DTLS runs on top of UDP transport layer [105, 51]. Some of the modifications include stopping the termination of the connection when packets are lost or out of order. Another example is the handshake process, where the possibility to add a verification query by the Server. This feature allows making sure that the Client is sending the packets from the authentic source address, which prevents DOS attacks [51].

By using DTLS, every time a message is sent or arrives at a device, the protocol dictates the decryption and decompressing that must be preformed [105]. However, DTLS protocol wasn't built for the IoT environment. Based on DTLS, the literature comprises the IPv6 over Low-power Wireless Personal Area Network (6LowPAN). This new versions tried to optimize it for the constraint devices situation [51, 105].

CoAP defines four security modes [105, 114]:

- ***NoSec:*** The *NoSec* mode means *"No Security"* which dictates that there is no security provided in the CoAP transmitted messages, hence packets are sent over UDP and IP.
- ***PreSharedKey:*** In the *PreSharedKey* mode, there is a list of pre-shared keys where each key have a list of nodes and may be used to validate the communication. In this mode there is the possibility to have a key for each device or for a group of devices. The mandatory cipher suite that must be implemented on this mode is the *TLS_PSK_WITH_AES_128_CCM_8* specified in [49].
- ***RawPublicKey:*** This is a mode where a device may have one or more asymmetric pairs of keys without certificates (*Raw Public Keys*). For this mode the mandatory cipher suite to be implemented is the *TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8* specified in [118].
- ***Certificates:*** This mode works around a certification chain, where authentication is based on public key. Each time a new connection is established, certificates from external devices must be validated. Sometimes, nodes may need more than one certificate as they may have more than one Authority. This mode must include an implementation of the *TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8* cipher suite, specified in [49]

In *Raw Public Key* and *Certificate* modes, connections that are established are set up with authentication in both ends of the communication which allows to maintain the connection in order to reuse it in future data exchanges.

Implementation

This protocol only defines how the communication should be done, allowing anyone to implement its features. Many implementations of CoAP were proposed throughout the years, as suggested in [43]. Some examples are the *Libcoap* [40] which is a C library and *CoAPthon* [64] a Python library. With each of this libraries it's possible to build our own server, clients and resources without much complexity. However, these implementations are not suitable for small embedded devices, such as *Arduino*. To be able to transmit information with CoAP protocol in this kind of devices specific *Arduino* libraries must be used, for example *CoAP_simple_library* is one of them.

2.3.3 LoRaWAN

LoRaWAN is a media access control (MAC) protocol for wide area networks. It is designed to allow low-powered devices to communicate with Internet-connected applications over long range wireless connections. Typically, a star architecture is used and three kind of possible devices are defined [5, 13]:

- *End devices*: nodes who send the data to the *LoRa gateway*
- *Gateway*: forward packets sent by those nodes to the *Network Server*
- *Network Server*: manages information that was exchanged. servers that route messages from End Devices to the right Application, and back.

This kind of architecture preserves the battery lifetime plus through the use of LoRa on the physical layer, long range is achieved [5]. Figure 2.10 shows a typically architecture for LoRaWAN protocol. Through this architecture all application complexity is implemented in the server [5, 13].

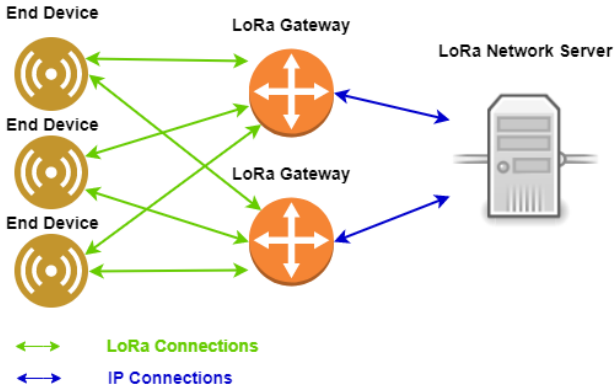


Figure 2.10: Lora's Star Architecture [13]

Functionality

LoRaWAN supports many gateways receiving data from nodes. Each gateway forwards the data received to the destination using the cloud-based network with, for example, WiFi or Ethernet connection. The complexity may be pushed to the cloud side of the network [5]. The communication occurs in an asynchronous way, and only when data is ready to be sent, which can be done via event-driven or scheduled way [5].

Message Format

LoRaWAN architecture is divided in two layers, the *Application layer* and the *MAC layer*, both on top of *Lora's Physical Layer*, as shown in figure 2.11 [81].

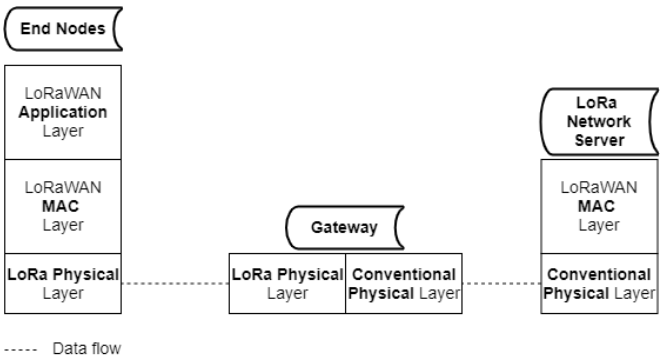


Figure 2.11: LoRaWAN Architecture Layers [81]

Figure 2.12 represents the *MAC layer* message format that is used by LoRaWAN. This format includes: a MAC header with one byte, a MAC Payload with variable size and the Message Integrity Code (MIC) (similar to a checksum) with four bytes that is build based on the previous MAC Header and MAC Payload fields with use of a Network Session Key [81, 98]. The *Network Session Key* is an application session key that is generated when a device joins to a network, being shared with the network, and is used for validate the exchanged messages between the nodes and the Network Server using the MIC [98].

Wrapped inside the MAC Payload is the Application message, which has the following format (also visible on the 2.12 figure): the first field is the ***Frame (FRM) Header*** with seven to twenty two bytes. It is followed by a multiplexing port called ***FPort*** with one byte and the ***FRM payload*** with variable size, which is encrypted with an *Application Session Key*. An *Application Session Key* is also generated when a device joins a network just like the *Network Session Key*, but this key is kept private, being used for encryption and decryption of the payload based on the AES 128 algorithm [81, 98].

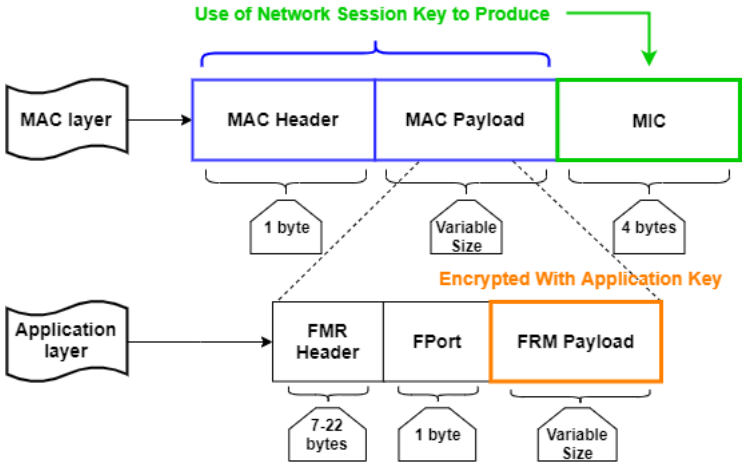


Figure 2.12: LoRaWAN Packet Format [81]

Security

LoRaWAN specifies a number of security keys [5]: NwkSKey, AppSKey and AppKey. All keys have a length of 128 bits. The algorithm used for this is AES-128, similar to the algorithms used in other low power standards (e.g. 802.15.4).

- The Network Session Key (NwkSKey) is used for interaction between the Node and the Network Server. This key is used to validate the integrity of each message by its Message Integrity Code (MIC check). This MIC is similar to a checksum, except that it prevents intentional tampering with a message.

- The Application Session Key (AppSKey) is used for encryption and decryption of the payload. The payload is fully encrypted between the Node and the Application Server component. This means that nobody except the user is able to read the contents of messages you send or receive.

- The application key (AppKey) is only known by the device and by the application and is used to derive the previously mentioned NwkSKey and AppSKey.

The session keys (NwkSKey and AppSKey) are unique per device and session. When a device joins to the network (this is called a join or activation), an application session key AppSKey and a network session key NwkSKey are generated using the AppKey. The NwkSKey is shared with the network, while the AppSKey is kept private. These session keys will be used for the duration of the session.

2.3.4 AMQP

In what concerns asynchronous messaging, one of the most recognized standards is the Java Message Service (JMS), but it is an Application Programming Interface (API) standard, not a specification of protocol [123].

In order to change this situation and have an open asynchronous messaging standard protocol that provides interoperability and is extendable at business scale, in 2006 the *AMQP* protocol was proposed by the working group JPMorgan Chase (JPMC), composed by companies such as Cisco Systems, Envoy Technologies, iMatix Corporation, IONA Technologies, Red Hat, TWIST Process Innovations and 29West [123].

AMQP is a publish-subscribe protocol that allows applications to send and receive messages defining the specification of what messages and from where they can be received. It also provides functionalities to configure security, reliability and performance [54, 126]. The protocol runs over TCP [126]. AMQP protocol was designed and built based on the finance requirements, which are very challenging because they require extremely high performance, throughput, scalability, reliability and manageability. In this kind of business area, few microseconds might be a real problem, making them loose or win against competition on a trade. Also losing messages is not an option, messages arrival must be assured as they might mean costs to the enterprise [123]. Figure 2.13 shows the AMQP architecture.

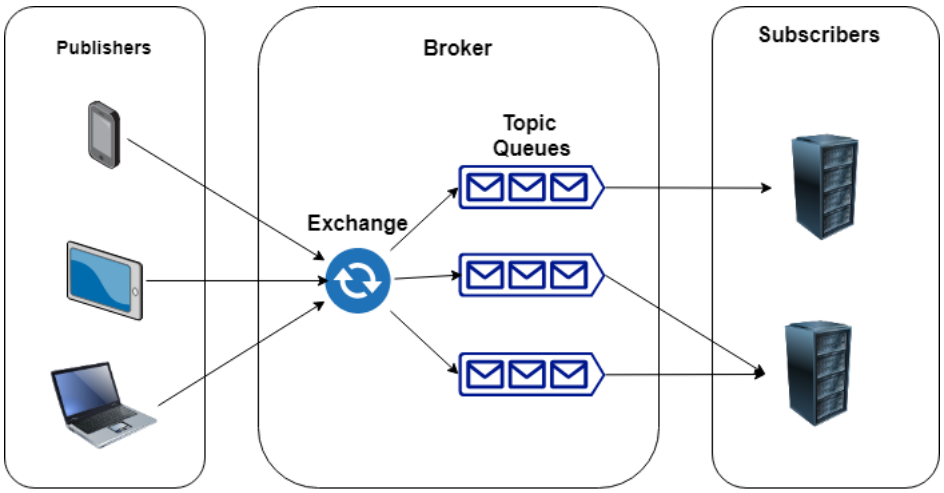


Figure 2.13: AMQP Architecture

Functionality

In AMQP architecture, client applications are known as producers ("publishers") while AMQP server is known as broker. Client apps create messages which are given to the broker. In the broker, messages are routed and are queued using a First In First Out (FIFO) order. They are being read by consumers from the queues where they are processed. These client applications are known as consumers ("subscribers"). AMQP divides messaging Broker tasks in two modules:

- *Exchanges*: is the component responsible for accepting messages. This messages are afterwards forwarded to specific queues based on a set of rules and criteria, never storing them [123, 126]. The messages contain routing keys which are used by "exchange" module in order to route them. There are three different types of exchange methods

  - *direct exchange*: A routing key is used to bind the exchange to a message queue. From then on, messages from publishers will only be routed to that queue if the routing key from the publisher matches the routing key used to bind the exchange and message queue [9].
  - *fanout exchange*: Message queue binds without routing key, meaning that it receives all messages published, there is no condition associated [9].

  - *topic exchange*: When a message queue binds to an exchange, a routing pattern ***P*** is associated with the binding and that queue will only receive messages that are published with a routing key ***P*** that matches the routing pattern, otherwise messages aren't not forwarded to that queue.

- *Message Queues*: is the component that stores messages and send them to the respective consumers. Usually messages are stored just until they are sent to the consumer [123, 126].

Therefore, there is a responsibility chain, where each processor acts on the received message, rejecting it, adding/modifying it or just forwarding it to the destination. This architecture improves flexibility because it allows developers to add more components or modify the ones that already exist [123].

Message Format

The AMQP message format is composed by five fields, as illustrated in figure 2.14. it is composed by an ***Header***, a ***Delivery-Annotations***, a ***Message Annotations***, a ***Bare Message*** and a ***Footer*** fields.
The ***Header*** field is divided in five sections [117]:

- ***Durable:*** A durable message means that even if an intermediary node in the communication unexpectedly terminates, the message is still preserved. This section is of type Boolean (1 byte) with False value by default.
- ***Priority:*** Responsible for carrying the level of priority of the message represented by a *ubyte* (1 byte) value, the higher number represents the higher priority.
- ***Time To Live (TTL):*** Defines the time that a message is considered to be "alive" in milliseconds, being discarded when this time has passed. This helps preventing delivery loops.
- ***First-Acquirer:*** If this field has a true value, it means that the message has not been acquired before, but if the value is false, then it is not guaranteed that the message has been acquired before, it only **may** have been. Represented by a Boolean value (1 byte)
- ***Delivery-Count:*** Number of unsuccessful delivery attempts represented by a *uint* (1 byte) value. Non zero values may mean a duplicate message.

Delivery-Annotations field is used to carry properties that aren't specific from the standard, such as information from the sending node to the receiving node. These annotations must be known by both ends for the properties to be transmitted.

Message Annotations is a field that carries properties of the message that are propagated in every hop of the communication. These annotations carry information about the message. In the communication process, some intermediaries may not understand the annotations's purpose, they may only be used as attributes which are filtered.

Each ***Bare Message*** is composed by:

- ***Properties:*** Standard properties of the message are saved here, such as the *Message-Id* - unique identifier of the message, *User-Id* - unique identifier of the user that produced the message or *"to"* - property which identifies the message destination address.
- ***Application-Properties:*** This field is reserved for structured application data, which may be used or filtered by intermediary nodes.
- ***Application-Data:*** Contains binary data corresponding to the payload.

Footer field is used to store details about the message delivery, only being calculated after the bare message has been totally finished, or being evaluated before seeing the message content. Examples of this details are hashes, signatures or encryption details [117].

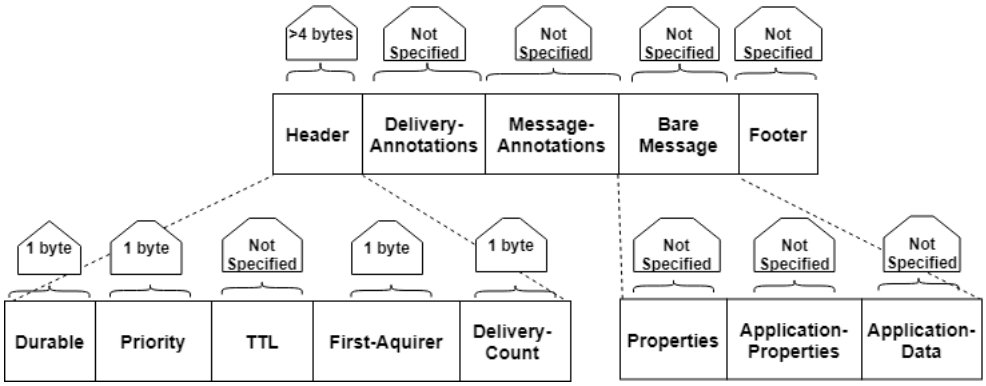


Figure 2.14: AMQP Message Format [117]

QoS

AMQP uses three different message delivery guarantees [126]:

- **"At most once"**: messages don't arrive more than once, so there can't be duplicates but on the other hand messages might be lost
- **"At least once"**: uses acknowledgements to guarantee that messages arrive at least once but adding the possibility to have duplicates
- **"Exactly once"**: messages are assured to be received once, and not more than once

The definition of these message delivery guarantees is very similar to the QoS level of MQTT protocol. The basic process of communication over AMQP using the RabbitMQ implementation can be seen in the following figure 2.15 [54].

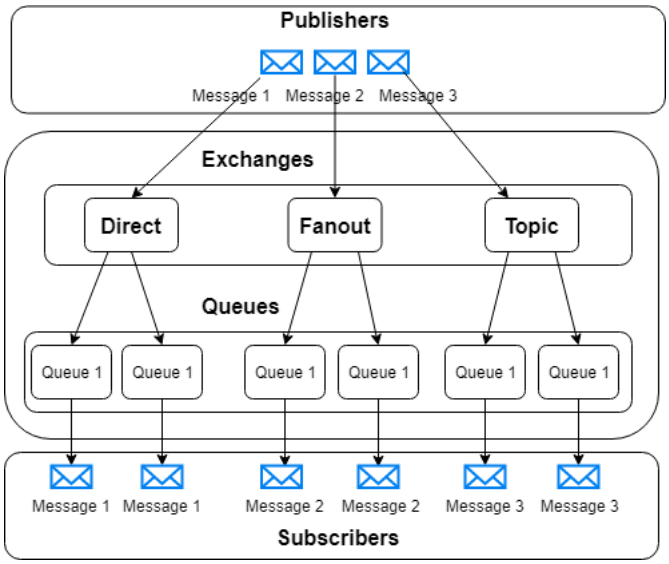


Figure 2.15: AMQP with RabbitMQ [54]

Implementations

A good and quite used implementation of this protocol is the RabbitMQ Broker that allows the communication over AMQP to be used. Since there are many libraries for developing clients (publishers and subscribers), it is easy to establish the communication between clients and the AMQP Broker. One good example is a Python library called *Pika* [90]. This Python library has appropriate functions for publishing messages to a specific Broker or subscribing to a topic on a specific Broker. This way, each end device may have its own *Pika* based implementation allowing them to connect to one or more Brokers, publishing and receiving messages on subscribed topics.

2.3.5 REST

The REST architectural style is used to build network applications. It follows the client-server style where the idea is the server is responsible to provide resources to clients that request for them. Resources may be anything in REST, as long as they can be named and identified. REST identifies resources through Uniform Resource Identifiers (URI's).

With this request-response approach, clients and servers must know each others addresses to communicate. This might be a problem sometimes, for example if Network Address Translation (NAT) is used. NAT is a technique that uses a hash to hide the IP addresses. In this case clients and servers may not know each-others addresses, making the communication impossible.

The URI's, or *namespace*, are defined as, for example, "http://127.0.0.1/nodes", where "http://" defines the protocol to be used, "127.0.0.1" is the address (localhost in this particular case) and "nodes" is the actual resource that we are trying to reach. A call for each URI can return a single value or a collection of values, in this example a possible return could be a list of nodes [92].

Functionality

REST uses the very well known Hypertext Transfer Protocol (HTTP) (GET, POST, PUT and DELETE) methods. It follows the Create, Read, Update and Delete (CRUD) model, having a method to *create*, *read*, *update* and *delete* a resource on the Server side.

POST and PUT methods are very similar, bringing some confusion to its difference, because they don't exactly match the *create* and *update* methods of the CRUD model. POST means that the server will decide how to deal with the information given by the client when receiving the method, in order to update its resource, while PUT means that the Server will replace the content received on the PUT request with the resource's content, creating the resource if it doesn't already exist.

GET method is used to fetch the resource information, but while doing it, the resource remains unchanged. All other methods change the resource, since POST and PUT may create or update the resource value both may change the resource. The DELETE method deletes the resource from the server, making it unavailable by other clients to perform actions on it [1, 92].

REST is stateless, which means that neither the server nor the client keeps track of each other's state, both keep only their own state. This way, servers don't keep information about previous requests done by clients, it just saves the current state of the resources. The same way, the clients keep their own state, not knowing the state of the server, having to ask for it with a GET request to know any of the resource values [92].

QoS

Nowadays most RESTfull Web services ignore QoS parameters, as their only concern is to provide the method's functionality. However this QoS parameters may be very useful in the case where there are multiple providers for the same kind of request. This way, the client could take advantage of this QoS parameters to choose the most suitable provider. These QoS parameters can be described in the HTTP payload [1].

Security and Privacy

To assure security in RESTfull Web services there's the need to secure not only the data that is being exchanged, but also the hole communication.

To secure data, its integrity must be assured as well as its confidentiality, which means that no one should be able to modify the data nor have access to it (actually knowing the content of the data).

To secure the communication, there should be an authentication process that ensures access control over it, allowing it to be private so that no one could intercept the communication and listen to what both ends are trying to exchange.

For this, there are some protocols that work on top of HTTP, as the Hyper Text Transfer Protocol Secure (HTTPS), which uses an additional security layer with Transport Layer Security (TLS) protocol. There are also other technologies being built to support authentication for HTTP-based services, such as *OpenId*, *XAuth* or *OAuth* [1].

2.3.6 OPC-UA

Another protocol being explored to be used within the IoT environment is the OPC-UA protocol. Open Platform Communications (OPC) is an open standard created by the *OPC Foundation* in 1996. It was designed to provide reliable and secure communication in many industries, such as automation or renewable energy industries. OPC is a standard that allows information to be exchanged between several devices from different vendors, increasing interoperability which is a big deal in IoT. Following the Client-Server pattern, its design is based on specifications made by industry vendors, end-users and software developers that define several features such as access to data in real-time, monitoring of alarms or events (with OPC Alarms and Events (OPC AE)) and access to historical and current data (with OPC Historical Data Access (OPC HDA) and OPC Data Access (OPC DA) respectively) [60, 87].

The OPC Foundation was motivated to create the OPC-UA standard because although all the other mentioned extensions of the OPC standard (OPC AE, OPC HDA and OPC DA) fulfill their purpose, there was no link between them, a value read with DA and HAD have no connection, but with OPC-UA all of them are related to each other [87].

Figure 2.16 represents the OPC UA Architecture.

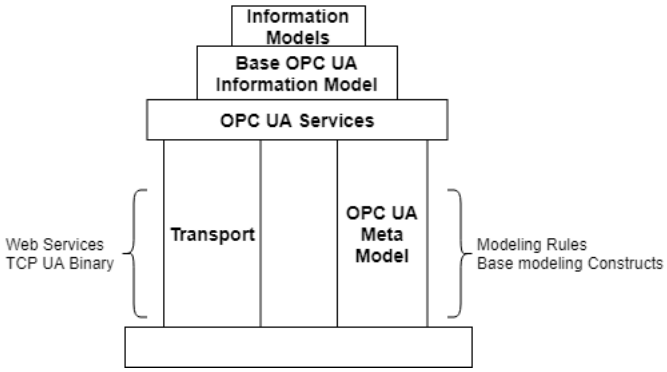


Figure 2.16: OPC UA Architecture [91]

This layered architecture's specification have five goals [59]:

- ***Function Equivalence* goal** where all COM OPC classic specifications are mapped to UA
- ***Platform Independence*** that allows communication on different platforms, for example, an embedded device and cloud infrastructure
- ***Secure* goal**, because security must be assured, using authentication or encryption
- ***Extensible* goal**, which defines the new features that must be handled without compromising older features, and finally
- ***Comprehensive* information modeling goal** where complex information is defined

The main components of OPC UA architecture are the ***transport mechanisms*** and ***data modeling*** as shown in figure 2.16. The transport mechanisms defined are the *UA TCP*, which is a simple protocol on top of TCP that provides some necessary mechanisms for OPC UA (example: *special error recovery* that allows OPC UA to survive from network interruptions) and *SOAP/HTTP* that provide a firewall mechanism for OPC UA applications to communicate on the internet via Web Sockets. Developing this layers requires a lot of time and effort, so OPC Foundation provides a set of OPC UA standards that have all the mappings implemented, which are available on the OPC UA Software Development Kit (SDK) [91].

In what concerns data modeling, in the classic OPC standard the data exchange was "raw"/"pure" as it works with items containing data, for example, the temperature obtained on a sensor, but OPC UA uses a ***Meta Model***, that is also known as *Address Space*, where data is represented trough objects, and where can be applied object-oriented techniques, such as hierarchies and inheritance, having the possibility to have an hierarchy where, for example, the previous temperature value is know to be from a specific sensor device in a specific house [87, 91].

In this Meta Model, there is the concept of a *meta node*, which is the base of it, and as so, there is also several node classes that can be defined to specialize different classes of nodes. On each node, there is a set of attributes, some mandatory, some optional. An example of mandatory attribute is the *identification (id)* that identifies the node uniquely and an example of an optional attribute is the *description* [87].

Functionality

As a follower of the Client-Server pattern, the OPC standard specifies the away of communicate between Clients and Servers, where Servers provide services do clients request them. These services are saved on an address space that is used by the clients when having the need for a service. This way, an OPC client connects to an OPC server being able to ask the server for services, such as access to *real time or historical data* and *events*. For example, an application that has the OPC DA feature will request for services on an OPC DA Server [91].

Most OPC interfaces are based on Microsoft's Component Object Model (COM) [96] and Distributed Component Object Model (DCOM) [95] technologies which brought an advantage to the development of OPC because a lot of computers are based on the Windows operating system that already has these technologies available, although this brings a dependency on the windows platform [91].

Message Format

A message exchanged between a client and server in OPC UA is separated in chunks, which are blocks that divide the message in different parts. Each chunk contains specified information about the message and the actuators. Sometimes messages are longer than the maximum network packet size, when this happens these messages are divided and reassembled in a process called "Chunking".

A typical UA TCP message chunk format is composed by [91, 35]: .

- A ***Message Header*** which identifies the *type of the message* and *message size*. The *type of the message* is defined with the use of a **three byte ASCII code**. The possible message types are the HEL (Hello) message, ACK (Acknowledgment) message, the ERR (Error) message or RHE (Reverse Hello) message and they are all specific to the UA TCP protocol [91, 58, 35].

  The *message size* contains the length of the message in bytes and is defined using an *UInt32* value, which is an unsigned integer number with 32 bit capacity (8 bytes) meaning that **8 bytes** are reserved to specify the length of the message [91, 58, 35].

  This means that every message will always contain at least 11 bytes 3 bytes for the message type plus 8 bytes for the size of the message. However, each message type adds some more fields, such as, for example, the *ReceiveBufferSize* and *SendBufferSize* that represents the largest message chunk that the sender can receive or send, respectively [58].

- A ***Message Body*** which contain the actual message (payload) encoded or one of the mentioned UA TCP-specific messages types that define the socket connection and possible errors.

Beginning with the HEL message type, there are six new fields added, where five of them use *UInt32* (8 bytes each, resulting in a total of 40 bytes) plus one field that uses the data type *String* which can't be more than 4096 bytes. Resulting on a total of 4136 maximum bytes for this kind of message. The following type of message is the ACK message, which has five more fields, all of them defined using an *UInt32* value, which means that this type of message has at most, 40 bytes. Next, the ERR message type only have two fields represented with *UInt32* and a data type *String*, which means a total of 4104 maximum additional bytes for this kind of message. Finally, the RHE type of message also contains two fields, both represented with String data type, which means a total of 8192 maximum additional bytes for this message [58].

The message structure is presented in figure 2.17 [58, 91].

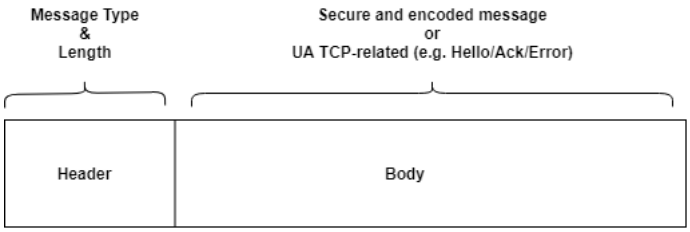


Figure 2.17: OPC UA Message Format [91]

Security

The first thing to be taken in consideration, in what concerns security in communication with OPC UA, is the range of scenarios where it is applied. To assure that the communication is secure, there were taken into consideration the need to have six security goals: *Authentication*, *Authorization*, *Confidentiality*, *Integrity*, *Auditability* and *Availability* [91].

To achieve these goals OPC UA uses many security measures. One of this measures consists on the specification of the *SessionID*, *TimeStamp* and *SequenceNumber* fields on the message that help prevent replays, being verified in every message exchanged. Another measure is the possibility to sign messages with, for , a Public and Private keys before sending them, This can be done by using certificates, managed by Public Key Infrastructures (PKIs), to provide the private keys, having to verify this signature for every message that is received. Through this way, we have assurance that the message received was not altered, otherwise the signature verification would have failed.

OPC UA assigns security responsibilities in a ***Security Architecture*** composed of three layers:

- *Application* layer, which is responsible for the *User and Product Authentication and Authorization*. It manages the session between the client and server
- *Communication Layer* secures the communication channel, being responsible for the *Application Authentication and Authorization* as well as *Confidentiality and Integrity* of the data exchanged by the use of encryption of the information and applying digital signatures
- *Transport Layer* layer, which is responsible for the *Availability*, dealing with the transmission of data via socket connection. It uses mechanisms for error recovery which may be explored with attacks like DOS

All layers and security responsibilities are shown on figure 2.18. With these architecture the application can change or replace any of these modules easily without having to redesign the whole system [87, 91].

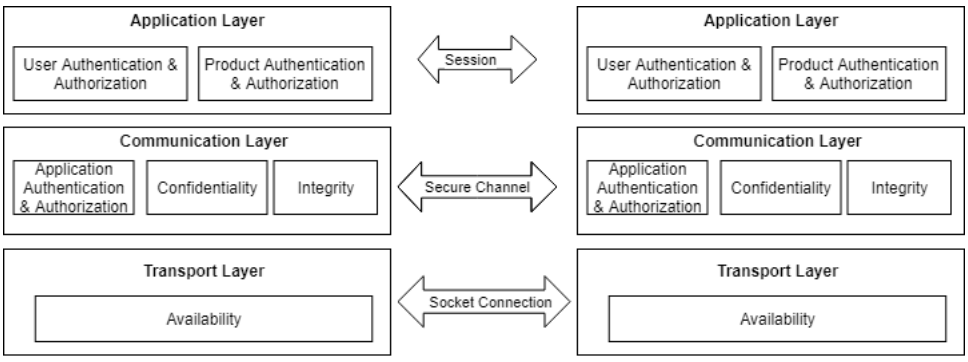


Figure 2.18: OPC UA Security Architecture [91]

Implementation

One important thing concerning OPC UA implementation resides on three tasks, the *data encoding*, *communication security* and *data transportation* (figure 2.19) [91]. Concerning data encoding, Extensible Markup Language (XML) or *UA Binary* may be used. The *UA Binary* is a serialization of the data into a byte string which is faster to do than with XML encoding, because the size of the message is smaller. However, if XML encoding was used, XML SOAP clients can interpret the data. Typically, XML encoding is only used with UA Web Services. [87].

In the transport layer, *UA TCP* or *SOAP/HTTP* may be used, both deal with the establishment of the connection between OPC UA client and server. *UA TCP* is a small protocol build on top of TCP with the purpose of getting the possibility to negotiate buffer sizes, share IP-Addresses and Ports, and create their own reacts to errors that occurred on transport level. The format of the UA TCP Message is described in sub-subsection 2.3.6. The SOAP/HTTP means SOAP over HTTP, meaning that SOAP, as a network protocol based on XML uses HTTP for data transportation, being widely used to have secure communication between Web Services and Web browsers [91].

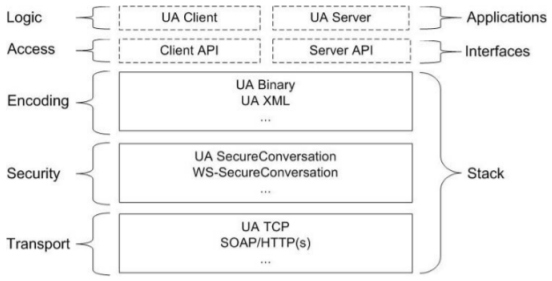


Figure 2.19: OPC UA Mapping [91]

2.4 IoT and Cloud Computing

In this section we revise the main technologies that allow to improve IoT systems in terms of scalability, elasticity, availability and node computation management. The first technology presented is the Cloud. We start by explaining how can IoT benefit with the use of the Cloud, what it can bring to the IoT, how it works, what type of clouds exist, and at the end a small explanation on the differences between Elasticity and Scalability. After this, a description of Monolithic and Micro-Services approaches is shown, explaining both ways of building a system and why we may benefit from using a Micro-Services approach when it comes to an IoT system. Afterwards Kubernetes is presented, explaining advantages and disadvantages that it can bring into a system, such as node recovery and update features. Finally we explain the Kafka distributing system platform, how it can bring many benefits to an IoT system, such as replication of data or parallel computing [109].

2.4.1 Cloud and its Benefits for IoT solutions

Both IoT and Cloud were proposed independently but there are some benefits of putting them together. From the IoT point of view, problems arise when a large number of constrained devices are integrated, since many devices have low energy capacity and computing power and often connections aren't stable. The cloud is able to partially solve this problem by providing virtual unlimited storage and computation power. These resources are considerable cheap since there is no need to buy hardware, software or perform maintenance on machines, the service being used is the only thing that has a price to be paid. Some IoT requirements that the Cloud is able to provide are *Device Management*, *Scalability*, *Storage* and *Real time Transmission*.

The cloud is divided in three different types, the *Private*, *Public* and *Hybrid Cloud* [85].

Private Cloud: These type of cloud provides better privacy control since the used servers, infrastructures and other resources are owned by an organization. They are physically located on the corporation's data-center. However, the whole infrastructure must be maintained, and since this infrastructure may be complex, the task would be very difficult. These sort of service is usually good when there is a need to have full control of the data, making it easier for the organization

to configure its resources and meet its IoT needs. Generally these type of cloud is very expensive and only owned by big corporations, government agencies, financial institutions or organizations between mid and large size that have a critical business operations and need to have exceptional control [37, 85].

Public Cloud: In this type of cloud, servers and the infrastructure are managed by cloud service providers, and they are the ones that will deal with many tasks that are time consuming and some of them very complex, such as ensuring limited access and employment of security mechanisms. Because of this, this kind of cloud is the most commonly used, some examples of these type of cloud are the Google Cloud Platform (GCP), Amazon Web Services (AWS) and Microsoft Azure (Azure). It should be noted that resources are shared with other organizations and clouds. It's possible to access cloud services and manage the user account on a browser. These type of clouds are very cheap and often used for online applications, storage and to perform experiments of some new developed software or service [37, 85].

Hybrid Cloud: There is another approach combining both previous type of clouds, this approach is called Hybrid Cloud. This type of cloud allows for separation of data to be used. Hybrid Cloud is maybe the *"Best of both worlds"* allowing for users to have the advantages of both previous types of cloud. Data and applications may be moved between private and public clouds as needed. When there is a need for High-volume and lower security a public cloud should be used. When there is a need for sensitive data, business-critical operations the private cloud should be used. An IoT system working within the cloud is shown below on fig 2.20 [37, 85].

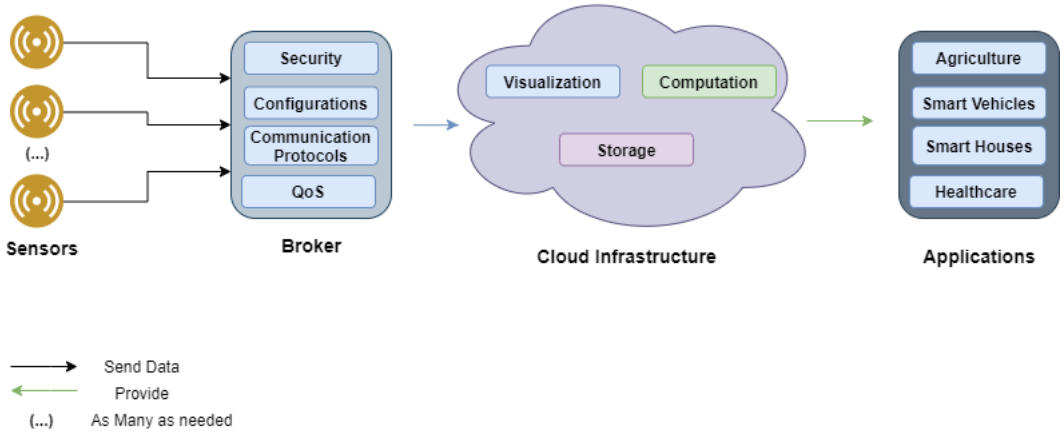


Figure 2.20: Cloud and IoT Network [85]

Benefits of using cloud resources to build new IoT solutions:

With the use of the cloud we get access to many resources. It offers numerous **storage** capacities to better suit users needs. The cloud allows us to gather and process the essential information that is needed within the raw data which is being received by the nodes. To do this, it process the data with advanced data mining tools, like Complex Event Processing (CEP) [85]. CEP [56] is a framework that allows conversion of raw events in relevant data while it is still being transmitted. The price charged for the storage is directly proportional to the amount of storage reserved and used. The more storage reserved for a machine, the more that machine is going to be charged over time. When it comes to *Computational resources*, since IoT devices are very constrained, having the computation of the data on such nodes would sometimes be infeasible, some king of tasks would be too complex for such nodes but not for the cloud. Plus, in the cloud there is also the possibility to choose between processing the incoming data in real-time, store the data and process it later, for example in a data base, or even have some kind of visualisation of the data, for instance with a dashboard.

In IoT, *Scalability* is one of the fundamental requirements to fulfill and cloud computing can help a lot in this factor. In most systems it's very common to have pikes, sometimes there is a need for more processing power due to more activity, like having a tremendous amount of requests by users. Therefore it's very important to have these kind of systems prepared to respond to such events. Systems can be **scaled up**, meaning that it is able to get more resources when needed or

scaled out, which means that the system is able to revert the resources obtained when they are not needed anymore.

Therefore, **Scalability** refers to the ability of a system to respond to an increase of workload without having the system's performance changed. A system is scalable if it capable of increasing the system resources such as CPU power, amount RAM or amount of nodes in the system when the workload compels it to. It is possible to do this management automatically or manually (in a periodic way). The main focus is that the system should always be able to deal with the increment workload.

On the other hand, there is the concept of Elasticity. **Elasticity** is a system's flexibility to manage its resources according to the workload necessity (increasing or decreasing). Even though very similar to scalability, it has a different meaning, in this case the concern is about the ability to respond to an increased and/or decreased workload, meaning that the system must be flexible and dynamic when provisioning resources, being able to get more resources and to withdraw them as the result of a better performance by the system. To have an elastic system, first it must be scalable [2, 85].

Both of these concepts are inherent to cloud platforms and help to create new efficient IoT solutions

2.4.2 Monolithic vs Micro Services

When building a system two different approaches may be used. We can have a **Monolithic** system or we can use **Micro-services**. Both have some benefits and drawbacks.

In a **Monolithic** approach all components of a system are packed together. A good example that illustrates a Monolithic system is a web application. In a web application, the system is composed by the *Client-side*, the *Server-Side*, *database*, and other layers may be used. All of them are built in a unique logical place. This may cause some difficulties when there is a need to update a certain part of the system, since the whole system must be updated. Moreover, scale a system like these would have its adversity's, because since the system is packed up together, everything would need to be scaled together. Another thing that might cause problems is a situation where a component fails, in this case the entire system would go down. In IoT, this issue remains, a monolithic IoT system would have all the logic components in a single place. The processing of data, the communication with the devices and database and the visualization, everything would be in a single location [85, 109].

Micro-services approach may solve some of these problems. Micro-services allow developers to build a system composed of many small independent services. In order to get the most of the integration of IoT and cloud computing, the use of micro-services is advisable. By decomposing the system in small independent components that implement their specific functionality within the system it becomes possible to update, scale, and recover from failures without the entire system being down, only the specific micro-services that are being modified (down, updated or scaled) are unavailable. However, it must be taken into consideration that the communication between Micro-Services is very expensive and it should be minimized. If changing a micro-service implies many changes in other micro-services, the interest of using micro-services may be lost [85, 109].

In summary, Monolithic applications are composed of a single executable having all of its features defined in one place. When the system needs an update, the entire system must be updated and redeployed, and therefore the whole system must go down to make the necessary changes and get back up after they have been applied. With Micro-services only the specific component that needs to be updated is actually stopped, updated and redeployed while the rest of the system keeps on running. Besides this, while the monolithic systems are updated they increase their size over time until a moment where it starts to get very complex to manage, causing more space for errors to appear, and as explained previously, failures in these type of system lead the entire system to crash. With micro-services concepts may be separated, decreasing complexity, and even if failure occurs, only the particular micro-service's implemented functionality goes down, all other features are up and running [85, 109].

2.4.3 Kubernetes

Micro-Services bring many advantages. However, having a system composed of a large number of components may lead to high complexity when managing them.

If many components go down, it would be very time consuming to configure all of them back up one by one, so there should be some kind of mechanism that could do it automatically. This is where kubernetes is able to help. Kubernetes is an open-source technology that allows micro-services to be extensible, configurable and automatically managed with fault tolerance [27]. It is an open source platform that manages container workloads and services. Containers are used to aggregate all that is needed to run a specific program, from dependencies to executable.

Looking a little back in time, traditional deployments would have applications being executed on servers. This worked fine, but resource allocation problems started to emerge since in a single physical server there were difficulties defining boundaries for each running application. For example, from all the applications running on a physical server one could request for most of the resources causing the other applications to have a decreased performance. Hence, Virtualization appeared as a solution [34].

Supported by the Virtualization concept it is possible to run numerous Virtual Machines (VMs) in only one individual physical machine with the use of what is called the *Hypervisor* [74]. The Hypervisor is the firmware/software that creates and runs the VMs. The machine where the VMs are created is called the *Host* and the VMs created are called *Guests*. Each machine runs its own Operating System (OS). The host may, for example, be running on a Windows OS and have VMs running Linux. With virtualization, applications become isolated between each Virtual Machine (VM), providing better security, scalability and improving resource boundaries. Plus, this method allowed costs to be reduced in terms of Hardware (HW). Only later containers appeared [34].

A lightweight approach of virtualization to deploy applications is based on applications running in containers. The objective of containers is to have all dependencies and all needed information aggregated in one unit so that the whole system works as intended regardless of where it is executed. This method avoids many problems when installing software in different platforms. With containers, applications may be deployed quickly and independently of the environment or infrastructure they are being executed in. To use containers, there are some available tools called *Container Runtimes*, namely the *Docker*, *CoreOS rkt* [78] or *Containerd* [78]. Docker is by far the most used Container Runtime by the community and it is free to use (open source). To have an owned application running on a docker container it's necessary to have a **Docker Image**. Docker images are very lightweight, they contain all information that the system needs to be able to run such as the code, run-time, other applications, settings and even system libraries. These images are executed in the *Container Runtime* by the *Docker Engine*, and it is at this moment that they become **Docker Containers**. Figure 2.21 shows that inside of a machine with its own OS Docker Engine runs Docker images in the Container Runtime and that the docker images may be of any kind from a Mosquitto, to a WebServer. Unlike VMs, Containers virtualize the OS instead of the hardware [74].

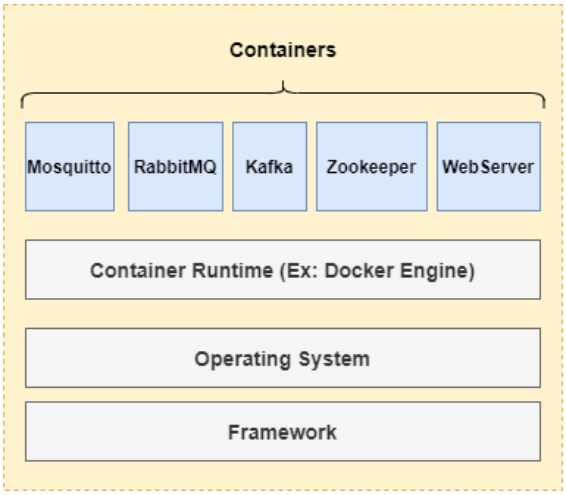


Figure 2.21: Container's role [74]

Kubernetes is the framework that will help us manage these containers by allowing **self-healing**, managing **storage**, executing **load balancing** and allowing for **secret** and **configuration** management [34, 77]. To do this, the desired state of the cluster must be declared using the Kubernetes *API objects*, meaning that the applications to be executed must be specified, for example, the container images to use must be described, the number of replicas needed for each of them, how much disk resources should be used or which network configuration to use. There are two different ways to use the kubernetes API objects, the first is the command-line interface called *Kubectl*, the other is to use the kubernetes dashboard which is not installed with kubernetes by default, but can be easily added (a step-by-step is provided in appendix A). After the desired state has been declared, the *Kubernetes Control Plane* will make sure that the cluster's state corresponds to the described one. To maintain the desired state, kubernetes will either restart containers if any of them fails or start them if more are needed and will also scale the replicas of an application when the workload obliges it [14].

Kubernetes Components:

Having Kubernetes installed means having a cluster composed of numerous machines or nodes (at least one), each running containerized applications. These nodes are where *Pods* are deployed [25]. Pods are the smallest and simplest deployment units in Kubernetes. They run as processes in the cluster and each of them may have one or many containers running applications inside them. Pods specify storage resources, have their own IP address and information about the container behaviour. If the system is supposed to be horizontally scalable, meaning that more machines are started in opposition of getting more resources to the only machine available, then each Pod should run a single instance of an application, meaning that it should have only one container, and many Pods should be used to run all the instances needed [26].

Otherwise, replication can't be achieved because if we put many containers in each pod, since every pod will have its own IP address, all the containers will be scaled together [85], plus if a pod with many containers fail, all the applications running in the containers will also fail.

To manage pods and the containers inside them the **Kubernetes Control Plane** is used [25]. Kubernetes Control Plane contains a collection of processes running on the cluster [14].

The whole Kubernetes system can be seen in figure 2.22 including the *Kubernetes Control Plane* components and the *Kubernetes Node* components. The Kubernetes Control Plane deals with global decisions and/or respond to events on the cluster. Although able to run in every node, it is usually working on the master node. The Kubernetes Node runs the *Kubelet* and *Kube-proxy* on every node of the cluster. Besides these components, there is also the concept of objects. Kubernetes may have use of many different objects, such as Services, Pods, Ingresses, Daemon Sets and many more. The ones that we considered to be the most important are described with more detail in this section.

Kubernetes Control Plane Components:

- **Kube-Controller Manager**:

  It is the unit responsible for running *controller* processes. This includes the *Node Controller*, the *Replications Controller*, the *Endpoints Controller*, the *Service Account Controller* and *Token* controller. The Node Controller is responsible for checking the node status and act when needed, meaning that it should be able to, for example, identify when a node goes down and respond by evicting the pods out of the crashed node. The Replication Controller is the one responsible for maintaining the number of replicas defined for the pods. Endpoints Controller is the controller that populates the objects of the cluster, such as pods or services. Finally, the ***Service Account*** and the ***Token*** controllers are responsible for creating default Kubernetes user accounts and tokens that give API access to new *Namespaces* [21].

- **Cloud-Controller Manager**:

  This component is the one that contains cloud control logic. It allows for Kubernetes to be linked to a cloud provider API. It is possible to separate components from those that interact with the cloud and those that interact only with the cluster.

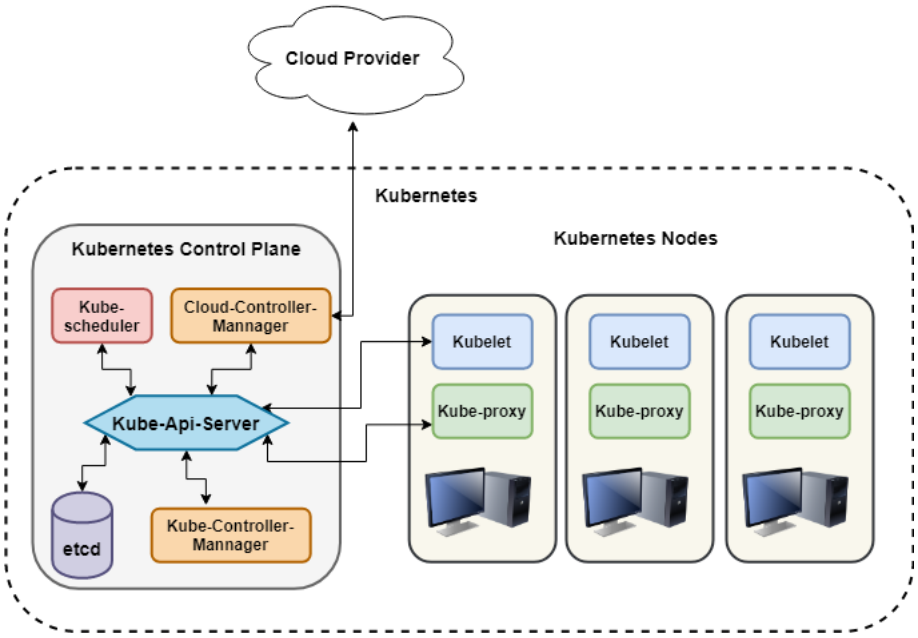


Figure 2.22: Kubernetes Components [25]

They are the *Node Controller*, *Route Controller* and *Service Controller*. The Node controller has the same responsibilities as in the Kube-Controller Manager, but has dependencies when dealing with the cloud provider, it needs to contact the cloud provider to determine if a node is down. The other two controllers also must interact with the cloud to obtain information. The Route controller is responsible for setting up routes in the cloud infrastructure. *Service Controller* needs to be able to create, update and delete *Load Balancers* from the cloud since it is the controller responsible for managing the *Services* of the cluster [25].

- **Kube API Server**:

  This component is responsible for exposing the Kubernetes API. It is the front end of the Kubernetes Control Plane and was designed to validate and configure data for the API objects, such as replication controllers, pods and services [20].

- **Kube Scheduler**:

  This component is responsible for finding new pods that are not assigned to any node and select an optimal node for them to run on. The selection of the node needs to take into consideration many factors, such as the individual and collective resources requirements, QoS, constraints related to the hardware, software or policy, where data is placed or even deadlines. If the scheduler can't find a node that satisfies the requirements for a Pod to be run on, then it isn't deployed, remaining unscheduled until the scheduler finds a node that is suitable for deployment [23].

- **etcd**:

  All the cluster data is stored in a key-value backup storage called etcd [25]. These data includes the cluster status, its configuration and specifications [79].

Kubernetes Node Components:

- **Kubelet**:

  Runs on each node of the cluster to make sure that each container is running in a Pod with the specified *PodSpecs*. PodSpecs are described in YAML or JSON files with the container's configuration such as number of replicas or port numbers. The Kubelet agent ensures that the specs defined are running and healthy. This management is only possible in containers built by kubernetes [24].

- **Kube-proxy**:

  Kube-proxy act as a network proxy on every node of the cluster. This is what controls the network rules of the nodes allowing for network communication to the Pods from internal and external devices. If the operating system has a packet filtering layer and it's available, then that will be used instead, otherwise kube-proxy will route the traffic on its own [22].

- **Container Runtime**:

  To run containers on machines a software called container runtime is needed. There are many open source container runtimes pre-built that are available to use with Kubernetes, some of them are the Docker, CoreOS rkt or Containerd. It is also possible to implement the Kubernetes CRI (Container Runtime Interface) and use it instead of the pre built ones [15].

Kubernetes Objects:

- **Namespace**:

  One physical cluster allows for multiple virtual clusters which are called Namespaces. When the working environment involves many users in a cluster (several dozen and up), it is advised to use different Namespaces, on the other hand, having a short number of users there is no need to actually make use of them [28].

- **Pod**:

  Pods are the smallest and simplest deployment units in Kubernetes. They run as processes in the cluster and each of them may have one or many containers running applications inside them. Pods specify storage resources, have their own IP address and information about the container behaviour.

- **Service**:

  For Pods to be able to communicate with internal or external devices, they need to have an IP address. Services are an abstracted way of providing this feature, allowing applications on Pods to be exposed in a network. This way there is no need to use unfamiliar service discovery mechanisms. Along with the IP addresses also a Domain Name Space (DNS) name is given to a set of Pods. Services make use of the *Selector* to pick the set of pods to which it's needed to distribute the IPs and DNS [31]

- **Ingresses**:

  Ingresses is another concept in the Kubernetes system that allows access to cluster services by external devices, usually using HTTP. They are also able to provide load balancing, SSL and name-based virtual hosting [18].

- **ReplicaSet**:

  This component is responsible for maintaining the number of replicas defined for a Pod. Replicas exist to guarantee that the applications inside Pods are not lost, ensuring availability. ReplicaSet creates or deletes Pods in order to get the specified number of replicas [30].

- **Deployment**:

  Kubernetes Deployments are objects where the desired state of a Pod and/or ReplicaSets is declared, describing its life cycle, namely how many replicas to run, which image should be deployed or how the updates should be done. The main difference between Deployments and ReplicaSets is the possibility of performing updates by the Deployment. The Deployment can manage ReplicaSets and provide updates to Pods, while ReplicaSets's main goal is to have the specified number of pod replicas running at a time [17].

- **StatefulSet**:

  StatefulSet is the object that deals with the management of stateful applications and the deployments. It scales a set of Pods and deals with the ordering and uniqueness of those Pods. They are very similar to deployments, but a StatefulSet maintains an identifier for each of their Pods. All of the Pods are created from the same specification, but they are all independent. It is advisable to use StatefulSets when using volumes because it facilitates the match of existing volumes with new Pods that have replaced older ones (due to failure, for example). However the downside is that individual Pods in a StatefulSet are known to be more susceptible to fail [32].

- **DaemonSet**:

  This object is responsible for making sure that a specified number of Nodes run a copy of a Pod. When Nodes are added to the cluster, these specified Pods are created in those Nodes. When the Nodes are removed from the cluster, the Pods are garbage collected. Pods created by a DaemonSet will be attached to the DaemonSet, so destroying a DaemonSet will also destroy all the Pods it has created. This is good, for example, when there is a need to have log collection on all Nodes of the cluster [16].

- **Job**:

  This component provides a way of creating Pods. When Pods are created with Jobs, they are expected to successfully terminate after its tasks are done. Every time a Pod completes its task, the job keeps track of it, and when a certain number of them terminates the job will be finished. If a Job gets deleted, all Pods created from that Job will also be deleted [19].

- **Volume**:

  Saving files, logs or other types of data in a container is a problem, firstly because data saved on containers disks are ephemeral and secondly because they may fail at any point in time and if so, data will be lost (even after they are restarted). To solve this problem Volumes may be used. Volumes let all containers in a pod access a common directory, allowing them to have access to the saved data after being restarted. Plus, in a Pod, there is often a need for data exchange between containers, this is also something that volumes can help with [33].

- **Persistent Volumes and Persistent Volume Claims**:

  Kubernetes also offers a Persistent Volume subsystem. This subsystem allows users and administrators to use an API that abstracts how storage is provided and consumed. The API resources are named *PersistentVolume* and *PersistentVolumeClaim*.

  Persistent Volumes are pieces of storage similar to volumes, but the lifecycle is independent of the Pods that uses them. They are resources of the cluster just as a node is.

  Persistent Volume Claims are requests that are made from users in order to get access to the Persistent Volumes storage. Pods in the cluster may request for different levels of resources like CPU and/or Memory, Persistent Volume Claims work in a similar way, where Persistent Volumes are requested in terms of size and mode (mode may be of type *ReadWriteOnce*, *ReadOnlyMany* or *ReadWriteMany*) [29].

2.4.4 Kafka System

In systems where a high number of data flows from point to point continuously, reliability and resilience are very important. Apache Kafka is a distributed event log that is able to help in such situations. This framework follows the publish-subscribe pattern [109], having topics to reference the data.

A typical Kafka system consists of a Kafka Cluster containing one or many servers called **Brokers**. Brokers are responsible for receiving data from *Producers* and forward it to the *Consumers*. Topics are used and may be divided in many **partitions**. In this case, messages are assigned to topics without a specific order, the only way to maintain the messages order is to have one partition. Usually messages are written to partitions in a Round-Robin way. Partitions retain these messages for a *retention period* (seven days by default) or until the maximum amount of messages for the topic is reached. Each partition may be replicated between several message brokers, with a property named *Replication Factor* that gives us a considerable assurance that messages will not be lost if a topic fails due to a broker failure. The only way for data to be lost is having all brokers that contain a certain partition replica fail.

This means that all replicas are identical, therefore to enable all replicas to be synchronized, a leader broker must exist for each partition. All partitions may be synchronized with the one running on the leader broker [85]. On figure 2.23 a system with two Brokers is shown. The first broker has two topics, ”Temp” referring to Temperature and ”Humidity” referring to Humidity data. The former topic is divided in three partitions, each with a replication factor of 1, meaning that these data is not considered crucial and therefore there is no need for replication.

However, the Humidity topic is divided in 2 different partitions, each has a replication factor of 2, meaning that partitions on that topic should be replicated to ensure fault tolerance. In this case, partition 1 from topic Humidity in broker 1 was considered to be the partition leader and a replica of it can be found in broker 2, while for partition 2 of the same topic the leader chosen was on broker 2 and the replica on broker 1 instead.

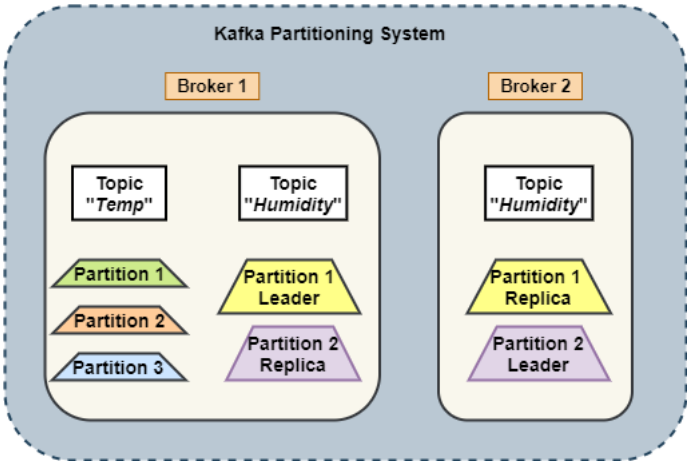


Figure 2.23: Kafka Topics, Partitions and Replication factor [85]

Producers and Consumers:

In the publish-subscribe pattern the nodes/devices that send messages to the broker are called publishers and the ones that receive the data are called subscribers. In Kafka these names are slightly changed to *producers* and *consumers* respectively. Since Kafka ensures no application level acknowledgment, Producers have no perception on whether the message arrives to the destination or not. When Producers send messages, they send the data to a specific partition of a topic [109]. Consumers receive data only if they subscribe to the topic. Many consumers may subscribe to a specific topic to get the same data and all will receive it.

Apache Zookeeper

When talking about Kafka it's impossible not to mention *Zookeeper*. Although Zookeeper and Kafka are independent frameworks, they usually run together. A catch phrase that helps understanding Zookeeper's role is "Zookeeper: Because Coordinating Distributing Systems is a zoo" [61], this phrase means that distributed systems are hard to coordinate, much like a zoo, and zookeeper main function is to help managing these kind of systems. Zookeeper helps Kafka managing all the nodes in the cluster. It is the one responsible for electing the partition leaders. Messages sent to the topic go to the partition leaders first and only after that they are replicated the partitions replicas in other kafka brokers [12]

Messages & Offsets

The data flowing in Kafka always contains a key, value and timestamp. Kafka client library applies a hash function to the message's key to understand to which partition the message is supposed to go. This is the default procedure, but a partitioning function is also an option [12]. Messages usually follow some specific formats, the most common are JSON and XML. Each record in a partition has a designated number of **offsets**. Offsets help consumers keep track of a certain position in a partition, allowing them to come back and review older entries. In case of failure during a read, both Zookeeper and Kafka may persist the consumer offsets in order to ensure that after they come back to life, they are able to continue reading from the point left of (the last offset read) before crashing. In Kafka, consumers offsets are saved in a topic named *"__consumer_offsets"* . Messages must be committed before consumers are able to read them. A message is considered committed when it is written to the partition leader and all in-sync replicas.

3: Requirements Analysis

An IoT system is composed by many small constrained devices, generally sensors, connected with each other, and this number has been increasing along the years. According to Stefan Ferber [53] since the beginning of the Internet's existence there were three big waves, where the first brought big changes to everything related to paper work, such as documents management and their exchanging. The second wave brought online commerce allowing companies to profit from online platforms. Finally the third wave is the IoT, which allows everyone to be connected to everything all the time.

Optimizing data streams from sensor devices to quickly capture a warning sate will allow for faster response to emergency situations. This means having to recognize warning signs and deliver alerts as soon as possible.

With IoT these tasks are done in a dynamic, quick and automated way [120]. Therefore there are many industries and applications already using IoT based systems. Nowadays IoT systems generally follow a similar architecture to the one shown bellow in figure 3.1. Next we present some of these applications, followed by main key requirements. Not every IoT system follows rigidly all of the presented requirements, and some others might even implement all of them and even some more, but we think that the pointed out requirements are the main ones that all IoT systems should have implemented.

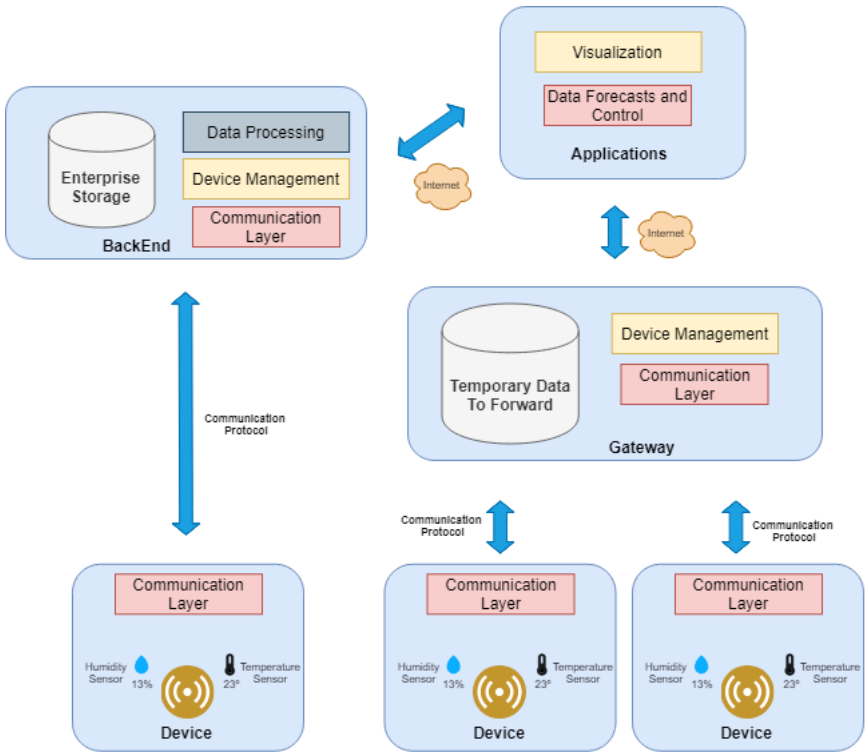


Figure 3.1: Typical IoT system in 2020 [69]

3.1 Use Cases

Healthcare: Many benefits were brought to medicine due to IoT, its now possible to have medical devices sending alerts to patients and help them (self-diagnose).

Nowadays there is even the concept of telemedicine, which means that clinical health care or appointments are possible to be done at distance with the use of many developed technologies such as wireless body area networks, which is composed of a single or multiple independent nodes like sensors or actuators, that are placed in clothes, body or even under the skin allowing for a better control of vital aspects of a patient. This network connects the nodes wirelessly with the use of a star or multi-hop architecture [75, 124].

Smart Cities: According to Albishi et al. [4], by 2025 mega-cities will start to appear. There is a fast growing number of population and infrastructures which will force city's boundaries to increase, crowding neighbouring cities and forming the mega cities. IoT can be used in such cities in many different ways, it is possible to have applications controlling traffic lights or parking lots [46] in order to accommodate user's lifestyle and increase productivity. There is already an European project being developed by Santander, the FP7 project, which aim is to deploy an IoT infrastructure with thousands of devices [4].

Smart Houses: Houses are becoming more and more smart, smartphones are able to interact with shutters, smart TVs or smart bulbs for example, allowing to pull the shutters down or up automatically at a given time of the day, control the smart TVs remotely with the smartphone, choose a desired color for a bulb, or even turn it on and off as pleased. All of this without the need to get up and do it manually. There are many other smart houses application and in the future, others may appear, such as refrigerators helping in stock managing, also, ovens may be able to cook for us someday [66, 36].

Smart Vehicles: There are many brands developing smart cars capable of downloading new features, help in driving and parking. Cars communicate with each other and with remote data centers or the cloud to be able to perform such tasks. A brand that is currently producing these kind of technology is the so famous Tesla company and also Audi [46].

Safety Of Users: Many devices such as temperature sensors, eventually wearable devices or pacemakers, airplane sensors or any other small device capable of capturing some fundamental characteristic must be extremely safe, any malfunction or technical error might happen that would lead to severe consequences. Any exploited vulnerability may lead to danger to life, so these devices should be developed taking into account the kind of dangers they may lead to and prevent it [66].

Smart Agriculture: IoT is also helping farmers. One good example is the use of sensors for irrigation of plants and trees when needed. The irrigation schedule is done automatically by those applications, so when the time to water these plants, there's no need to go one by one watering them, everything is automatic. There are many more applications of IoT in agriculture, such as applications to monitor soil and weather conditions. for example, soil moisture sensors send their information to a gateway, wirelessly, which in turn forwards it to the cloud where farmers may check on their terrain and monitor its conditions. [72]

Consumer IoT: Also there's a distinction from IoT systems and technologies that are developed for consumers and those that are developed for Industries, bringing the so called the *Consumer IoT* and *Industrial IoT*. The Consumer IoT is related to applications where the main focus is on the costumers, such as smart houses, tracking systems or smart watches and many more. these kind of services are very well received by the users because they are very appealing and smooth some small tasks.

Industrial IoT: In Industrial IoT, applications and systems are specifically developed for Industrial environments. This means having heavy machinery involved with all these technologies. these kind of systems will minimize physical work, making some jobs automated, therefore having more productivity. A good example of a practical use of IoT in such environment is the management

of storage systems, checking the stock with a device sending the data to an application that handles how much stock should be available and when to get more of the specific product would help a lot [121].

Another practical example are the use of devices capable of measuring energy, water or natural gas consumption, so that service providers may be able to adjust prices according to the consumption, time of day or season. these features allow users and consumers to automate tasks and reduce costs [120].

3.2 Requirements

- **Interoperability:** The number of devices in an IoT system is very large, which means many manufacturers and companies will be involved in the production of these devices. For devices to be able to cooperate as a group or to be integrated in one, they must be interoperable. For example, if a device is only able to send data through MQTT, it wont be able to communicate with another one that only is able to receive, for example, from CoAP. In this case our proposal should be able to greatly help;

- **Reliability:** An IoT system should always be reliable, because many data is flowing from end to end, and if some of these data is sent to the wrong place, not sent at all or sent with the wrong information, many costs may be applied, and this means not only monetarily. For example, a device that tracks the heart beat of a patient that is in risk of a heart attack failing to send an alert with the correct state of the patient condition could be very problematic;

- **Scalability:** In IoT systems, billions of devices may be running and sending data at the same time. Since this number may increase or decrease the system should be able to acquire or withdraw resources as needed. This allows for better use of resources, because if there is a need for more processing power, for example, the system will respond well without having its performance diminished, but if the number of devices sending data decreases and some resources aren't needed anymore, then they won't be reserved anymore, being able to be used in other tasks [66];

- **Security:** Since a huge number of data is flowing from hop to hop, personal information might be at risk along the way and should be protected. This protection must ensure Confidentiality, Integrity and Availability of the user personal data [66]. This is extremely crucial when data contains personal information such as bank transactions or facial recognition [76]. There are already many technologies developed for security, but many of them don't fit the IoT requirements, such as entering a password everytime access to a device is needed. In the case of IoT, many complex keys to authenticate devices isn't viable, otherwise it would be too complex. New security concepts are needed allowing ease of use at certain levels of privacy and protection [53]. One way to do this is by securing all devices of the network one by one before connecting them to the network. Another way is to make every device to follow standards from regulatory bodies like GSMA [10] or any relevant industry standard;

- **Resilience:** The system may have numerous reasons to fail, from network failure to protocol or implementation errors, anything might cause a failure as there are many devices involved in the system. Therefore, responding to these failures in the best way possible is very important. The system should recover from faults and try to avoid them. This is divided in two different modes. In the first one, the reactive mode, the system detects and recovers from faults as they happen. On the second mode, the proactive mode, systems save and monitor their own state. Before an undesired state is detected the system adjusts to prevent it [110];

- **Availability:** These kind of system is always running all day, if any update or adjustment is needed, stopping it is not a reliable option, the system must be running constantly. There must be mechanisms that allow the system to be available no matter what. A good option for this would be the use of micro-services and Kubernetes to manage them without the need to stop the entire system;

- **Device Management**: Changes in the devices collecting the data should be available, for example, if we have a device monitoring the temperature of a fridge, if it's Summer time, we

may want temperature alerts on a certain level of degree, but if it's Winter time, this level may change, we don't want to have to go to all the devices and make the change one by one, it is essential that devices can receive at least some orders or change some configurations;

- **Real time Transmission:** In some case, the data collected by sensors should be transmitted in real-time. For example, a sensor capturing data of the water level on a dam at each second might be too slow, maybe the need for the information is more urgent, perhaps if the rate isn't less or equal to 5ms the water would overflow [46, 36];
- **Robust connectivity**: One of the biggest problems or difficulties in IoT systems is that devices used in such systems are often very small with very low computing power and energy capacity, plus the network links are many times weak (the internet connection may be unstable). To improve this situation, more efficient devices with more energy capacity are needed. [53];

- **Big Data/Analytics**: Big data means big amount of data, with lots of variety and velocity. Big data is directly related to storage struggles. Such amount of data is very hard to store and to analyse because we don't usually need all of the data. The biggest problem here is to have good analytics and semantics to ensure that from all the data being received, the relevant parts are identified and collected properly. For this, there are already some mechanisms in place to filter irrelevant data [53].

4: Fault-Tolerant and Interoperable IoT architecture

In this chapter we present a fault tolerant and interoperable IoT architecture. The proposed architecture is able to provide a *resilient, ensuring availability, scalability, fault tolerance, interoperability and simple management* system. It was designed to satisfy all requirements exposed in section 3.2.

First of all, we describe the system architecture as a whole which can be seen in figure 4.1.

The proposed IoT solution consists on a framework capable of receiving and processing data that is sent using one of the following protocols: *MQTT*, *CoAP*, *REST* and *LoRaWAN*. *LoRaWAN* was a protocol that we implemented using LMIC library, unfortunately in order to perform tests on this protocol, we would have to make the experiments on an urban or rural area, which was un-doable due to Corona Virus pandemic. Additionally, *LoRaWAN* protocol was not implemented inside the system because LoRa requires a special ship to receive LoRa packets. In our implementation a gateway was used. This gateway is composed by a Draguino LoRa Shield on top of a Raspberry Pi.

The system stores data in Kafka and uses kubernetes cluster. It is based on micro-services implemented by docker containers that runs inside kubernetes pods. Each pod in the Kubernetes cluster runs only one container because Kubernetes allows scaling of pods, not containers. In order to scale every individual container, each pod must contain only one container, otherwise all containers in a pod would be scaled.

The proposed architecture is based on the following containers: the *MQTT Broker*, *CoAP Server*, *REST Server*, *Kafka system* with *Zookeeper*, *Proxy* and *Controller*. All of these containers/components are inside of the Kubernetes cluster and are described below.

Besides these components there are also two more participants in the system. From outside of the cluster, we have *Producers* who are devices that generate/collect data and send it to the system, and the *Receivers* that will subscribe data feeds and consume it.

In order to handle different protocols receiving and sending data on our system we created a Kafka solution. Every time data is requested by any protocol, the data sent is the data that was previously stored in Kafka, this way a protocol A requesting data that only was sent using protocol B may be received, moreover these data is persisted and we may even request for older messages or a bunch of them using a bulk command (explained later).

Internally, the *Librdkafka* library was used to establish the communication with Kafka. The complexity of using this library within each broker and server was high due to the fact that it would have to be compiled along with all of them (Mosquitto, CoaP Server and REST).

In order to solve this problem we created the *Proxy* component which is responsible to receive data from the brokers/servers and send it to *Kafka*.

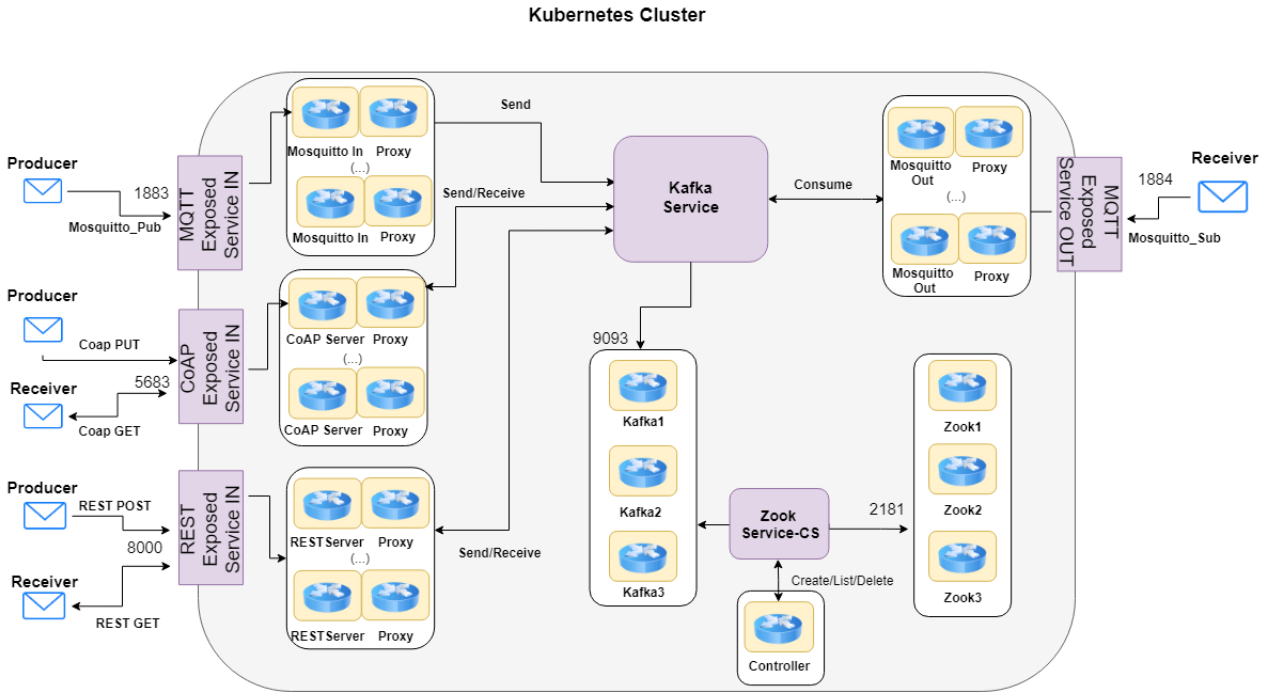


Figure 4.1: System Architecture

4.1 Components of the architecture

MQTT Broker:

In order to handle the MQTT protocol, our proposal uses the Mosquitto [1] framework as MQTT Broker. This broker was incorporated in the kubernetes cluster. There are two pods running instances of the Mosquitto, one that deals with incoming data into the cluster *Mosquitto In* and another one that deals with MQTT data going out of the cluster *Mosquitto Out*.

Since our objective is to prevent data loss, all data is secured in Kafka, meaning that every client that wants data, must get it from Kafka. To send data to Kafka our Proxy is used with the Librdkafka library. Both pods running the Mosquitto In and the Mosquitto Out also have our Proxy running inside of them running a socket server that listens to connections on port 5151. When requests arrive to the socket server a new thread is created to handle the request. This handler is the one that uses the Librdkafka to perform actions on Kafka.

When a Producer sends data, that data is received by the Mosquitto In instance that uses our Proxy to send it to Kafka where data is stored.

Receivers that want data subscribe to the Mosquitto Out instance. In this case (MQTT protocol) our *Proxy* is responsible for continuously publish data stored on Kafka to the Mosquitto Out. Mosquitto In listens to the port 1883 and Mosquitto out listens to the port 1884.

If only one instance of Mosquitto was used, that Mosquitto would be listening on a single port for incoming and outgoing data, which means that a Producer would send data to the Mosquitto on a chosen port, for example 1883, and that data would be sent to Kafka for storage, which in turn would be published to the Mosquitto again in port 1883 creating a loop where mosquito sends data to Kafka and vice versa. Having two instances allows data to flow from one end of the cluster to the other with no loops involved.

The data flow of MQTT packets in the system can be seen in figure 4.2.

[1] `https://github.com/eclipse/mosquitto`

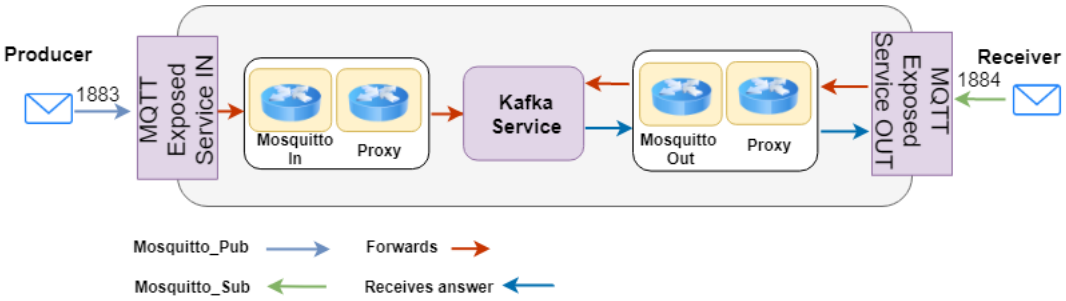


Figure 4.2: MQTT data flow

CoAP Server:

This server deals with incoming and outgoing CoAP data. When a Producer needs to send data to our system using CoAP protocol, that data is sent to this server, which then forwards it to Kafka using our proxy, much like the Mosquitto broker.

Similarly to the Mosquitto Broker, Proxy is running alongside the CoAP server in the same pod and it runs a socket server that listens to TCP requests on port 5151. When Producers send data to the CoAP server, the server sends the received request to this socket server. For each request received the socket server creates a new thread to handle the request. This handler calls the Librdkafka library to perform actions on Kafka (in this case to store data). The same thing happens when the request is sent by a Receiver to *get* data. The request is sent to the socket server creating a new thread to request data from Kafka.

Since in this case the pattern used is request-response, there is no need to have Kafka continuously sending the data stored again to CoAP server, which means there is no possible loop. Consequently, in this case only one instance of the server is needed.

POST requests are used to send data to CoAP and GET requests to ask the server for data. When a Producer uses CoAP to send data to the cluster, the pod running CoAP server receives that request and sends it to Kafka using the Proxy. When a Receiver wants data, a GET request is sent to the CoAP server inside the cluster that uses the Proxy to get that data from Kafka.

The implemented server is based on *libcoap* [2] library. It includes resources to interact with the Proxy that is responsible for getting data from Kafka. To send POST requests the client must use the following command:

```
$ echo <mymessage> | coap-client -m put coap+tcp://[<address>]/<topic> -f -
```

There is a limitation concerning the transport layer protocol. According to the CoAP specifications, it is a protocol able of working with TCP and UDP protocols. However, in our system it works only with TCP requests. This is a limitation that comes from the use of Bare Metal load balancer [6] in kubernetes cluster.

In the future work this should be changed to allow UDP requests. A possible solution may be the use of other load balancer to handle incoming traffic. The same thing happens with Receivers, only TCP requests are handled.

In terms of protocol operation, there are two different options to perform GET requests, the normal way which uses the command

```
$ ./coap-client -m get coap+tcp://[<address>]/<topic>
```

to request for the last message, and the bulk way that requests for the last given number of messages from the system (counting from the last message received on Kafka), e.g.

```
$ ./coap-client -m get coap+tcp://[<address>]/iotbulk -e 3
```

which will return the last 3 messages that were received on Kafka.

[2] https://github.com/obgm/libcoap

REST Server:

Very similarly to the CoAP server, the REST server is the one responsible for receiving data sent via REST protocol. The implemented REST server is based on *ulfius* library [3].

When a Producer sends data to the REST server, that is running on a specific pod along side the Proxy, it forwards that data to the Proxy, which uses a socket server, listening for TCP connections on port 5151, and threads to handle the requests. This handler uses Librdkafka to perform the requested actions on Kafka (POST or GET).

In this case, the following command is used to send data to the system.

```
$ ./simple_client <address> <port> POST "/mibroker/iot/" "name=mymessage"
"Content-Type: application/x-www-form-urlencoded"
```

This command will post the message "mymessage" on the resource named "iot" to the REST server using the given address and port. The resource "iot" is used as topic when sending the data to Kafka because Kafka doesn't work with resources.

In order to retrieve data from the system, Receivers may use one of the following options: the normal or bulk options.

The normal option or command requests for the last message introduced in Kafka, while the bulk command requests for the last given number of messages introduced in Kafka.

Both commands are equivalent if a bulk size of one is considered. Otherwise, they will differ from the quantity of data that will be returned.

The normal command is expressed as:

```
$ ./simple_client <address> <port> GET "/mibroker/iot"
```

The bulk option or command, that allows requesting the server for the last n messages (e.g., 3) can be expressed as:

```
$ ./simple_client <address> <port> GET "/mibroker/iot:3"
```

Kafka Cluster:

As explained in section 2.4, Kafka is a system that follows the publish-subscribe pattern and allows message delivery to be scalable, replicated between many Kafka brokers which contributes to fault tolerance and parallel consumption using **partitions**, **offsets** and **replication**.

Being a publish-subscribe system, messages are distinguished with the use of topics. These topics must be created taking into account the replication factor and the desired number of partitions, which will be used to provide fault-tolerance and high consuming performance over data inside the system. Each topic is created by issuing the following command:

```
$ ./kafka-topics.sh --create --zookeeper <zook_addr> --replication-factor
<num_rep_factor> --partitions <num_part> --topic <topic>
```

These topics are divided in partitions and each partition has internal offsets. The partitioning allows replication of each partition between the available Kafka brokers in a Kafka cluster, hence allowing parallel processing. The offset is used as an identifier of each message in a partition. The anatomy of a topic can be seen in figure 4.3 where three different partitions for a single topic with multiple offsets are represented. When Producers send data to that topic the most recent data is set with a higher offset value (identified), so older messages are the ones with the lowest value [116].

[3] https://github.com/babelouest/ulfius

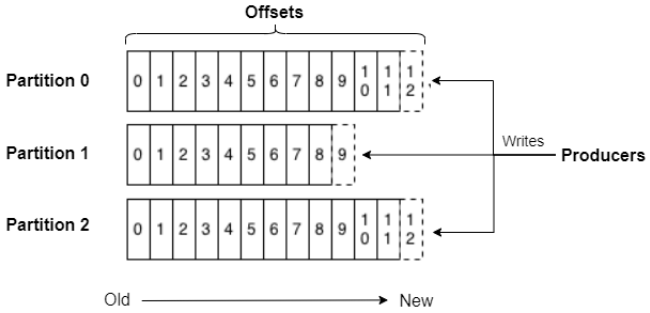


Figure 4.3: Kafka topic anatomy

Our system was designed to be scalable and fault-tolerant. To achieve this, the number of Kafka servers or brokers available, at an instant, must be controlled so that if one of them fail the system would still deliver messages. Hence in our system we created a cluster with 3 Kafka brokers. This number is scaled very easily, due to the Kubernetes capabilities.

Each Kafka broker is used to store the incoming data and keep that data available to be used for a *retention period*. This retention period is controlled by the administrator, through the configuration files, and it assumes, by default, a value of 7 days. It can be changed with the command:

```
$ ./kafka_2.12-2.5.0/bin/kafka-topics.sh --zookeeper <address> --alter
--topic <topic> --config retention.ms=1000
```

This command will change the retention period to 1 second, meaning that messages will be saved in Kafka in the given topic for 1 second and then they will be deleted.

When replication is used, Kafka nominates a leader for each partition. This leader will receive all messages targeted to that partition and then replicates it to the other replicas. If the leader partition fails for any reason, a new leader election begins and the elected partition takes over. Besides allowing fault tolerance, the use of partitions also allows multiple consumers to read from a topic in parallel [116] granting faster throughput processing. In figure 4.4 the concept of partitions is illustrated.

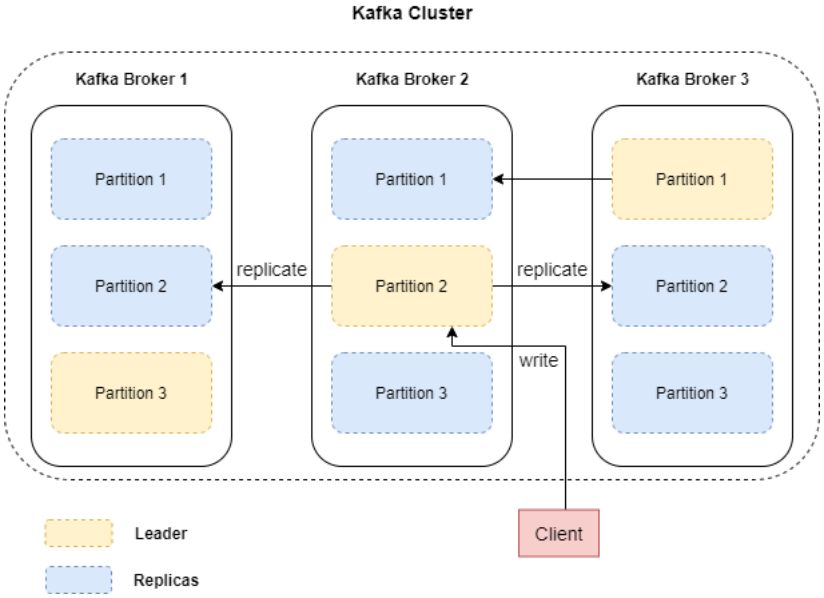


Figure 4.4: Kafka Partitions

Regarding the consumers, they are able to read data starting from a chosen offset. Each message in the Kafka system is identified by a tuple defined by the message topic, partition and offset within the partition.

Zookeeper Cluster:

Since Kafka works in cluster fashion, there is a need for server coordination. Zookeeper is responsible for helping Kafka instances on its coordination. The zookeeper is the brain that does the election of the Kafka leader. Without zookeeper, Kafka could not operate well.

If only one zookeeper instance is present in the cluster and it crashes for any reason, Kafka instances would become not coordinated and they would crash leading to a failure of the whole system.

To avoid possible crashes of single Zookeeper node, the same strategy that is used in Kafka must be used here. If Zookeeper instances are replicated and if one instance crashes there are still other instances to control Kafka cluster. Hence there is in fact a cluster of Zookeepers in the system instead of a single instance.

Services & Exposed Services:

The proposed solution makes use of many components and those components receive data from outside of the cluster and send it to other components, but this link is not so trivial.

In order to allow the communication between components, services must defined to expose the functionalities of each pod to outside the cluster, using IP addresses or hostnames.

However, general Services aren't visible from outside of the Kubernetes cluster. Exposed services have external IP addresses assigned to them. The assignment of external IP addresses in the cloud is made by *LoadBalancers* and/or *Ingresses* but since our environment is on bare metal, we used a framework called *metalLB* [6] (an open-source LoadBalancer built to work on bare metal kubernetes systems) to have external IP addresses on our external services.

Internal Services were created for each component. They are used to establish the connection between all components inside the kubernetes cluster.

These services are specified in YAML files created by us. They are available in the GitHub page [4] and are also described in section 4.2.

Besides Internal services, External services were also created. They were created by using a specific command in the terminal:

```
$ kubectl expose deployment <deployment_name> --port=<port_number(s)>
--type=LoadBalancer --name=<service_name> -n <namespace>
```

This command was used to create external services for Mosquitto In, Mosquitto Out, CoAP server and REST Server. It configures TCP connections on ports 1883 for Mosquitto In, 1884 for Mosquitto Out, 5683 for CoAP, 8000 for REST and 7000 for WebSocket connections. All ports and respective pods are visible in figure 4.1.

Clients trying to send or receive data from any of these components use the external IP address provided by these external services. The presented command exposes the service on a deployment, meaning that all instances of that deployment become endpoints of the external service created (many instances may exist if the number of replicas is more then one). Data coming to the external service is then redirected for one of these endpoints.

Pod	Ports
Mosquitto In	1883
Mosquitto Out	1884
CoAP	5683
REST	8000
WebSocket	7000

Table 4.1: Summary of all ports used for each Pod

[4] https://github.com/jsoares11/MasterThesisExperiences/tree/main/YAML%20files

Proxy:

The proposed architecture builds an intermediate computing layer which will serve as an abstraction hiding different protocol implementations which we called *Proxy*. Proxy is the one responsible for the communication between every server or broker with the Kafka system. By allowing this communication, it is possible to use Kafka to store the incoming data, and also to request for older data when receivers ask for it. This communication was build using sockets and a library named *Librdkafka* that can be found on the following url: `https://github.com/edenhill/librdkafka`.

When a pod with a server or broker runs in the kubernetes system, it also runs our proxy, so in fact there are many proxy instances as servers or brokers in the system. Resuming, our Proxy is a middleware that allows communication between any server/broker and Kafka. Hence, allowing devices that use different protocols to interact with each other, all implemented protocols can be translated between them (MQTT, CoAP, REST).

The Proxy is composed by four different threads started by *proxy.c*. The threads are: the *producer.c*, *kafka_socket_server.c*, *kafka_publisher.c* and *mqtt_publisher.c*. Each of them have their own responsibilities inside the proxy which are described next.

To make sure that data isn't lost in case the proxy isn't able to handle the throughput (data being send faster than it is being forwarded) a linked list is used in order to form a queue of messages that will be sent to Kafka. If a message can't be sent because a previous message is still being processed it will be waiting in the queue (linked list). Figure 4.5 represents the proxy architecture.

- ***producer.c:*** When a Producer sends data to our system, it sends the data to an external service that forwards it to the desired server or broker. For example, a Producer sending data using MQTT protocol sends data to the *"mosquitto-service-in"*, because this is the external service that grants *Mosquitto In* external access. Inside of the pod *Mosquitto In* there are three separated programs running, a *Mosquitto* broker accepting incoming data, a *Mosquitto_client* subscribing to that same Mosquitto broker and our *Proxy*. When data arrives to Mosquitto IN, it arrives in the Mosquitto broker, that forwards it to anyone subscribing, so the Mosquitto_client receives the data and outputs it to the **stdin**. Our Proxy uses the thread *producer.c* and the *PIPE* command in linux to keep listening to the output of the Mosquitto_client stdin and also receive the data. Data is then placed on the queue to be saved in Kafka.

- ***kafka_socket_server.c:*** Starts a socket server listening and accepting connections from the CoAP and REST servers on port 5151. When a server connection request is accepted, the *kafka_socket_server* calls a *connection_handler* thread to deal with it, this way *kafka_socket_server* doesn't take much time to answer other requests because the previous requests are being processed by the *connection_handler* in other threads. Requests assumes only two types, *consumer* or *producer* (mode "C" or mode "P" respectively). The handler makes a call to *kafka_interface.c* where the *librdkafka* library is used to send data to kafka depending on the mode used. If the mode used was the consumer mode, kafka will return the requested messages, otherwise the data sent to kafka will be stored.

- ***kafka_publisher.c:*** This thread handles the queued data. Its responsibility is to check if there is data in the queue, if so *kafka_interface.c* is called and used to publish the data in Kafka with the use of Librdkafka library.

- ***mqtt_publisher.c:*** In this thread the "Online" mode is handled (mode "O"). It is responsible for continuously consume data from Kafka and publishing it to Mosquitto Out, allowing subscribers to receive data from the system.

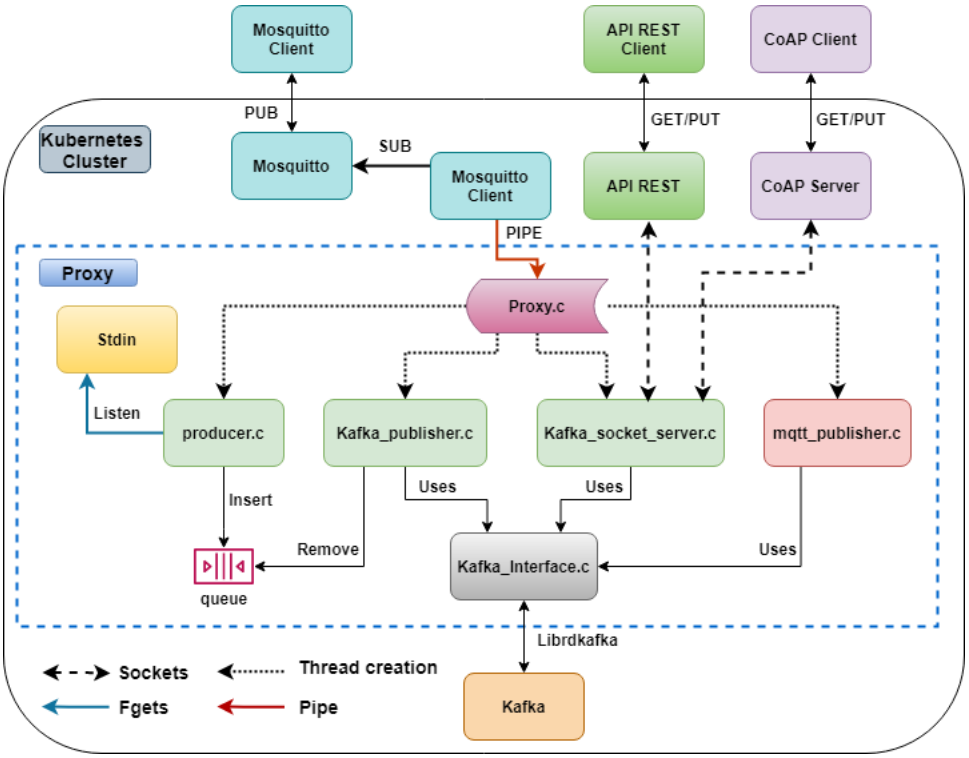


Figure 4.5: Proxy Architecture

4.2 Kubernetes Environment

Previously, we have seen that having the cloud working along side with IoT systems helps them satisfy many IoT requirements but even though this would be a good path to explore, it was decided to use a *bare metal* Kubernetes system instead (instead of using a well known cloud infrastructure, we decide do build our own cloud). This decision was due to the prices of the cloud.

In order to test the proposed system, we would need many instances, at least one to work as the master node, and three other nodes to work as slaves. Besides this, we are going to need two more nodes to make requests to the cluster, meaning that we would need a minimum of 6 machines to be running while implementing our system and while running experiments on it.

There is another factor that had a big impact on this decision, since kubernetes requests machines with at least two CPUs, the use of the simplest machines on cloud providers wouldn't be enough to run it, so the machines reserved to the system would be more expensive. This would bring considerable high charges. With a *bare metal* system, we can have a kubernetes system running on any desired set of machines for free, with the downside of not having access to all the previously mentioned features that the cloud provides.

In this work, a Kubernetes Cluster composed by 5 nodes, one master node and four slave nodes was created. To install the cluster we had to make some actions which are described in appendix A. Here, steps we took in order to install the Kubernetes cluster on dual-core devices running on Ubuntu 18.04 operating system are demonstrated. It's important to notice that although these steps allow kubernetes to be installed there are scripts available on our GitHub page [5] to facilitate the installation.

After installing the kubernetes cluster, we need to specify some *YAML* files that are used to create/running our components in the cluster.

These *YAML* files contain the desired configurations for each component. In order to build our proposal, we created a set of *YAML* files, that is composed by: *kafka.yaml*, *zookeeper.yaml*, *mosquitto-in.yaml*, *mosquitto-out.yaml*, *coap.yaml*, *rest-api.yaml*, *controller.yaml* and *pv-rwo.yaml*.

The container image used is the same for all components and is named *"josesoares/proxy"*.

[5] `https://github.com/jsoares11/MasterThesisExperiences`

This image is available on *docker hub* [73] [6].

Although having the same image, the behaviour is not the same for all components because inside each of these files there is a specific command that executes the desired functionality of the instance. The command calls one of four scripts: *"start-mqtt-rep3"*, *"start-coap-rep3"*, *"start-rest-rep3"*, *"controller"*. These scripts are available in this Thesis GitHub page [7]

Additionally, two types of components/services were defined: stateless and stateful services. Zookeeper and Kafka were defined as StatefulSets (stateful components/services) because we wanted Zookeeper and Kafka to be able to keep their state across different states of the cluster. This is also the reason why they are both the only ones that claim Persistent Volumes. On the other side we have Deployments (stateless components/services) that do not keep their state. In our implementation, all protocol specific server uses Deployments to run inside the cluster. Since data is kept by kafka, keeping specific protocol server state is not important for global achievements of the system.

A brief description of each YAML file is presented below. All of these files can be found in the repository of this thesis.

- ***kafka.yaml:***

  The purpose of this file is to define the configurations of Kafka cluster that will run inside kubernetes cluster as part of our system architecture. This definition includes a specification of services that are associated with it. A *kafka-hs* service is defined. The "hs" stands for *Headless Service* [31] which is the type of service used when there is no need for load-balancing, these kind of services don't have a cluster IP allocated, instead the name of the StatefulSet is used to associate the Service with the StatefulSet. This is done by using a field named *"selector"* in the Service definition with the name of the StatefulSet to be associated with, in this case "kafka" which is visible in figure 4.6. DNS is then able to understand that data coming to the headless service must be mapped to an available pod from the Kafka StatefulSet. The kafka-hs service listens to TCP connections on port 9093.

  The Kafka cluster is defined as a *StatefulSet*, as we can see in the field named *"kind"*, shown in 4.7. The field *"name"* defines the name of the StatefulSet that is used by the Service selector. Since we are interested in fault-tolerance, the number of Kafkas in the cluster should be more than one, so the *"replicas"* field is set to 3. To have access to Persistent Volumes a volume claim is defined in the field *"volumeClaimTemplates"* as visible in figure 4.7. Here, the access mode is defined as *ReadWriteOnce*, meaning that the volume can only be mounted as read-write by a single node, and some resources are requested (100Mi - one hundred MebiBytes).

  In order to execute Kafka inside the container a specific command must be used. The command used is the following:

  ```
  $ kafka_2.12-2.5.0/bin/kafka-server-start.sh kafka_2.12-2.5.0/
  config/server.properties
  ```

  This command starts Kafka using the Kafka configurations inside of the file *server.properties*. *Kafka.yaml* file uses this command on a field named *"command"* that is visible in figure 4.7.

[6] https://hub.docker.com/repository/docker/josesoares/proxy

[7] https://github.com/jsoares11/MasterThesisExperiences/tree/main/YAML%20files/scripts.

```
---
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      app: kafka
    name: kafka-hs
  spec:
    ports:
    - name: kafkaport
      port: 9093
      protocol: TCP
    selector:
      app: kafka
```

Figure 4.6: Excerpt of *kafka.yaml* with the configurations of the Service

```
---
- apiVersion: apps/v1
  kind: StatefulSet
  metadata:
    labels:
      app: kafka
    name: kafka
  spec:
    replicas: 3
    (...)
    containers:
    - command:
      - sh
      - -c
      - "exec kafka\_2.12-2.5.0/bin/kafka-server-start.sh
        kafka\_2.12-2.5.0/config/server.properties"
        (...)
    image: josesoares/proxy
    (...)
    volumeClaimTemplates:
    - apiVersion: v1
      kind: PersistentVolumeClaim
      metadata:
        creationTimestamp: null
        datadir
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 100Mi
```

Figure 4.7: Excerpt of Kafka.yaml with the configurations of the StatefulSet

- ***zookeeper.yaml***

  The *zookeeper.yaml* file is very similar to the *kafka.yaml* in the sense that it also defines a headless service named *"zk-hs"*, a StatefulSet named *"zk"*, a Persistent Volume Claim and three replicas. Besides that, it also defines a normal service named *"zk-cs"*. The *"cs"* stands for *client service* which is a normal service meaning that it has an assigned IP address, but

that address is internal (only valid inside the cluster).

The headless service accepts TCP data on ports 2888 and 3888 and the client service also listens for TCP connections on port 2181. Port 2888 is used by zookeepers to exchange information between them, while port 3888 is reserved for the process of leader election. The port 2181 is used by Kafka instances to connect to a running Zookeeper, this is defined in the server_properties file, the Kafka configuration file. All these configurations are presented in figures 4.8 and 4.9.

To run the zookeeper inside the cluster, the following command was used:

```
$ ./kafka_2.12-2.5.0/bin/zookeeper-server-start.sh kafka_2.12-2.5.0/
config/zookeeper.properties
```

It starts the zookeeper with the configuration file *zookeeper.properties*, where some parameters such as client/connection timeout, maximum number of connections or maximum number of clients at the same time are defined.

```
---
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      app: zk
    name: zk-hs
  spec:
    ports:
    - name: server
      port: 2888
      protocol: TCP
      targetPort: 2888
    - name: leader-election
      port: 3888
      protocol: TCP
      targetPort: 3888
    selector:
      app: zk
---
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      app: zk
    name: zk-cs
  spec:
    ports:
    - name: client
      port: 2181
      protocol: TCP
      targetPort: 2181
    selector:
      app: zk
```

Figure 4.8: Excerpt of zookeeper.yaml with the configurations of its Services

```
---
- apiVersion: apps/v1
  kind: StatefulSet
  metadata:
    labels:
      app: zk
    name: zk
  spec:
    replicas: 3
    (...)
    containers:
    - command:
      - sh
      - -c
      - ./kafka_2.12-2.5.0/bin/zookeeper-server-start.sh
        ./kafka_2.12-2.5.0/config/zookeeper.properties
    (...)
    image: josesoares/proxy
    (...)
    volumeClaimTemplates:
    - apiVersion: v1
      kind: PersistentVolumeClaim
      metadata:
        creationTimestamp: null
        name: datadir
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 100Mi
```

Figure 4.9: Excerpt of zookeeper.yaml with the configurations of the StatefulSet

- ***mosquitto-in.yaml***

  The Mosquitto In YAML file contains the configuration to run a Mosquitto Broker with a set of configurations, like listening ports or connection timeout. Associated with Mosquitto itself, there is a service named "mosquitto-cs-in" that listens for TCP connections on port 1883 (default port for MQTT communication) as shown in figure 4.10.

  The Mosquitto In YAML also defines its content as a deployment (figure 4.11). This deployment is named *"moquitto-deployment-in"*.

  Inside the image *josesoares/proxy* there is a script named *"start-mqtt-rep3"*, which receives five parameters, the mode (*"in"* or *"out"*), the topic that is going to be used, the Kafka and Zookeeper addresses, and the MQTT port that will be used. In this case the mode is *"in"*, telling our Proxy that the container will run a Mosquitto In. To use the script the following command was used:

  ```
  $ ./start-mqtt-rep3 in iot Kafka-hs.kafka.svc.cluster.local:9093
  zk-cs.kafka.svc.cluster.local:2181 1883
  ```

  This command will start a Mosquitto broker on the former container in the given MQTT port and will also start the Proxy.

```
---
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      app: mosquitto-in
    name: mosquitto-cs-in
  spec:
    ports:
    - name: clientin
      port: 1883
      protocol: TCP
      targetPort: 1883
    selector:
      app: mosquitto-in
```

Figure 4.10: Excerpt of mosquitto-in.yaml with the configurations of the Service

```
---
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: mosquitto-deployment-in
    labels:
      app: mosquitto-in
  spec:
    replicas: 1
    (...)
    containers:
    - name: mosquitto-in
      command:
      - sh
      - -c
      - ./start-mqtt-rep3 in iot kafka-hs.kafka.svc.cluster.local:9093
        zk-cs.kafka.svc.cluster.local:2181 1883
    (...)
    image: josesoares/proxy
```

Figure 4.11: Excerpt of mosquitto-in.yaml with the configurations of the Deployment

- ***mosquitto-out.yaml***

  This YAML file is very similar to the previous one (mosquitto-in.yaml). The main difference is the mode and the port that is used. This instance is listening for TCP connections on ports 1884 and a service named *"mosquitto-cs-out"* was created 4.12.

  This component is deployed as a Deployment with the name *mosquitto-deployment-out* 4.13.

  To run the instance we used the command:

  ```
  $ ./start-mqtt-rep3 out iot kafka-hs.kafka.svc.cluster.local:9093
  zk-cs.kafka.svc.cluster.local:2181 1884
  ```

  This command uses the same script used for mosquitto-in.yaml named *"start-mqtt-rep3"* and receives the same arguments, but the mode and port used was *"out"* and 1884, respectively.

  Since mosquitto allows to feed data in near real-time from our system, and to offer more ways to retrieve data in real-time, a websocket service was also included in this component. This websocket is listening for incoming connections in port 7000 and when new data arrives

at mosquitto out, this websocket server also receives that data and forwards it to all clients registered on it.

```
---
 - apiVersion: v1
   kind: Service
   metadata:
     labels:
       app: mosquitto-out
     name: mosquitto-cs-out
   spec:
     ports:
     - name: clientout
       port: 1884
       protocol: TCP
       targetPort: 1884
     - name: gwsocket
       port: 7000
       protocol: TCP
       targetPort: 7000
     selector:
       app: mosquitto-ou
```

Figure 4.12: Excerpt of mosquitto-out.yaml with the configurations of its Services

```
---
 - apiVersion: apps/v1
   kind: Deployment
   metadata:
    name: mosquitto-deployment-out
     labels:
       app: mosquitto-out
   spec:
     replicas: 1
     (...)
     containers:
      - name: mosquitto-out
        command:
        - sh
        - -c
        - ./start-mqtt-rep3 out iot kafka-hs.kafka.svc.cluster.local:9093
          zk-cs.kafka.svc.cluster.local:2181 1884
     (...)
     image: josesoares/proxy
```

Figure 4.13: Excerpt of mosquitto-out.yaml with the configurations of the Deployment

- ***coap.yaml***

  The coap.yaml file contains the configurations to build a pod with a container running a CoAP server and the Proxy. To do this, the following command was used:

  ```
  $ ./start-coap-rep3 iot kafka-hs.kafka.svc.cluster.local:9093
  zk-cs.kafka.svc.cluster.local:2181
  ```

  This command uses a script named *start-coap-rep3* that receives as arguments the CoAP resource (that will be used as topic in Kafka), the Kafka and Zookeeper addresses. The use of this script can be seen in figure 4.15. In the same figure is also visible that a Deployment was used. To allow communication between this and other components a service was defined with the name *"coap-cs"*, which listens to TCP connections on port 5683. This component was configured to also listen to UDP connections, but these packets would not arrive to the pod. This is still a limitation of our system and should be fixed in a future work. These configurations can be seen in 4.14.

```
---
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      app: coap
    name: coap-cs
  spec:
    ports:
    - name: clientinudp
      port: 5683
      protocol: UDP
      targetPort: 5683
    - name: clientintcp
      port: 5683
      protocol: TCP
      targetPort: 5683
    selector:
      app: coap
```

Figure 4.14: Excerpt of coap.yaml with the configurations of its Service

```
---
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: coap-deployment
    labels:
      app: coap
  spec:
    replicas: 1
    (...)
    containers:
     - name: coap
       command:
       - sh
       - -c
       - ./start-coap-rep3 iot kafka-hs.kafka.svc.cluster.local:9093
         zk-cs.kafka.svc.cluster.local:2181
    (...)
    image: josesoares/proxy
```

Figure 4.15: Excerpt of coap.yaml with the configurations of the Deployment

- ***rest-api.yaml***

  The rest-api.yaml file contains the configurations needed to start a pod with a container running a REST server and the Proxy. These configurations are described here and can be seen in figures 4.16 and 4.17. The defined service listens TCP connections on port 8000 and is named *"rest-cs"*. The defined Deployment is named *"rest-deployment"*. The following command was used to start both the REST server and Proxy:

  ```
  $ ./start-rest-rep3 iot kafka-hs.kafka.svc.cluster.local:9093
  zk-cs.kafka.svc.cluster.local:2181
  ```

  This command uses a script named *start-rest-rep3* that receives as arguments the resource that should be used (which will be used as topic in Kafka), the Kafka and Zookeeper addresses to be used, by order.

```
---
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      app: rest
    name: rest-cs
  spec:
    clusterIP: None
    type: LoadBalancer
    ports:
    - name: clientin
      port: 8000
      protocol: TCP
      targetPort: 8000
    selector:
      app: rest
```

Figure 4.16: Excerpt of rest.yaml with the configurations of its Service

```
---
 - apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: rest-deployment
     labels:
       app: rest
   spec:
     replicas: 1
     (...)
     containers:
      - name: rest
        command:
        - sh
        - -c
        - ./start-rest-rep3 iot kafka-hs.kafka.svc.cluster.local:9093
          zk-cs.kafka.svc.cluster.local:2181
     (...)
     image: josesoares/proxy
```

Figure 4.17: Excerpt of rest.yaml with the configurations of the Deployment

- ***controller.yaml***

  The controller.yaml file is used to define the configuration component. In our implementation, this controller does not require a specific service. It is composed only by a Deployment and is named *"controller-deployment"*. The configurations of this Deployment may be seen in figure 4.18.

  The controller component was configured to have only one instance and to listening for administrator commands. This means that the container is available to be used by advanced users or admins. To access this container the following command may be used on the cluster:

  ```
  $ kubectl exec -n kafka --stdin --tty podname -- /bin/bash
  ```

  Once inside, it gives access to all micro-services defined by the proposed architecture and running over the kubernetes cluster.

  Inside this component, there is a script named *"controller"*, which allows to *list*, *delete* or *create* a topic from Kafka.

  To use the script the following arguments must be given: *method* ("create", "list" or "delete"), the *topic* to be used, the *replication factor*, the *number of partitions* and the *zookeeper's address* to use.

```
---
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: controller-deployment
    labels:
      app: controller
  spec:
    replicas: 1
    (...)
    containers:
     - name: controller
       command:
       - sh
       - -c
       - tail -f /dev/null
    (...)
    image: josesoares/proxy
```

Figure 4.18: Excerpt of controller.yaml with its configurations

- ***pv-rwo.yaml***

  Lastly, the pv-rwo.yaml file is responsible for creating the Persistent Volumes that are used by the Kafka and Zookeeper clusters. Sixteen Persistent Volumes were created to allow any component of the system to claim them if needed. The name of the Persistent Volumes created vary from *"pv001"* to *"pv016"*, and each of them is configured to provide 500Mi of storage, have access mode *"ReadWriteOnce"* and have the contents on the path *"share/pvXXX"* where "XXX" represents the number of the persistent volume in question. After applying this YAML file the Persistent Volumes are created and the components may use Persistent Volume Claims to request their use. All of these configurations can be seen in 4.19.

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv016
spec:
  capacity:
    storage: 500Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/share/pv016"
```

Figure 4.19: Excerpt of pq-rwo.yaml with its configurations

All these YAML files are available at `https://github.com/jsoares11/MasterThesisExperiences/tree/main/YAML%20files`. They allow anyone to replicate our system and/or adjust some values to configure the cluster to have more containers (components) or replicas of a specific component.

In terms of loading these configurations (YAML files), the command *"kubectl apply"* is used.

```
$ kubectl apply -f <filename.yaml>
```

When this command is applied all configurations are loaded to a default namespace. In order to change the namespace, the command must be change to include the *"-n"* flag and the name of

new namespace. Note that there is no default mechanisms to establish the communication between different namespaces.

```
e.g., $ kubectl apply -f  filename.yaml -n kafka
```

Since we have many YAML files to be loaded, we simplify the loading process by creating a shell script (named *create-kafka-env*) that loads all YAML files at once. This script is also available in the thesis repository [8].

This script contains the commands needed to create the namespace, to configure the load balancer (*metalLB*), to load the system (all defined YAML files) and to define the external services needed to expose our system.

Upon concluded the loading process, this script lists the created services, giving the external IP addresses that are used to exchange data with the system. The script is described next:

```
kubectl create ns kafka
kubectl apply -f metalLB
kubectl apply -f pv-rwo.yaml -n kafka
kubectl apply -f zookeeper.yaml -n kafka
kubectl apply -f kafka.yaml -n kafka
kubectl apply -f mosquitto-in.yaml -n kafka
kubectl apply -f mosquitto-out.yaml -n kafka
kubectl expose mosquitto-service-in -n kafka
kubectl expose mosquitto-service-out -n kafka
kubectl apply -f rest-api.yaml -n kafka
kubectl expose rest-external-service -n kafka
kubectl apply -f coap.yaml -n kafka
kubectl expose coap-external-service -n kafka
kubectl apply -f controller.yaml -n kafka
kubectl get svc -n kafka | grep "mosquitto-service-in"
kubectl get svc -n kafka | grep "mosquitto-service-out"
kubectl get svc -n kafka | grep "rest-external-service"
kubectl get svc -n kafka | grep "coap-external-service"
```

[8] https://github.com/jsoares11/MasterThesisExperiences

5: Experiments

As discussed before IoT applications are growing in number. We need new IoT solutions to be resilient, able to respond as soon and efficiently as possible in case of failure of any service. Furthermore, we don't want the system to be dependent of every component, since there are many components and one failure must not lead to the whole system failure, hence the use of micro services is crucial.

We want to see if the system will still deliver messages even if a node dies, see if the number of topics in the system have a significant influence on its performance. Also, we want to investigate if the number of Kafka instances in the Kafka cluster or Zookeepers instances in the Zookeeper cluster has a big impact on the system's behaviour. These behaviours must be studied on our system, therefore experiments were performed.

Since we have a big IoT system with many functionalities, as described in chapter 4, we divide the experiments in 4 groups:

- the first group corresponds to testing **communication protocols**
- the second group corresponds to the test of the **system** as a whole
- the third group is the test of the **Kafka system**
- finally the test of **Kubernetes cluster** functionalities

These experiments should help us to evaluate if our IoT system is **interoperable, resilient, scalable, easy to manage and that ensures data availability**. All of the experiments are described in section 5.2. With these experiments we are be able to understand what greatly changes our system's behaviour, what are the key aspects that have the most impact on its performance.

This chapter first describes the setup used to make the experiments, then the testing scenarios are presented where each test is described. After that, the implementation that allowed those experiments to be made is presented. In the next chapter Results of these tests can be found along with conclusions on the system's behaviour.

5.1 Setup Configuration

A kubernetes cluster with five virtual nodes, one acting as the master node, and four acting as the slave nodes was used to run the experiments. Two more virtual machines were used to act as the Producer and Receiver. All virtual machines run the Ubuntu 18.04 operating system and have two CPUs. The master, producer and receiver nodes were all given 3108 MB of RAM while the slaves had only 2048MB of RAM.

Nodes/ Stats	**Master**	**Slave1**	**Slave2**	**Slave3**	**Slave4**	**Producer**	**Receiver**
RAM (MB)	3108	2648	2648	2648	2648	3108	3108
Storage (GB)	40	40	40	40	40	40	40
#CPUs	2	2	2	2	2	2	2
OS	Ubuntu 18.04	Ubuntu 18.04	Ubuntu 18.04	Ubuntu 18.04	Ubuntu 18.04	Ubuntu 18.04	Ubuntu 18.04

Table 5.1: Virtual Machine Node's Configuration

5.2 Testing Scenarios

5.2.1 Protocols

Scenario 1: Delay and Packet Loss

The objective of this experiment is to understand how the protocols behave when passing through the whole system. We want to check the packets lost and delay from the Producer to a Receiver (on the other side of the architecture).

At the end we should be able to compare the delay and packet loss of all protocols and compare them in order to determine its main differences.

To calculate the delay and packet loss a timestamp is captured in the moment that each message is being sent and a sequential number starting with the value 0 is added to the payload of the message. This way the receiver will get the message with sequential number 0, 1, 2, 3 and so on. Afterwards, we can check if any message got lost. The experiment was taken under three different sending rates:

- one message for each second,
- one message for each 100 milliseconds and
- one message for each 10 milliseconds.

Each test ran during two hours (each protocol and rate).

5.2.2 System

Scenario 2: Messages with big size - delay vs msg size

The experiments done under scenario 1 were made with a fixed message size comprised of 39 Bytes. To understand the capacities of the system in terms of handling big sized messages this test comprises sending messages with size of 2000Bytes. By checking the delay it's possible to compare the results with the results from scenario 1 and see if the delay increased, if so it's fair to infer that system performance is changed depending on the message size.

Scenario 3: Messages with variable size - delay vs msg size variation

In order to evaluate if the performance of the system is affected by the variation of messages's size being sent, in this test messages are sent following the same behaviour as in scenario 1, but the payload of messages varies from 100Bytes to 1000Bytes (randomly). If any node crashes or the delay is substantially increased it means the system wasn't well prepared to handle such size variation.

Scenario 4: fail recovery (node suddenly unavailable)

The objective of this experiment is to understand how the system handles nodes that fail due to lack of resources or eventual crashes caused by bugs. With the use of Kubernetes it is expected that after approximately five minutes [103], kubernetes recognizes that the node is unavailable and try to reallocate its pods on other available worker. By manually turning off a node while a Producer sends data to a Receiver each second, we will be able to understand if the pods are transferred to other available node. If data stops being received the moment the node goes down and after a reasonable amount of time (the time needed to get pods reallocated and restarted) data starts being received again, then we are able to confirm that the system is resilient and fault tolerant.

Scenario 5: low vs high throughput

Having one Producer versus having many Producers must have some impact on the system. With this experiment we have one producer sending data every second versus 10 Producers simultaneously, all starting at the same time. At the end of both tests, we compare the levels

of CPU and RAM of all slave nodes to determine if there was indeed an impact on the system behaviour.

5.2.3 kubernetes

Scenario 6: Time spent to add a node on a cluster with few vs many nodes

To study the influence of adding a node to a system we will start a cluster containing only one node and will add another node to it, checking the time it takes from the "Not ready" state to "Ready" state. The *Kubectl get nodes* command is used.

In order to run this experiment, we used a script called *add_node_time.sh* that can be found in the GitHub repository of this thesis [1]. This script is called with a Linux method named *timeout* that allows us to run our script for a specific amount of time (in our case for 5 minutes). In this time the script keeps executing the command *Kubectl get nodes* and saves the state of the nodes with a timestamp on a file, when the state changes from "Not Ready" to "Ready" we calculate the amount of time that kubernetes needs to add the node. This test is repeated on clusters containing 5 and 10 nodes. At the end we compare the time spent when adding the node on the different clusters to understand if the number of nodes in a cluster has an important impact on the time needed by the cluster to add a new node.

Scenario 7: Scaling delay

To understand the time that Kubernetes requires to scale pods, in this experiment a cluster containing one pod running Mosquitto IN will be scaled to ten pods, twenty, thirty, forty, fifty, sixty and seventy pods.

The time needed by Kubernetes to execute the scale command on each of these cases is saved and afterwards they are all compared.

This way we are able to study the behaviour of Kubernetes in terms of scale time. The time spent to execute the scaling is calculated by checking the time taken for the pods status change from *"Creating* to *Running"*. This status is visible by using the command *Kubectl get pods*.

5.2.4 Kafka

Scenario 8: Replication Factor

With this experiment we intend to verify if the system is affected by the number of Kafka pods. Summing up, we want to see the impact of having a big replication factor vs having a low replication factor and compare the results to understand if this factor is crucial on the system's performance. To verify this, the same method as in scenario 1 will be used: data will be flowing from the Producer to the Receiver and the delay will be calculated for both cases along with the CPU and RAM usage of the slaves running Kafka pods. If the delay, RAM or CPU usage increases it means that replication factor has a big impact on the system.

Scenario 9: Number of partitions

In order to understand if the number of partitions has an impact on our system we perform the same test done in scenario 7 where data is flowing from the Producer to the Receiver and see if the delay is directly affected by the number of partitions using the measured delay plus CPU and memory consumption.

5.3 Implementation:

In order to evaluate all scenarios defined in section 5.2 it was necessary to implement the Producers and Receivers for all Protocols, the node state monotorization (state *"Ready"* vs state *"NotReady"*),

[1] `https://github.com/jsoares11/MasterThesisExperiences/tree/main/implementation/add_node`

the code that allows a user to perform actions on the topic being used and the packet loss calculation. For the Producers, Receivers and the Monitor we picked C as the programming language to work with, as for the remaining Shell scripts were used. Most experiments were made for a period of two hours, for the cases where this was not true the period of the test is described in the *Results* section (section 6).

In this context the following list of files was generated. All of these files are available in the GitHub repository of this thesis [2].

- "**exp_mqtt_pub**": **Producer** that uses MQTT protocol to send packets;
- "**exp_coap_put**": CoAP **Producer** using CoAP protocol to send packets;
- "**exp_rest_put**": **Producer** that sends packets using the REST protocol;
- "**exp_mqtt_sub**": A **Receiver** that uses MQTT protocol to receive packets;
- "**exp_coap_get**": **Receiver** responsible of getting CoAP packets;
- "**exp_rest_get**": A **Receiver** using the REST protocol to receive packets;
- "**monitor**": Script that monitors the cluster node's status to evict pods on crashed nodes;
- "**controller**": Script used by the "Controller" pod with the only purpose of allowing a user to list, delete or create a topic. This allows users to redefine the partitions and replication factor of the used topic;
- "**packet_loss**": Script used for packet loss calculation given a file with the sequential number of all received messages;
- "**get_cpu_usage.sh**": Script responsible for capturing the CPU being used over a period of time;
- "**get_mem_usage.sh**": Script that captures the RAM being used in a period of time;

The message format used for all experiments was the same. They were sent to the system as raw text, but once inside the system and the Proxy, the current timestamp is added at the beginning of the message and the format is changed to conform with a JSON format. After this, the message is saved in Kafka, so Receivers get data in JSON format. In our experiments, the message payload used in fixed message sizes had the following message:

```
fixed message: "standard test with fixed message size"
```

Once inside the Proxy, data would be translated to include the timestamp and to conform with JSON sintaxe/format:

```
{1598554879520:"standard test with fixed message size"}
```

5.3.1 Producers

The Producers are programs that keep sending data to the system at a specific **rate**, during a certain **amount of time**. These parameters where configured directly in source code files, which implies a new source files compilation for each experimentation. Each message sent has a timestamp that indicates the sending instant from the producer. This timestamp is considered part of the payload inside the rest of the system.

These programs receive as arguments the address of the Broker or Server that will be used to send messages. The *amount of time* that the Producers were sending messages was the same for all experiments, and had a duration of two hours, while the *rate* was relative to each experiment (between 1 to 1000 message(s) per second).

The implementation of each Producer is similar for all protocols, in Algorithm 1 a pseudo-code is presented to illustrate the logic behind the producers. Since they are similar we used the MQTT protocol as example. All implemented Producers are available on this thesis repository with the names *"exp_mqtt_put.c"* for MQTT, *"exp_coap_put.c"* for CoAP and *"exp_rest_put.c"* for REST.

During the experiment, the producer will keep on sending messages. A system call to *"mosquitto_pub"* is used to send messages. Each message is composed by a **topic** and **message payload**. In our experiments the topic used was always the same *"iot"*.

[2] https://github.com/jsoares11/MasterThesisExperiences/tree/main/implementation

Algorithm 1: Mqtt producer pseudo-code

```
1 Producers(addr, log);
  Input  : addr - The address of the respective server/broker
           log - The name of the file to save the timestamp of sending
  Output: log with timestamp
2 do
3 |  system(mosquitto_pub addr message);
4 |  log(timestamp);
5 |  sleep(rate);
6 while elapsed <finish_time;
```

Regarding CoAP and REST protocols, clients based on *libcoap* and *"ulfius"* libraries were implemented to support both these protocols.

As described before, each protocol implementation has an external service and address that is used to communicate with the system. Each of these Producers will establish the connection with the corresponding service on the system, in this case MQTT producer is connected to Mosquitto In, CoAP to CoAP Server and REST to Rest Server.

5.3.2 Receivers

In this work, the receivers did not follow the same approach for all protocols.

The MQTT receiver differs from the others because this protocol follows the *publish-subscribe* pattern instead of *request-response* pattern.

This receiver receives as arguments the address of the Mosquitto and the name of the log file where the received messages and arriving timestamps are registered.

With the use of the functions *"popen"* and *"fgets"* from C library we are able to use *"mosquitto_sub"* and get the messages it receives. The *"popen"* command is used to create another process that runs a program and creates a pipe between them, returning a pointer to a stream that can be used for read or write data.

In this case the Mosquitto client (*"mosquitto_sub"*) is called and the *"fgets"* function is used to read/obtain the received messages.

In Algorithm 2 a pseudo-code of the receiver behaviour is presented. First the *"popen"* function is used to get the stream, and then, while *fgets* is able to read data from that stream, the JSON message is unwrapped to get the wanted fields.

The unwrapping was done with the use of the function *"strtok"*, which breaks a message into a series of tokens using a delimiter (the delimiter used was the character ":"). These fields are the *timestamp* (current timestamp), *sent_timestamp* (received in the payload), *payload received* and *sequential number*.

A log file is created at the beginning. Everytime a message is received the unwrapped fields are appended to the log file.

Algorithm 2: Mqtt receiver pseudo-code

```
1 MQTT Receiver(addr, log);
  Input  : adr - The address of the respective broker
           log - The name of the logfile to use
  Output: log with received timestamp, sent timestamp, payload and sequential number
2 cmd = "mosquitto_sub addr";
3 fp = popen(cmd, "read");
4 while fgets!=NULL do
5 |  strtok(msg_rcved);
6 |  log(timestamp, sent_tmstmp, payload, seq_num);
7 end
```

In the case of CoAP and REST implementations the Receiver is continuously sending get requests to the system until a timeout without receiving data is reached, in our case 20 seconds without receiving data means the experiment is over.

These Receivers receive as arguments the address of the server that will be used and the log file name. A pseudo-code is shown in Algorithm 3.

In this implementation the *"popen"* function was used to call a CoAP client and/or a REST client that sends the get requests to the given address, while the *"fgets"* function was used to read messages that arrive from the server. Similar to MQTT the *"strtok"* function was used to unwrap the received messages in order to get the message fields which are stored in the log file.

Algorithm 3: CoAP and REST receiver pseudo-code

```
1  CoAP and REST Receivers(addr, log);
   Input  : addr - The address of the respective server
            log - The name of the logfile to use
   Output: log with received timestamp, sent timestamp, payload and sequential number
2  cmd = "./coap-client -m get addr";
3  # or "./simple_client GET addr" (for REST);
4  while elapsed <timeout do
5  |  fp = popen(cmd, "read");
6  |  while fgets!=NULL do
7  |  |  strtok(msg_rcved);
8  |  |  log(timestamp, sent_tmstmp, payload, seq_num);
9  |  end
10 end
```

5.3.3 Monitor

The Monitor is responsible for continuously check the status of all nodes in the cluster. If a node has a "Not Ready" status for more then 6 minutes, the monitor evicts pods from that node. If the node never returns, it must be manually deleted from the cluster.

The monitor doesn't have capabilities to delete any node, only starts the eviction of pods whose nodes had crashed.

If the node returns then the eviction will proceed and the node will be cordoned (meaning that it is marked as an unschedulable node), hence to use it again you must uncordon the pod using the command:

```
$ kubectl uncordon <node_name> -n <namespace>
```

Once started, the Monitor runs forever, and checks the nodes's status with the use of the following command:

```
$ kubectl get nodes -o wide | awk '{if(NR>2) print $1 $2}';
```

This command uses the *"awk"* function to get only the "Status" column from the output of the command *"kubectl get nodes"*, leaving out the columns "Roles", "Age" and "Version". It is executed in Monitor using the *"popen"* command, followed by *"fgets"* to have access to the output of the *"popen"* command. This output is used to check if a node status is found as "Not Ready". When it happens, we need to check how long the node is "Not Ready". For that we need to execute the following command:

```
$ kubectl describe node <nodename> | grep "Ready" | awk '{ if(NR <= 1)
print $3 $4 $5 $6 $7}');
```

This command uses the *"awk"* command to get the last heartbeat heard from the node in kubernetes, leaving out unnecessary information. The numbers 3 through 7 represent the columns where the data of the last heartbeat is represented (e.g. Wed, 09 Sep 2020 14:41:51).

The *"fgets"* function is used to get the output of the *"popen"* function and the field heartbeat received is forwarded to *"is_node_dead"* function, which evaluates if a node was "Not Ready" for the last 6 minutes.

This *"is_node_dead"* function uses the function *"strtok"* to separate the day, year, hour, minutes and seconds of the heartbeat and of the current time.

Both timestamps are then evaluated in order to determine its difference. If the difference is bigger then six minutes then we assume that the node crashed and proceed to evict its pods. To evict the pods a new thread is created allowing the monitor to keep checking other nodes while others are being evicted. The thread uses the following command to evict the pods:

```
$ kubectl drain <node_name> --force --ignore-daemonsets  --delete-local-data
```

Algorithm 4 shows a pseudo-code that illustrates the logic of the monitor.

Algorithm 4: Monitor pseudo-code

```
1  Monitor(finish_time, rate);
   Input  : finish_time - Time of the experiment
            rate - The time between each status observation
2  while elapsed <finish_time do
3  |  cmd = "kubectl get nodes -o wide | awk print $1 $2";
4  |  fp = popen(cmd, "read");
5  |  while fgets!=NULL do
6  |  |  strcmp(status, "Not Ready");
7  |  |  cmd2 = "kubectl describe node <node_name> | grep 'Ready' | awk print $3 $4 $5
   |  |   $6 $7";
8  |  |  fp = popen(cmd2, "read");
9  |  |  if fgets!=NULL then
10 |  |  |  evict ← isNodeDead();
11 |  |  |  if evict == true then
12 |  |  |  |  cmd3 = "kubectl drain <node_name> force ignore-daemonsets
   |  |  |  |   delete-local-data"
13 |  end
14 end
```

5.3.4 Controller

The controller is used to create, list or delete a topic in Kafka. The following command is used to enter in the Controller pod:

```
$ kubectl kubectl exec -n kafka --stdin --tty podname -- /bin/bash
```

The Controller has a set of configurations that must be specified in its arguments. The configurations are:

- *"Method"*: The method to either **Create**, **List** or **Delete** topics from Kafka;
- *"Topic"*: The topic to be created or deleted;
- *"Replication Factor"*: Replication Factor on a newly created topic
- *"Number of Partitions"*: Number of partitions on a newly created topic
- *"Zookeeper Server"*: The address of the Zookeeper in the form ¡address¿:¡port¿

Taking into account the configurations given at the startup of the controller, it will execute different operations:

- If the method given as argument is "create", it executes the following command, which creates a new topic in Kafka.

  ```
  $ ./kafka_2.12-2.5.0/bin/kafka-topics.sh --create --zookeeper <zook_addr>
    --replication-factor <rep_factor --partitions <num_partitions>
    --topic <topic>
  ```

- If the method passed as argument was "list", the Controller will execute the respective command, which allows to list topics from Kafka.

```
$  ./kafka_2.12-2.5.0/bin/kafka-topics.sh --list --zookeeper <zook_addr>
```

- If none of the above methods is passed as argument, the Controller checks if given method is "delete". If true, the topic mentioned in arguments is deleted from Kafka. In order to do this the Controller will issue two different commands to the system: first will change the retention period of messages in the topic to 1 second, which means that after 1 second messages are destroyed; secondly deletes the topic. Both commands are listed next:

  ```
  $  ./kafka_2.12-2.5.0/bin/kafka-topics.sh --zookeeper <zook_addr> --alter
  --topic <topic> --config retention.ms=1000

  $  ./kafka_2.12-2.5.0/bin/kafka-topics.sh --zookeeper <zook_addr> --delete
  --topic <topic>
  ```

  The first command is necessary because Kafka will not destroy topics that contain data. Without this command the deletion of the topic takes the time defined for data retention in Kafka. In Algorithm 5 the script is illustrated.

Algorithm 5: Controller pseudocode

```
1 Controller(method,topic,rep_factor,num_part,zook_addr);
  Input  : method - create, list or delete
           topic - the Kafka topic in question
           rep_factor - the number of replicas to use for the topic in question
           num_part - the number of partitions to use
           zoo_addr - the address of the zookeeper to use
2 if method="create" then
3 |  ./kafka-topics.sh –create <zook_addr> <rep_factor> <num_partitions> <topic>
4 if method="list" then
5 |  ./kafka-topics.sh -list <zook_addr>
6 if method="delete" then
7 |  ./kafka-topics.sh <zook_addr> –alter–topic <topic> –config retention.ms=1000
  |   ./kafka-topics.sh <zook_addr> –delete–topic <topic>
```

5.3.5 Packet Loss

To calculate the number of messages that did not arrive into the system or arrived twice, a Packet Loss script was built. This script evaluates the sequential number generated by producers. This sequential number is received in each message by the Receiver that logs the value in the log file. From that log, it was possible to get all the sequential numbers. If any number is missing, it means the correspondent message was lost, and it counts as a lost message.

It should be noted that since CoAP and REST send *get* requests instead of subscribing to the respective servers if the *get* request gets to the server faster than the *put* request (from the Producer) the server will answer the Receiver the last message it has on the system, meaning that the Receiver will get the same message twice. These cases are not counted as messages lost.

To get data from Kafka, servers use the respective external Service, which uses *metalLB* to pick one available Kafka broker, meaning that the Kafka used is not always the same. Since replication of data between Kafka brokers is not instantaneous, this means that for a short period some of the Kafka Brokers might still have previous values when the Receiver asks for data. Hence, sometimes older messages might be received by Receivers, these kind of situation is not counted as messages lost.

The implemented script is named *"check_packet_loss.c"* and it receives as argument the name of the file containing the sequential numbers (this file must contain only the sequential numbers). The script reads the file line by line, using the command *"fgets"*, and compares the current line to the previous one. If the current line has a smaller or equal value than the last line, then no packets are lost (the value is assumed to be a duplicated). If the current message differs from the last message by one value, then the number is sequential, which means there was no packets lost. Otherwise, if the last message has a smaller value than the second last, and their difference

is bigger than one, messages were lost. The number of messages lost is equal to the difference between the current line and the second last message minus one (current - second last - 1). For example, in a file with the sequential numbers {1,2,3,1,6} the script will assume that the second value "1" is duplicated and it is not counted as a message lost, but messages four and five were not received and are counted as packets lost (6 - 3 - 1 = 2 messages lost). The script is better illustrated in algorithm 6.

Algorithm 6: Packet Loss pseudo-code

```
1  check_packet_loss(filename);
   Input  : filename - the name of the file containing all sequential numbers
2  last = 0;
3  sndlast = 0;
4  while fgets != NULL do
5  |  if current <= last then
6  |  |  continue;
7  |  if current - last == 1 then
8  |  |  continue;
9  |  if last <sndlast then
10 |  |  if current - sndlast >1 then
11 |  |  |  lost += current-sndlast-1;
12 |  else if last - sndlast == 1 then
13 |  |  lost += current - last - 1;
14 end
```

5.3.6 CPU Consumption

Another measure used to evaluate our system is CPU consumption. In order to quantify the metric, a CPU consumption script (*"get_cpu_usage.sh"*) was created. It can also be found at the thesis repository. This script executes the *"mpstat"* command in nodes to get the CPU consumption. The *"mpstat"* command outputs in the standard output activities of all available processors in the machine. This output includes information about the following:

- %usr: Indicates the "amount" of time spent, in percentage, by the CPU to run applications (at user level);
- %nice: In Linux, the CPU priority is measured in nice values (between -20 and 20) [97]. The higher the nice value, the lowest the priority. The nice field in *"mqstat"* indicates the percentage of CPU usage when the execution happens at user level with nice priority (positive nice value, meaning low priority);
- %sys: Show the CPU percentage usage at the system level (kernel);
- %iowait: Indicates the percentage of time that the processors were idle at a moment that the system was overloaded with disk I/O requests;
- %irq: The percentage of time spent by processors to service hardware interrupts;
- %soft: Indicates the percentage of time spent by the processors to service software interrupts;
- %steal: Shows the percentage of time spent waiting by virtual CPUs while the hypervisor served other virtual processors;
- %guest: Percentage of time spent by the CPU to run virtual processors;
- %idle: The percentage of time that the CPU was idle (no outstanding Input/Output (I/O) requests).

To use this script, a package called *"sysstat"* must be installed in the Linux environment.

The developed script *"get_cpu_usage.sh"* receives three arguments: the rate to be used (the amount of time between each "get"), the number of times that is needed to "get" the usage and

the name of the file in which the usage values of all fields are going to be saved. To use the script consider the following example:

```
$ ./get_cpu_usage 1 60 test
```

In this example the script reads the CPU usage every second for a total of 60 times (meaning for a total of 60 seconds/1 minute) and place each value in a file named "test".

5.3.7 RAM Consumption

To capture the RAM consumption of slave nodes in the experiments defined in section 5.2, a script named *"get_mem_usage.sh"* was created. This script uses the *"free"* command to access the memory usage. The *"free"* [67] command provides information about the total amount of free and used physical and swap memory in the system. It also displays information about the buffers and caches used by the kernel. The output of this command contains the following fields [80]:

- **total:** The total memory capacity in the machine;
- **used:** Represents the used memory and is calculated as total - free - buffers - cache;
- **free:** The unused memory in the machine;
- **shared:** Express the memory used by tmpfs (a virtual memory file system);
- **buffers:** The memory that is used by kernel buffers;
- **cached:** Represents the memory that is used for cache;
- **buffers/cache:** The sum of the previous *buffers* and *cache* fields (buffers cache);
- **available**: An estimation of the available memory that can be used to start new applications.

To use the *"free"* command, a package must be installed in the Linux environment called *"gawk"*. The developed script *"get_mem_usage.sh"* receives only one argument, the name of the file to save the memory usage. It is important to notice that this script has an infinite loop (*"while true"*), hence its usage should be heedful. The script is designed to work this way because it is intended to be used with the command *"timeout"*. The *"timeout"* command allows the user to run a command in Linux for a set amount of time. The amount of time between each execution of *"free"* is set by using the *"sleep"* command that waits one second inside the loop, this way, with the *"timeout"* command, we tell the script to run until a specific amount of time every second. An example of the use of the script can be seen next:

```
$ timeout 5s ./get_mem_usage.sh test_ram
```

In this example, the *"get_mem_usage.sh"* script runs for five seconds and saves the captured memory usage in the file named *"test_ram"*. The following Algorithm (7) shows the script in a pseudo-code fashion.

Internally, the script uses three commands, the *"grep"*, *"strftime"* and *"awk"* commands. The *"grep"* command is used to filter the output of *"free"* command in order to get only the line containing the word "Mem". The *"strftime"* command is used to get the timestamp, which allows to understand the value of the RAM usage in a specific moment, and the *"awk"* command is used to add the timestamp to the line that is going to be saved in the log file.

Algorithm 7: get_mem_usage.sh pseudo-code

```
1 get_mem_usage.sh(log);
  Input : log - the name of the log file to save the usage
2 while true do
3 |   free -m | grep "Mem" | awk {print strftime()} » log
  |   sleep 1
4 end
```

6: Results

In this chapter we present the results obtained after running the experiments, determining what aspects define our system, what went wrong, why, and how could the system be changed in order to solve some of the aspects that went wrong. The log files from these results may be found at `https://github.com/jsoares11/MasterThesisExperiences/tree/main/result_logs`.

6.1 Scenario 1: Latency and Packet Loss

In this scenario, three protocols were used to evaluate the system. Firstly, the MQTT protocol and data rates of 1, 10 and 100 messages per second were used. Figure 6.1 shows the average latency obtained from this experiment.

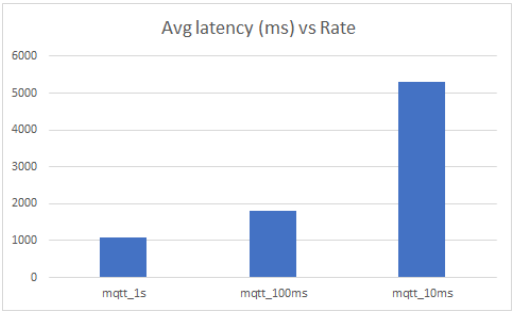


Figure 6.1: MQTT Average Latency (ms) and Packet Loss vs Rate

From Figure 6.1 we can see that an average latency of 1082 ms, 1810 ms and 5297 ms is obtained for 1, 10 and 100 messages per second, respectively. There is a clear tendency to get higher latency when the rate increases.

Concerning packet loss, there were no losses in any of these tests. However, packets were received in a different order than the send order. Since there was replication involved, data was consumed from different Kafka instances which led to a different order when receiving packets. This issue is solved by adding a slight delay to the consumer request.

The second protocol used was the CoAP protocol. In this case, the results differ significantly when compared with MQTT. Figure 6.2 shows the results, concerning latency and losses, obtained from this experiment.

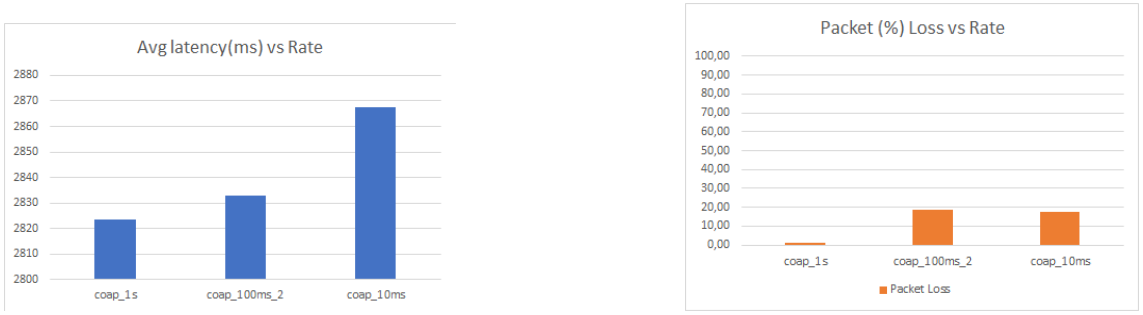


Figure 6.2: CoAP Average Latency (ms) and Packet Loss vs Rate

The results showed an average latency of 2823 ms, 2833 ms and 2868 ms for rates of 1, 10 and 100 messages per second, respectively. Similar to the first case, here there is also a very small tendency to have more latency when the rate gets higher.

Concerning packet loss, for 1 message per second 0,93% of the packets were lost, while for 10 messages per second 18,52% of the total packets sent were lost, and for 100 messages per second 17,58% packets never arrived. In this experiment we noticed that the CoAP server shows low performance to handle a high number of requests per second.

The third protocol used was the REST protocol. In terms of latency the results obtained are a little higher than CoAP. These results are presented in figure 6.3.

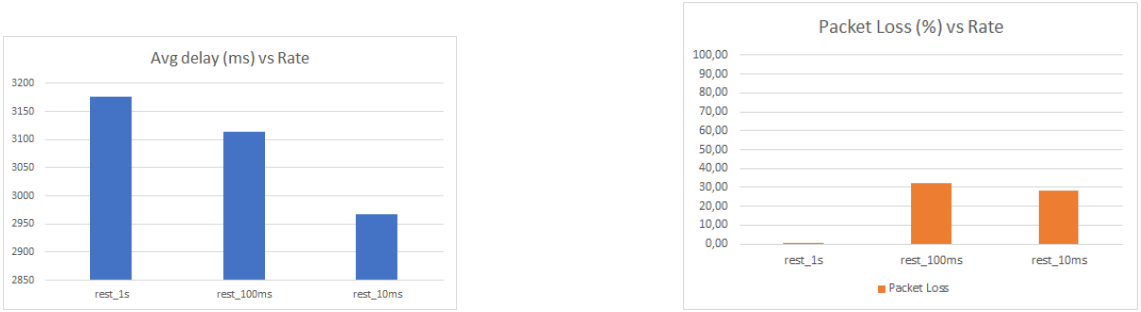


Figure 6.3: REST Average Latency (ms) and Packet Loss vs Rate

The average latency obtained in this experiment for 1 message per second rate was 3176ms, for 10 messages per second was 3115ms and for 100 messages per second was 251ms. In this case, contrary to MQTT and CoAP results, there doesn't seem to be a tendency to have more latency when the rate gets higher.

In terms of packet loss, due to an internal error that occurred inside the experiment script (*"exp_rest_get"*) when sending the get request the receiver in the experiment would stop consuming data. All data from that point on was not received, but not counted as packet loss because the system was still running (no nodes or pods have crashed). Nonetheless, checking the data that was received until the crash it was possible to see that all rates had packets lost. For rate of 1 message per second 0,5% of the packets were lost, for rate 10 messages per second 32,5% of the messages were lost and for rate 100 messages per second 28,3% of the messages were lost.

6.2 Scenario 2: Messages with big size - latency vs msg size

In this scenario, the same tests made in scenario 1 for MQTT were repeated with a bigger message size (2000B). The results obtained from this experiment were compared with the results from the previous experiment (scenario 1) in order to understand the impact of the message size in the system. The figure 6.4 shows a comparison of the results obtained from this experiment and from the experiment done in scenario 1 in terms of latency.

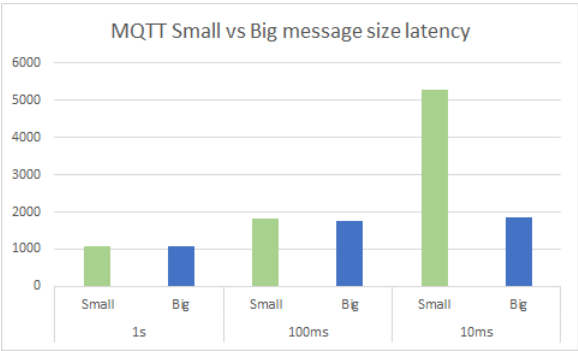


Figure 6.4: MQTT Small vs Big message latency

From the above figure it is possible to see a latency of 1075ms, 1757ms and 1840ms for rates of 1, 10 and 100 messages per second, respectively. There seems to be a tendency to have a bigger delay for higher sending rates.

Excluding the latency obtained for a small sized message with rate of 100 messages per second, results from both experiments seem very similar when comparing small versus big sized messages. There seems to be a tendency to have a similar latency when messages have small or big message size.

Concerning the packets lost, there were no packets lost for all experiments.

6.3 Scenario 3: Messages with variable size - latency vs message size

In this experiment the same approach of scenario 2 was used. To evaluate the impact the message size, the same tests made for MQTT in scenario 1 are going to be repeated, but a variable message size (between 100 B and 1000 B) was used. The results from this experiment were compared with the results obtained in scenario 1. Figure 6.6 shows the comparison made between both experiments results obtained for each rate, concerning the latency and packet loss.

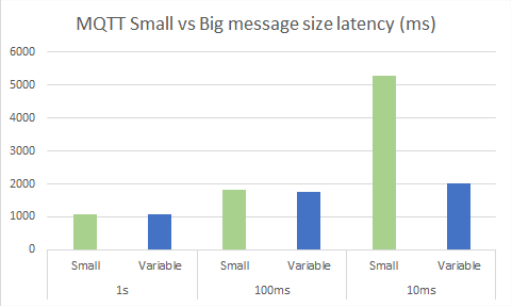


Figure 6.5: MQTT Small vs Big message latency

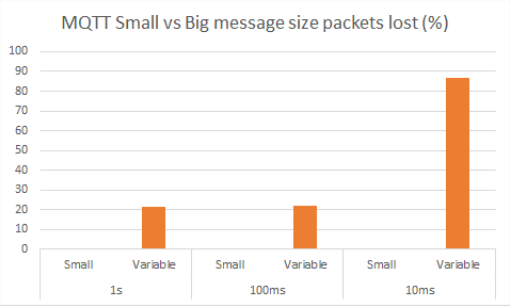


Figure 6.6: MQTT Small vs Big message packets lost

In terms of latency, the values obtained for rates of 1, 10 and 100 messages per second are 1079 ms, 1760 ms and 2019 ms, respectively. These values seem to be similar to the values obtained in scenario 1, except for the results using rate of 100 messages per second.

Concerning packet loss, 21,5 %, 22,27 % and 86,6 % of packets were lost for rate of 1, 10 and 100 of messages per second, respectively. Using messages with variable size, we found out that the system has a big struggle handling the requests. Higher rates originate higher the number of packets lost.

6.4 Scenario 4: fail recovery (node suddenly unavailable)

In this experiment we had a Producer sending data with MQTT to a Receiver for 20 minutes. When the timeline reached half the test (10 minutes), a node was manually *"crashed"* (forced crashed) by turning off one of the slave nodes, in this case the Mosquitto IN service goes down, so that we could see data stop being received. The expected behaviour was that kubernetes would listen to the heartbeats of the node and understand it had crashed. After that, kubernetes should wait about five minutes before evicting the pods in the node and reschedule them to another available node. Unfortunately, we found out that there is a bug concerning *Daemon Sets* in Kubernetes, that leads to a misbehave of the *Node Controller*. The Node Controller is responsible for evicting the pods out of the nodes that crash, but as we understood, it doesn't deal well with Daemon Sets unless the cluster is on the cloud. "Mosquitto In" doesn't use Daemon Sets itself, but since we use bare metal kubernetes cluster, to be able to communicate externally *metalLB* is used to give external ip addresses to services, and metalLB uses Daemon Sets.

This way the actual behaviour was the node status becoming *"NotReady"* but the pods were still running as if the node was alive. To bypass this problem a script called *node_monitor* was built. This script checks all node's status, if it finds a status *"NotReady"* first it checks for how

long it has been in that status. If the node has been *"NotReady"* for more then six minutes, the script enforces an eviction of the pods in that node. The result is an instant rescheduling of the pods and the comeback of the service.

Figure 6.7 shows the packets received on the log file, where it's visible that packets were being received until around the packet number 350. After that, for a while no message arrived, this is due to the fact that the node monitor hasn't yet recognized the node as unfeasible (because kubernetes takes some time to recognize the node as "NotReady" and that time is added to the six minutes monitor waits). The next message received is around the message number 400 and from then on the messages are received again. This shows us that the node has gone down (in our case intentionally), its status became *"NotReady"* and the Monitor took action to evict the nodes and bringing the service back up.

In figure 6.7 the number 0 represents messages that have not arrived and the number 1 represents the messages that have arrived. Meaning that from number 357 until 419 all messages were lost on a total of 1062 messages sent. This totals a number of 62 packets lost from the moment the node crashed until the pods started running on a new node.

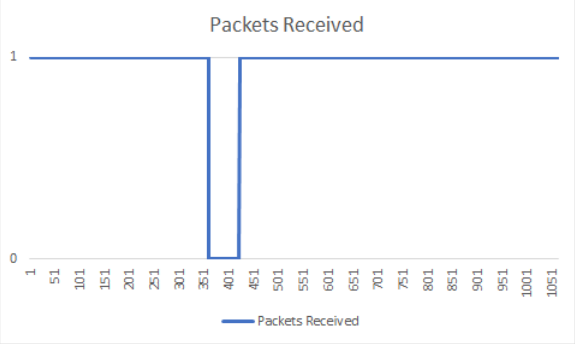


Figure 6.7: #Packets Received

6.5 Scenario 5: low vs high throughput

To evaluate the system behaviour on high vs low throughput we had a Producer sending data to the Receiver using MQTT protocol and measured the CPU and RAM levels of all slave nodes. After this, the same experiment was repeated, but with 2 different virtual machines sending data. Each machine was responsible to run 5 Producers in parallel, having in total 10 Producers sending data to the system.

Figures 6.8, 6.9 and 6.10 show the results of this experiment in terms of CPU and RAM usage levels. These figures show that for all slaves there is no much difference in CPU level from having a single Producer sending data or having ten, hence having low vs high throughput has no impact on the system.

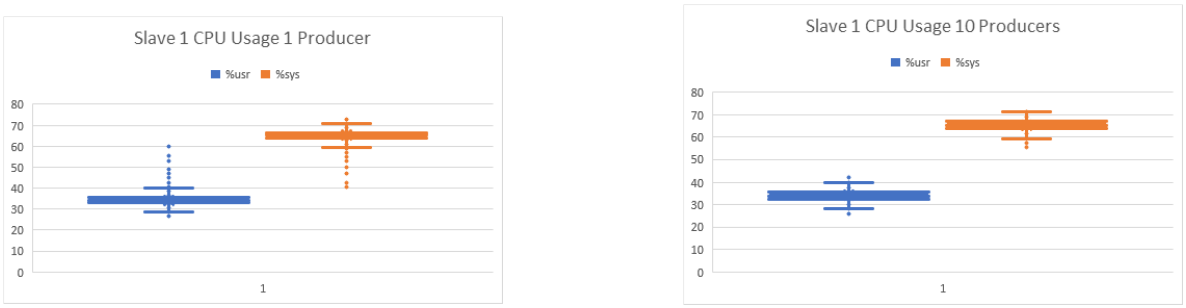


Figure 6.8: Slave 1 - CPU level - 1 Producer vs 10 Producers

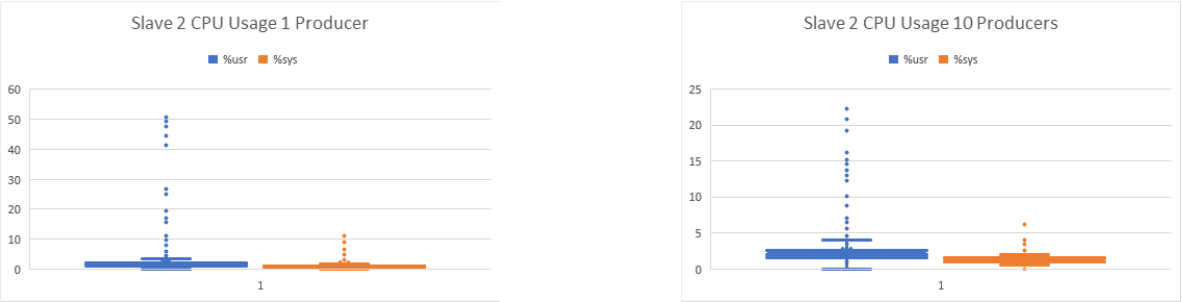


Figure 6.9: Slave 2 - CPU level - 1 Producer vs 10 Producers

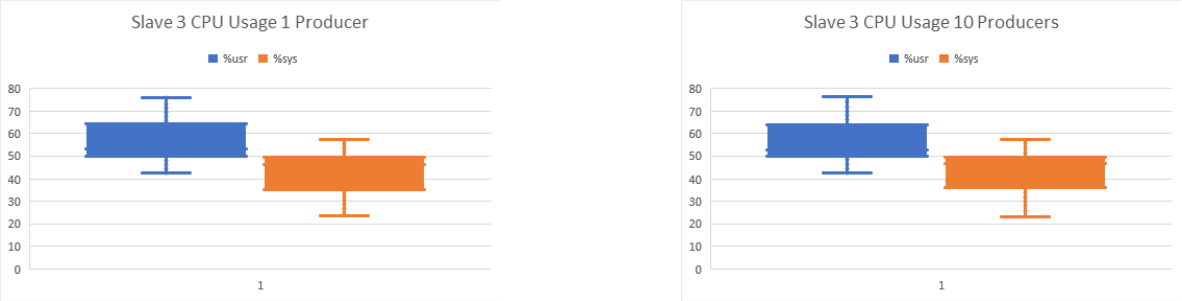
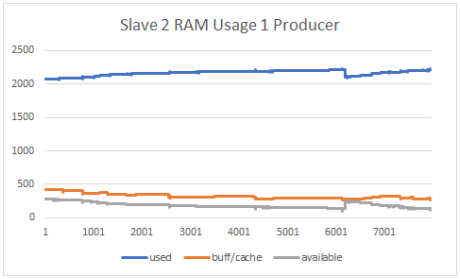


Figure 6.10: Slave 3 - CPU level - 1 Producer vs 10 Producers

In terms of RAM, the same is not true. We found out that the usage of RAM increases in all slave nodes during the experiment. Hence, to determine if the throughput has impact on the system, we decided to compare the impact of one producer with the impact of 10 producers. Although, for tests with only one producer the starting point of RAM usage was around 1900 MB, and for 10 producers was around 1400 MB. After searching the logs for an explanation we found out that the buffer/cache was higher for tests with 10 Producers, which means that in the first test each slave saved information in cache. Looking at the figures 6.11, 6.12 and 6.13 we can see an orange line representing the free space, a blue line representing the memory used and a grey line representing the cache. The figures show that RAM level increases for both tests (low and high throughput), but not which test had the highest increment of memory usage. It can be identified in Tables 6.1 and 6.2, which represent the maximum and minimum values for RAM usage. With these tables we are able to compare the growth of memory usage in each slave node.

Figure 6.11: Slave 1 - CPU level - 1 vs 10 Producers



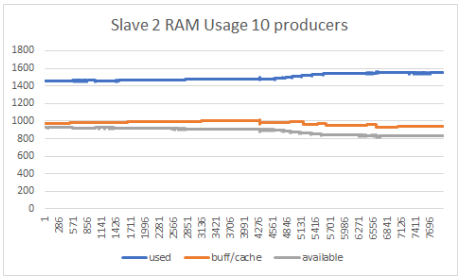
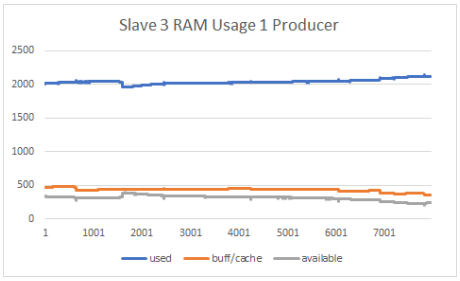


Figure 6.12: Slave 2 - CPU level - 1 vs 10 Producers



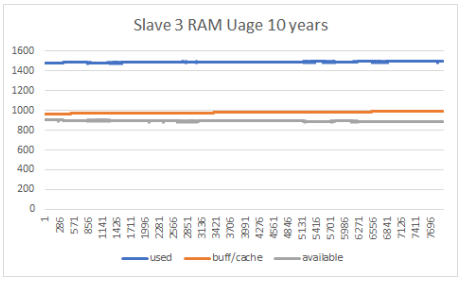


Figure 6.13: Slave 3 - CPU level - 1 vs 10 Producers

Node vs RAM (MB)	**Slave1**	**Slave2**	**Slave3**
Máx Used	2108	2233	2143
Min Used	1841	2071	1964
Increased	267	162	179

Table 6.1: Increment of RAM usage on tests with one Producer

Node vs RAM (MB)	**Slave1**	**Slave2**	**Slave3**
Máx Used	1839	1558	1499
Min Used	1431	1455	1479
Increment	408	103	20

Table 6.2: Increment of RAM usage on tests with ten Producers

From tables 6.1 and 6.2 we can see the growth of the memory usage concerning the throughput. It is possible to see that on both cases (low and high throughput) the slave 1 has a higher increase of RAM used than the other slaves. For low throughput slave 1 has the usage of RAM increased by 267 MB, while slave 2 and 3 have 162 MB and 179 MB respectively. For high throughput slave 1 has the usage of RAM increased by 408 MB, while slave 2 and 3 have 103 MB and 20 MB respectively. The reason for this behaviour is that the pod that handled the requests was running on the slave node 1, hence, needed more memory to work than the other slaves.

Based on this experiment we conclude that throughput does indeed have an impact on the system behaviour, but it is mainly directed to the usage of RAM and only affects the nodes that are directly connected to the work that is being done (in this case, Mosquitto In).

6.6 Scenario 6: Time spent to add a node on a cluster with few vs many nodes

The starting point for this experiment consists on a cluster with one master and one slave nodes. From here on forward we added nodes to the cluster checking the time it took to become *"Ready"*.

A script named *"add_node_time.sh"* is used along with the function *timeout* of the linux library in order to stop the program after a specific time. The script runs a *while true* that calls the command *"kubectl get nodes"* and saves the state of a desired node on a pre-defined file along with the timestamp.

The timestamp was calculated using the command *strftime* of the Linux distribution. Running this script before adding the node to the cluster and stopping it after the node is ready outputs a file containing the first moment that the node was recognized but wasn't available on the cluster, status: "Not Ready", and also the moment when that status changed to *"Ready"*.

With both those timestamps we were able to determine the time needed to add one single node to the kubernetes cluster having already 1, 3, 5, 8 and 10 nodes.

The results show that for any of these clusters the amount of time is very similar, around 30 to 40 seconds. There is no sign of a tendency, maybe the tests were few and with a cluster with more nodes would actually have an impact but with a cluster containing from one to ten nodes the latency is very similar. The results are shown in figure 6.14. From this figure is clear that the time spent doesn't follow the number of nodes that are added (there is no direct relation in terms of time consuming).

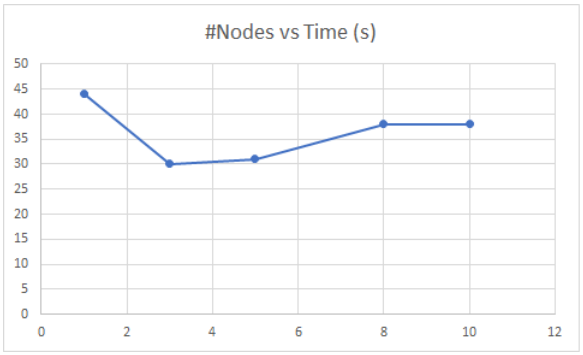


Figure 6.14: #Nodes in the cluster vs time spent to add another node

6.7 Scenario 7: Scaling delay

This experiment started on a cluster with a master and three slave nodes. Firstly, we created a deploy of the system without replication of any component.

In order to evaluate how much time each scaling would take, a program called *"scale_time.sh"* was used along with the Linux command *"timeout"* in order to run the script for a specific amount of time, similar to *"add_node_time.sh"*.

This script runs a *while true* that calls the *"kubectl get replicasets -n kafka"* command which shows information (namely, the *Desired* number of pods, *Current* existing pods and those that are *Ready* under a *Replicaset*) in the namespace *kafka*.

This information is saved in a given file and is used to find the first time that the cluster finds out about the need to scale the chosen pod (*Desired*) and also to find the first time that the number of needed pods are all ready to be used in the cluster (*Ready*).

For our experiments we picked *Mosquitto_in* as the pod subject and started by scaling it with the command:

```
$ kubectl scale deployment mosquitto-deployment-in --replicas=10 -n kafka
```

A scale of 20, 30, 40, 50, 60 and 70 pods were performed. All went fine until the scale for 70 pods. Apparently, our cluster can't handle scaling 70 pods, the result of trying is a crash of one or

more nodes. Until the 70 pod scaling the results obtained show that in fact the higher the number of pods we tried to scale is too much and requires longer processing times from kubernetes.

Also, we learned that if the cluster doesn't have slaves with enough resources to handle such number, then nodes may crash, leading to a need to scale down the pods or increase the cluster resources. These results are illustrated in Figure 6.15.

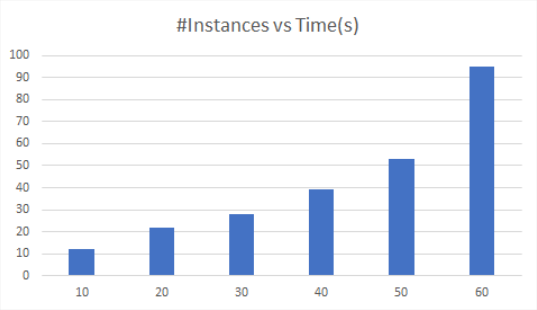


Figure 6.15: #Instances vs time spent scaling them

6.8 Scenario 8: Replication Factor

In this scenario we had an MQTT Producer sending data to the system and an MQTT Receiver subscribing that data. The experiment comprised tests with replication factor 1, 3, 7, 10 and 15.

The initial system is composed by 3 slave nodes that comprises 3 Kafka pods. In this experiment this setup was changed to support a replication factor bigger than 3. To do this we scaled the Kafka pods with the command:

```
$  kubectl scale statefulsets kafka --replicas=<#replicas> -n kafka
```

Tests with replication factor (#replicas) 1, 3 and 7 had an average latency of 1644 ms, 1541 ms and 1545 ms, respectively. In terms of packet loss, there were no packets lost in any of the tests. These results can be seen in figure 6.16.

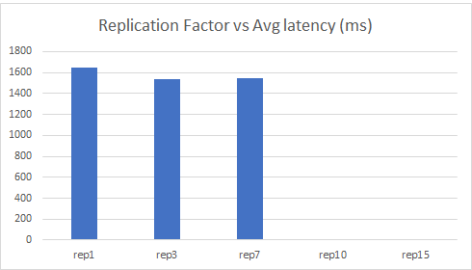


Figure 6.16: Replication factor vs Avg. latency

Despite the difference of 100 ms in the test with replication factor 1, we assume that this is related to the lack of parallelism that is offered by the Kafka replication feature. In this case producers and consumers are using the same replica to write and read data, which may introduce some extra latency.

Hence we concluded that replication factor has some impact on write and read delays. The performance of the system improves when many replicas are used. Since many producers and consumers may interact with the system, at the same time, it is important to distribute the load of writing or reading by different replicas, which has impact on the behaviour of the system.

In this experiment we noticed that replication factors of 10 and 15 are not supported since slave nodes are constantly crashed. In our opinion this is due to the lack of resources, mainly RAM memory that is allocated to each slave node.

6.9 Scenario 9: Number of Partitions

In order to study if the behaviour of the system is influenced by the number of partitions in a topic, tests with MQTT Producer sending data and MQTT Receiver subscribing that data were performed with topics configured with 1, 50 and 100 partitions. The *Controller* component was used to change the number of partitions of the topic for each test.

Figures 6.17 and 6.18 show the average latency and the packet loss that result from this experiment. The test with a single partition had an average latency of 1531 ms without losses. This value is similar (around 1548 ms) for 50 partitions, while it takes 27253 ms when 100 partitions are considered. In terms of packet losses, the system showed no losses for 1 and 50 partitions. Concerning the test with 100 partitions 1,42% of the packets were lost. In this case the system shown high level of CPU and RAM usage, originating the crash of the node after one and half hours.

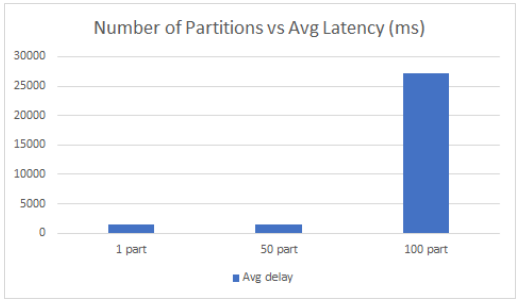


Figure 6.17: Number of Partitions vs Avg. latency

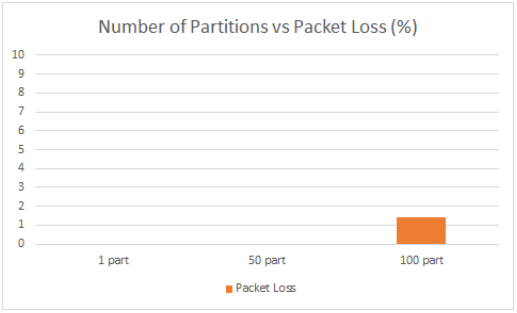


Figure 6.18: Number of Partitions vs Packet Loss

This test allows to conclude that there is a limitation on our system from which the number of partitions within a topic is going to be troublesome. In the current version of the system that number is between 50 and 100 partitions. Other configurations must be tested in order to evaluate the capability to handle more partitions.

7: Conclusion

Summary

With this work we have gained a considerable amount of knowledge relative to IoT systems, namely, how IoT started to appear, what the community is working on, which tools are being studied and built in order to have systems fulfilling the most possible requirements defined in 3.2.

We had the opportunity to study many technologies such as the communication Technologies, the patterns and protocols that are used in the communication between devices and IoT platforms.

Additionally, we learned how systems work in the context of resource constrained environments, where multiple failures may happen due to many different reasons, that vary from simple programming bugs to bad design of the system as a whole.

In IoT systems it is very important to be prepared to accommodate new technologies and devices that raise quickly. So it is important to have flexible and adaptable architectures. In this context, Cloud computing, micro-services, Kubernetes and Kafka are frameworks that help to create good solutions. For example, with the use of the cloud infrastructure it is possible to obtain more resources that allow, for instance, to use kubernetes to deal with large number of devices from a single cluster. It is important to manage devices easily at a single point. Additionally, it is important to deal with large amount of data. For that, Kafka offers a good solution for data persistence, fault tolerance and concurrent readings (high performance).

In this thesis we proposed, developed and evaluated an architecture designed for IoT applications where a large number of devices, producing data and/or consuming it, may be connected. The proposed architecture is designed to run in a cluster fashion, where kubernetes and kafka brokers are used to offer high degree of reliability and performance. Moreover, the architecture is supported by micro-services that offer capabilities to be easily scaled and updated. This structure of micro-services is also important to add or remove data protocols that are used in the whole system. As demonstrated in the experimental section, the architecture is able to deal with different data protocols (e.g., MQTT, CoAP and REST), it is resilient, ensures data availability, is scalable and offer mechanisms to deal with faults (fault-tolerance mechanisms). In order to simplify the use of the architecture, device and data management mechanisms were also included in its design.

Accomplishments

Taking into account the final version of the architecture and the requirements specified for this work, the major requirements were achieved. However, we do not deal with security aspects. This is an important requirement that should be addressed in the future work. In terms of data transmission, Robust connectivity was defined as requirement for this thesis. We do not address any issue related with data protocols, we just use them based on the default configurations.

Problems

The first problem that we found was related with the transport layer used by CoAP protocol. The CoAP protocol offers support to be used with UDP or TCP transport layer. However, in our solution, due to the use of a basic load balancer (MetalLB), the architecture is limited to the use of TCP protocol.

The second problem that we found was related with kubernetes pods eviction. According to the specification of kubernetes is should be done automatically. However, we noticed that this is

not true for our implementation, and we were forced to implement a solution to avoid this problem and redistribute the pods by other nodes automatically. A new process was needed to monitor the nodes and take care the needed actions.

Finally, testing conditions were not ideal. The experimental setup was composed by four slave nodes with 2600 MB RAM, and therefore some experimental conditions were limited to this setup, which may be insufficient to demonstrate all capabilities of the proposed architecture.

Future Work

As future work we would point some directions that we do not have time to explore:

- Evaluate the performance of the system when many data topics are used simultaneously;
- Develop a strategy to deal with UDP protocol, at the transport layer, when CoAP is used in the system;
- Implementation of more micro-services to deal with other data communication protocols, such as AMQP or OPC-UA;
- Explore the possibility to integrate the LoRaWAN server inside the cluster;
- Study and implement mechanisms to automatically add or remove nodes from the cluster, in run-time, to avoid unnecessary resources when they are not needed, keeping the system energy efficient;
- Study and explore debugging tools and mechanisms to improve the system administration, allowing easy debugging and reconfiguration;
- Testing the whole architecture in a cluster environment with more resources and different loads.

Appendix A: Kubernetes Installation

To install a Kubernetes cluster some commands must be applied. These commands are applied **manually** or using **scripts** that were build to automate the installation.

The *manual* installation is presented below as a simple guide to help the user or administrator to replicate the work done in this thesis. In order to make it simple, a set of *scripts* were created. One script creates and configures a master node and the other script allows to configure the slave nodes that will integrate the cluster. These scripts can be found in `https://github.com/jsoares11/MasterThesisExperiences` with the names *"kubernetes_installation_master.sh"* and *"kubernetes_installation_slave.sh"*.

Installation and configuration based on scripts:

To install Kubernetes cluster through the scripts, firstly choose the node type (master or slave), download the corresponding file from the repository and give it permissions to run on the machine. The execution permissions can be changed through the terminal by using the following command:

```
Master node: $ sudo chmod +x kubernetes_installation_master.sh
Slave node: $ sudo chmod +x kubernetes_installation_slave.sh
```

After running the master script, a kubernetes cluster containing only the master node is created and a token is generated as output (besides other information). This token must be saved because it will be used together with join function which allows slaves to join to the cluster.

In order to add nodes to the cluster, the script *"kubernetes_installation_slave.sh"* must be executed in all slave nodes. This script receives as an argument the name of the slave node within the cluster (this will replace the hostname of the machine). The following commands are used to configure the master and slave nodes:

```
$ ./kubernetes_installation_master.sh
$ ./kubernetes_installation_slave .sh slave1
```

When a slave wants to join to the cluster, it must run the join command where the arguments must be the ones retrieved by the execution of the *"kubernetes_installation_master.sh"* script. The join command must be similar to the following one, where ¡address:port¿, ¡token¿ and ¡generated value¿ are filled with the information retrieved by the script.

```
$ kubeadm join <address:port> --token <token> --discovery-token-ca-cert
  -hashsha256:<generated value>
```

The kubernetes also includes a Dashboard. It is a feature that is not enabled by default. However, it can be turned on by following the *manual* installation (step 7), described next.

Installing Kubernetes manually:

Step 1: Docker Installation

The first tool needed to install Kubernetes is the Docker. Before installing Docker, an update must be done to keep the system updated in the last version and to avoid missing dependencies that might be needed later. After this, the Docker may be installed and enabled. This must be done in all nodes that will be part of the cluster. The following commands are used to do the Docker installation:

```
$ sudo apt-get update
$ sudo apt install docker.io
$ sudo systemctl enable docker
```

In the process of software installation a *"Y/N"* option might be shown, make sure to hit *"Y"* followed by *"Enter"* to proceed with the installation. The command *"docker –version"* can be used to verify if the docker installation ran without any error. If the installation finished successfully, the docker version should be printed.

Step 2: Download Signing Key

After Docker installation, a signing key must be obtained, again for all the nodes. For this, the *curl* command must be used, which is usually already installed in the Ubuntu 18.04 distribution. Otherwise, a previous installation of the *curl* command is necessary and only then the download of the key is possible:

```
$ sudo apt install curl
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo
  apt-key add
```

Step 3: Download Kubernetes Repository

After getting the Kubernetes signing key on all the nodes, we need to add the Kubernetes repository in our nodes and install the *Kubeadm* tool. This tool allows users to easily create a simple Kubernetes cluster and/or install other necessary tools. After installing *kubeadm* check the installation version to make sure that the installation was successful (optional). The following commands can be used to do that:

```
$ sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial
  main"
$ sudo apt install kubeadm
$ sudo kubeadm version
```

Step 4: Disable Swapping

The swap memory should also be disabled on all the nodes because kubernetes doesn't behave properly in systems with swap memory enabled. To enforce swap memory to be permanently disabled, the *swapoff* command must be used and a specific file must be edited ("/etc/fstab", in Ubuntu 18.04 distribution). In this file, the "swap" field must be deleted or commented (last line of the file). Otherwise swap memory will be enabled again after node restart.

```
$ sudo swapoff -a
$ sudo nano /etc/fstab
```

Step 5: Changing hostnames

Suggestive hostnames should be given to all the nodes composing the cluster. On the master node consider using, for example, *"master-node"* and for the slaves *"slave-node-1*, *"slave-node-2* and so on for all the slave nodes in the system. The following commands are used to perform these actions:

```
$ sudo hostnamectl set-hostname master-node
$ sudo hostnamectl set-hostname slave-node-1
```

Step 6: Run Kubernetes and Join slave nodes

The Kubernetes cluster may now be created in the master node. To do this, run the following command on the master node:

```
$ sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

It is important to notice that kubernetes will trow some warnings related to the number of CPUS, when trying to install Kubernetes on machines with less than two CPUs. It is possible to proceed the installation by ignoring the warnings, but this is not recommended since the system will crash by lack of resources when synchronization and node analysis are performed. To ignore the warnings use the option *-ignore-preflight-errors=NumCPU* as shown below. To avoid this problem, it is recommended that the devices used in the Kubernetes cluster have, at least, two CPUs.

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16 kubectl
--ignore-preflight-errors=NumCPU
```

This command will output four other commands. The first three commands are used to create a directory named *".kube"* where a configuration file, named *"config"*, is created. The last command in the output should be saved because it is going to be used to join slave nodes into the cluster.

```
$ mkdir -p ~/.kube
$ sudo cp -i /etc/kubernetes/admin.conf ~/.kube/config
$ sudo chown $(id -u):$(id -g) ~/.kube/config
$ kubeadm join <address:port> --token <token> --discovery-token-ca-cert-hash
  sha256:<generated value>
```

Before executing the last command in all slave nodes, a pod network acting as an intermediate between the nodes and the network should be deployed. To do this, use the *apply* command:

```
$ sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/
  master/Documentation/kube-flannel.yml
```

After running the join command on all slave nodes, the created cluster can be verified in the master node through the following command:

```
$ kubectl get nodes
```

Step 7(optional): Enabling Kubernetes Dashboard

It is possible to do everything on Kubernetes through the terminal, but there is also a Dashboard available to be used. The Dashboard allows to perform the same actions on the cluster as the terminal however in a more pleasant way.

The Dashboard isn't enabled by default, to enable it, some steps are needed. First, the deployment of the Dashboard must be done, to do that a *YAML* file available on the Kubernetes GitHub [1] must be applied. After doing this, access to the Dashboard is possible, locally, but to sign-in a *token* is needed.

The *token* is used by kubernetes to protect the access to the Dashboard. To get this *token*, a Service Account must be created by applying another *YAML* file available on our GitHub repository [2].

[1] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/
[2] https://github.com/jsoares11/MasterThesisExperiences/blob/main/k8s-dashboard-admin-user.yaml

After applying it, the *token* will be created. To access the token the command *kubectl* must be used with the option *describe* to describe the *admin-user* and check the existing *secrets*.

After finding the secret, we are now able to use the *describe* option again and check what token it has. After accessing the token, it may be used in the dashboard to login. The code to all of these steps is described below:

Deploy the Dashboard:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/
  v2.0.1/aio/deploy/recommended.yaml
```

Run the Dashboard:

```
$ kubectl proxy
```

Access the Dashboard locally:

`http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/`

Sign-in to the Dashboard:

Create Service Account

```
$ kubectl apply -f https://gist.githubusercontent.com/chukaofili/
  9e94d966e73566eba5abdca7ccb067e6/raw/
  0f17cd37d2932fb4c3a2e7f4434d08bc64432090/k8s-dashboard-admin-user.yaml

or

$ kubectl apply -f https://github.com/jsoares11/MasterThesisExperiences/
  blob/main/k8s-dashboard-admin-user.yaml}
```

Find the token:

```
$ kubectl get sa admin-user -n kube-system
$ kubectl describe sa admin-user -n kube-system
$ kubectl describe secret admin-user-token-<generatedValue> -n kube-system
```

Bibliography

[1] Paul Adamczyk et al. “REST and Web Services: In Theory and in Practice”. In: *REST: From Research to Practice* (June 2011). DOI: 10.1007/978-1-4419-8303-9_2.

[2] Chris Adams. *Scalability Versus Elasticity: What's the Difference, and Why Does It Matter?* URL: https://www.parkplacetechnologies.com/knowledge-center/blog/data-center/scalability-versus-elasticity-whats-difference-matter/. (accessed: 25.05.2020).

[3] SS Ahamed. “THE ROLE OF ZIGBEE TECHNOLOGY IN FUTURE DATA COMMUNICATION SYSTEM.” In: *Journal of Theoretical & Applied Information Technology* 5.2 (2009).

[4] Saad Albishi et al. “Challenges and Solutions for Applications and Technologies in the Internet of Things”. In: *Procedia Computer Science* 124 (2017), pp. 608–614.

[5] LoRa Alliance. “White paper: A technical overview of LoRa and LoRaWAN”. In: *The LoRa Alliance: San Ramon, CA, USA* (2015), pp. 7–11.

[6] Dave Anderson. *METALLB*. URL: https://metallb.universe.tf/. (accessed: 20.09.2020).

[7] Mehdi Anteur et al. “Ultra narrow band technique for low power wide area communications”. In: *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2015, pp. 1–6.

[8] Joe Armstrong. “erlang”. In: *Communications of the ACM* 53.9 (2010), pp. 68–75.

[9] Matthew Arrot et al. “AMQP, Advanced Message Queuing Protocol, Protocol Specification”. In: JPMorgan Chase & Co., Cisco Systems, Inc, 2006.

[10] GSM Association. *Introduction to the GSMA IoT Security Guidelines and Assessment*. URL: https://www.gsma.com/iot/introduction-gsma-iot-security-guidelines-assessment/. (accessed: 24.09.2020).

[11] GSM Association. *Narrowband – Internet of Things (NB-IoT)*. URL: https://www.gsma.com/iot/narrow-band-internet-of-things-nb-iot/. (accessed: 03.01.2020).

[12] Jonas Auer. “Distributed Data Store for Internet of Things Environments”. Bachelor's Thesis.

[13] Aloÿs Augustin et al. “A study of LoRa: Long range & low power networks for the internet of things”. In: *Sensors* 16.9 (2016), p. 1466.

[14] The Kubernetes Authors. *Concepts*. URL: https://kubernetes.io/docs/concepts/. (accessed: 14.05.2020).

[15] The Kubernetes Authors. *Containers overview*. URL: https://kubernetes.io/docs/concepts/containers/overview/. (accessed: 27.05.2020).

[16] The Kubernetes Authors. *Daemonset*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. (accessed: 27.05.2020).

[17] The Kubernetes Authors. *Deployments*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. (accessed: 27.05.2020).

[18] The Kubernetes Authors. *Ingress*. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/. (accessed: 29.05.2020).

[19] The Kubernetes Authors. *Jobs*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/. (accessed: 29.05.2020).

[20] The Kubernetes Authors. *Kube-apiserver*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/. (accessed: 27.05.2020).

[21] The Kubernetes Authors. *Kube-controller-manager*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/. (accessed: 27.05.2020).

[22] The Kubernetes Authors. *Kube-proxy*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. (accessed: 27.05.2020).

[23] The Kubernetes Authors. *Kube-Scheduler*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. (accessed: 26.05.2020).

[24] The Kubernetes Authors. *Kubelet*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/. (accessed: 26.05.2020).

[25] The Kubernetes Authors. *Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components/. (accessed: 26.05.2020).

[26] The Kubernetes Authors. *Kubernetes Pod Overview*. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/. (accessed: 26.05.2020).

[27] The Kubernetes Authors. *Learn Kubernetes Basics*. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/. (accessed: 29.04.2020).

[28] The Kubernetes Authors. *Namespaces*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. (accessed: 27.05.2020).

[29] The Kubernetes Authors. *Persistent Volumes*. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/. (accessed: 24.09.2020).

[30] The Kubernetes Authors. *Replicaset*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. (accessed: 27.05.2020).

[31] The Kubernetes Authors. *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/. (accessed: 30.04.2020).

[32] The Kubernetes Authors. *Statefulset*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. (accessed: 27.05.2020).

[33] The Kubernetes Authors. *Volumes*. URL: https://kubernetes.io/docs/concepts/storage/volumes/. (accessed: 27.05.2020).

[34] The Kubernetes Authors. *What is Kubernetes?* URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (accessed: 25.05.2020).

[35] The open62541 authors. *open62541 Documentation*. (accessed: 4.12.2019).

[36] Gaurav Awasthi. *The Five Essential IoT Requirements and How to Achieve Them*. Cognizant Connected Products, 2019.

[37] Microsoft Azure. *What are public, private, and hybrid clouds?* URL: https://azure.microsoft.com/en-us/overview/what-are-private-public-hybrid-clouds/. (accessed: 23.05.2020).

[38] A Banks et al. *MQTT Version 5.0: OASIS Standard*. 2019.

[39] Rui Miguel Ribeiro Barbosa. "Sistema de monitorização de consumo de água utilizando tecnologia Sigfox". In: (2017).

[40] Olaf Bergmann. *C-Implementation of CoAP*. URL: https://libcoap.net/. (accessed: 11.12.2019).

[41] Pravin Bhagwat. "Bluetooth: technology for short-range wireless apps". In: *IEEE Internet Computing* 5.3 (2001), pp. 96–103.

[42] Bjorn A Bjerke. "LTE-advanced and the evolution of LTE deployments". In: *IEEE Wireless Communications* 18.5 (2011), pp. 4–5.

[43] Carsten Bormann. *Implementations*. URL: https://coap.technology/impls.html. (accessed: 10.12.2019).

[44] Jonathan de Carvalho Silva et al. "LoRaWAN—A low power WAN protocol for Internet of Things: A review and opportunities". In: *2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*. IEEE. 2017, pp. 1–6.

[45] José Cecílio et al. “Experimental Evaluation of LoRa Network Reliability over Water Surfaces”. In: (2019).

[46] Confluent. *Internet of Things (IoT) and Event Streaming at Scale with Apache Kafka and MQTT*. URL: https://www.confluent.io/blog/iot-with-kafka-connect-mqtt-and-rest-proxy/?fbclid=IwAR35FDxWmp6q6y7zd1716VPSsn0LocCc_MDhFUqIPTDvoElsD5pNtV6hBYQ. (accessed: 16.05.2020).

[47] P. Costa et al. “The RUNES middleware: a reconfigurable component-based approach to networked embedded systems”. In: *2005 IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications*. Vol. 2. Sept. 2005, 806–810 Vol. 2. DOI: 10.1109/PIMRC.2005.1651554.

[48] Michelle Cotton et al. “Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry.” In: *RFC* 6335 (2011), pp. 1–33.

[49] McGrew D. and D. Bailey. “AES-CCM Cipher Suites for Transport Layer Security (TLS)”. In: *RFC 5246* (2008).

[50] Dialogic. *E-UTRAN*. URL: https://www.dialogic.com/glossary/e-utran-. (accessed: 30.09.2020).

[51] Jasenka Dizdarevic et al. “A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration”. In: (Jan. 2019).

[52] Adam Fendelman. *What Is GSM in Cellular Networking?* URL: https://www.lifewire.com/definition-of-gsm-578670. (accessed: 14.12.2019).

[53] Stefan Ferber. *Ten challenges the international IoT community needs to master (1/2)*. URL: https://blog.bosch-si.com/internetofthings/ten-challenges-the-international-iot-community-needs-to-master-12/?fbclid=IwAR0XkCQ2MiJ9RiDUmQoIup2jaYJNP6o_vDNYmFw4mtt5qzKmX46h6nRJyCU. (accessed: 16.05.2020).

[54] Joel L Fernandes et al. “Performance evaluation of RESTful web services and AMQP protocol”. In: *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE. 2013, pp. 810–815.

[55] Erina Ferro and Francesco Potorti. “Bluetooth and Wi-Fi wireless protocols: a survey and a comparison”. In: *IEEE Wireless Communications* 12.1 (2005), pp. 12–26.

[56] Apache Software Foundation. *Complex Event Processing*. URL: https://databricks.com/glossary/complex-event-processing. (accessed: 30.09.2020).

[57] Eclipse Foundation. *Eclipse Mosquitto™ An open source MQTT broker*. URL: https://mosquitto.org/. (accessed: 7.12.2019).

[58] OPC Foundation. *OPC UA Online Reference*. URL: https://reference.opcfoundation.org/v104/Core/docs/Part6/7.1.2/. (accessed: 4.12.2019).

[59] OPC Foundation. *Unified Architecture*. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/. (accessed: 29.11.2019).

[60] OPC Foundation. *What is OPC?* URL: https://opcfoundation.org/about/what-is-opc/. (accessed: 29.11.2019).

[61] The Apache Softare Foundation. *ZooKeeper Recipes and Solutions*. URL: https://zookeeper.apache.org/doc/r3.6.1/recipes.html. (accessed: 29.05.2020).

[62] 3GPP MCC Frédéric Firmin. *The Evolved Packet Core*. URL: https://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core. (accessed: 30.09.2020).

[63] Ala Al-Fuqaha et al. “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications”. In: (2015).

[64] G.Tanganelli, C. Vallati, and E.Mingozzi. “CoAPthon: Easy Development of CoAP-based IoT Applications with Python”. In: (2015).

[65] David Gay et al. “The nesC Language: A Holistic Approach to Networked Embedded Systems”. In: *SIGPLAN Not.* 49.4 (July 2014), pp. 41–51. ISSN: 0362-1340. DOI: 10.1145/2641638.2641652. URL: http://doi.acm.org/10.1145/2641638.2641652.

[66] GeeksforGeeks. *Challenges in World Of IoT*. URL: https://www.geeksforgeeks.org/challenges-in-world-of-iot/?fbclid=IwAR3YR88X-xYHCc6sxDkHwPR6sPZJ0v0Yp4l7_RsoP2pC1u_4pI2cq8bYbiM. (accessed: 16.05.2020).

[67] GeeksforGeeks. *free Command in Linux with examples*. URL: https://www.geeksforgeeks.org/free-command-linux-examples/. (accessed: 1.10.2020).

[68] Amitava Ghosh et al. "LTE-advanced: next-generation wireless broadband technology". In: *IEEE wireless communications* 17.3 (2010), pp. 10–22.

[69] Shariq Haseeb et al. "Connectivity, interoperability and manageability challenges in internet of things". In: *AIP Conference Proceedings*. Vol. 1883. 1. AIP Publishing LLC. 2017, p. 020004.

[70] Danny Hughes et al. "LooCI: a loosely-coupled component infrastructure for networked embedded systems". In: *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*. ACM. 2009, pp. 195–203.

[71] Augustine Ikpehai et al. "Low-power wide area network technologies for Internet-of-things: A comparative review". In: *IEEE Internet of Things Journal* 6.2 (2018), pp. 2225–2240.

[72] Digi International Inc. *About the Industrial IoT: Definition, Use Cases and Application Examples*. URL: https://www.digi.com/blog/post/about-the-industrial-iot-definition-use-cases-and. (accessed: 18.05.2020).

[73] Docker Inc. *Docker Hub Quickstart*. URL: https://docs.docker.com/docker-hub/. (accessed: 26.09.2020).

[74] Docker Inc. *What is a Container?* URL: https://www.docker.com/resources/what-container. (accessed: 26.05.2020).

[75] SM Riazul Islam, M Nazim Uddin, and Kyung Sup Kwak. "The IoT: Exciting possibilities for bettering lives: Special application scenarios". In: *IEEE Consumer Electronics Magazine* 5.2 (2016), pp. 49–57.

[76] Nick Ismail. *IoT in the enterprise: Requirements, challenges and predictions*. URL: https://www.information-age.com/iot-enterprise-challenges-predictions-123469561/. (accessed: 20.05.2020).

[77] Asad Javed et al. "CEFIoT: a fault-tolerant IoT architecture for edge and cloud". In: *2018 IEEE 4th world forum on internet of things (WF-IoT)*. IEEE. 2018, pp. 813–818.

[78] Container Journal. *5 Container Alternatives to Docker*. URL: https://containerjournal.com/topics/container-ecosystems/5-container-alternatives-to-docker/. (accessed: 19.09.2020).

[79] Luc Juggery. *A Closer Look at Etcd: The Brain of a Kubernetes Cluster*. URL: https://medium.com/better-programming/a-closer-look-at-etcd-the-brain-of-a-kubernetes-cluster-788c8ea759a5. (accessed: 20.09.2020).

[80] Michael Kerrisk. *free(1) — Linux manual page*. URL: https://man7.org/linux/man-pages/man1/free.1.html. (accessed: 2.10.2020).

[81] Jaehyu Kim and JooSeok Song. "A dual key-based activation scheme for secure LoRaWAN". In: *Wireless Communications and Mobile Computing* 2017 (2017).

[82] Joonsuk Kim and Inkyu Lee. "802.11 WLAN: history and new enabling MIMO techniques for next generation standards". In: *IEEE Communications Magazine* 53.3 (2015), pp. 134–140.

[83] Patrick Kinney et al. "Zigbee technology: Wireless control that simply works". In: *Communications design conference*. Vol. 2. 2003, pp. 1–7.

[84] Evgenii Kosenko. "Industrial IoT development for condition monitoring purposes". In: (2018).

[85] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Complex event processing for integration of Internet of Things devices". Bachelor's Thesis.

[86] Mads Lauridsen et al. "From LTE to 5G for Connected Mobility." In: *IEEE Communications Magazine* 55.3 (2017), pp. 156–162.

[87] Stefan-Helmut Leitner and Wolfgang Mahnke. “OPC UA–service-oriented architecture for industrial applications”. In: *ABB Corporate Research Center* 48 (2006), pp. 61–66.

[88] P. Levis et al. “TinyOS: An Operating System for Sensor Networks”. In: *Ambient Intelligence*. Ed. by Werner Weber, Jan M. Rabaey, and Emile Aarts. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148. ISBN: 978-3-540-27139-0. DOI: 10.1007/3-540-27139-2_7. URL: https://doi.org/10.1007/3-540-27139-2_7.

[89] Roger Light. *Eclipse Mosquitto - An open source MQTT broker*. URL: https://github.com/eclipse/mosquitto. (accessed: 11.12.2019).

[90] Roger Light. *pika 1.1.0*. URL: https://pypi.org/project/pika/. (accessed: 9.12.2019).

[91] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC unified architecture*. Springer Science & Business Media, 2009.

[92] Srijan Manandhar. “MQTT based communication in IoT”. MA thesis. Tampere University of Technology, 2017.

[93] Borja Martinez et al. “Exploring the performance boundaries of NB-IoT”. In: *IEEE Internet of Things Journal* 6.3 (2019), pp. 5702–5712.

[94] Kais Mekki et al. “A comparative study of LPWAN technologies for large-scale IoT deployment”. In: *ICT express* 5.1 (2019), pp. 1–7.

[95] Microsoft. *[MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol*. URL: https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-dcom/4a893f3d-bd29-48cd-9f43-d9777a4415b0. (accessed: 2.12.2019).

[96] Microsoft. *Component Object Model (COM)*. URL: https://docs.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal. (accessed: 2.12.2019).

[97] Chetan Mishra. *A brief guide to priority and nice values in the linux ecosystem*. URL: https://medium.com/@chetaniam/a-brief-guide-to-priority-and-nice-values-in-the-linux-ecosystem-fb39e49815e0. (accessed: 1.10.2020).

[98] The Things Network. *Session Keys*. URL: https://www.thethingsnetwork.org/docs/lorawan/security.html. (accessed: 11.12.2019).

[99] Yoav Nir and Adam Langley. “ChaCha20 and Poly1305 for IETF Protocols”. In: *RFC 7539 (Informational), Internet Engineering Task Force* (2015).

[100] Bob O’Hara and Al Petrick. *IEEE 802.11 handbook: a designer’s companion*. IEEE Standards Association, 2005.

[101] OASIS. *Summary of new features in MQTT v5.0 (non-normative)*. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdf#page=136&zoom=100,0,96. (accessed: 6.12.2019).

[102] Eldad Perahia and Michelle X Gong. “Gigabit wireless LANs: an overview of IEEE 802.11 ac and 802.11 ad”. In: *ACM SIGMOBILE Mobile Computing and Communications Review* 15.3 (2011), pp. 23–33.

[103] Andrew Pruski. *Adjusting pod eviction time in Kubernetes*. URL: https://dbafromthecold.com/2020/04/08/adjusting-pod-eviction-time-in-kubernetes/. (accessed: 30.09.2020).

[104] Gowri S. R., Kwame-Lante W., and Bhaskar K. “A Distributed Publish-Subscribe Broker with Blockchain-based Immutability”. In: (2018).

[105] R. A. Rahman and B. Shah. “Security analysis of IoT protocols: A focus in CoAP”. In: *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*. Mar. 2016, pp. 1–7. DOI: 10.1109/ICBDSC.2016.7460363.

[106] Usman Raza, Parag Kulkarni, and Mahesh Sooriyabandara. “Low power wide area networks: An overview”. In: *IEEE Communications Surveys & Tutorials* 19.2 (2017), pp. 855–873.

[107] Siegmund Redl, Matthias Weber, and Malcolm Oliphant. *GSM and personal communications handbook*. Artech House, Inc., 1998.

[108] Brecht Reynders and Sofie Pollin. “Chirp spread spectrum as a modulation technique for long range communication”. In: *2016 Symposium on Communications and Vehicular Technologies (SCVT)*. IEEE. 2016, pp. 1–5.

[109] J.J. Schutte. “Design of a development platform to monitor and manage Low Power, Wide Area WSNs”. Master’s Thesis. University of Twente, 2019.

[110] Scratch. *Iot A Uni*. URL: https://scratch-itea3.eu/sota/iot_a_uni/list?orderby=req_type&ordertype=ASC. (accessed: 21.05.2020).

[111] TechTarget SearchSecurity. *timing attack*. URL: https://searchsecurity.techtarget.com/definition/timing-attack. (accessed: 07.12.2019).

[112] Keith Shaw. *802.11: Wi-Fi standards and speeds explained*. URL: https://www.networkworld.com/article/3238664/80211-wi-fi-standards-and-speeds-explained.html. (accessed: 27.12.2019).

[113] Zach Shelby et al. “RFC 7252: The constrained application protocol (CoAP)”. In: *Internet Engineering Task Force* (2014).

[114] Z Shelby et al. “Constrained application protocol (coap), draft-ietf-core-coap-13”. In: *Orlando: The Internet Engineering Task Force–IETF* (2012).

[115] Pivotal Software. *Understanding RabbitMQ*. URL: https://www.rabbitmq.com/. (accessed: 14.12.2019).

[116] Kevin Sookocheff. *Kafka in a Nutshell*. URL: https://sookocheff.com/post/kafka/kafka-in-a-nutshell/. (accessed: 4.10.2020).

[117] OASIS Standard. “Oasis advanced message queuing protocol (amqp) version 1.0”. In: *International Journal of Aerospace Engineering Hindawi www. hindawi. com* 2018 (2012).

[118] Dierks T. and E. Rescorla. “The Transport Layer Security (TLS) Protocol Version 1.2”. In: *RFC 6655* (2012).

[119] The HiveMQ Team. *Quality of Service 0,1 & 2 - MQTT Essentials: Part 6*. Online. 2019. URL: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/.

[120] Phillip Tracy. *The top 5 industrial IoT use cases*. URL: https://www.ibm.com/blogs/internet-of-things/top-5-industrial-iot-use-cases/?fbclid=IwAR1n90JBVo64R56UbWZzmN0UWUt9h6mg1SAg6UkAkgCQBEv7ljUgqXuVWnM. (accessed: 17.05.2020).

[121] Team Tweaks. *Consumer IoT vs. Industrial IoT – What are the Differences?* URL: https://www.teamtweaks.com/blog/consumer-iot-vs-industrial-iot-what-are-the-differences. (accessed: 17.05.2020).

[122] Nadeem Unuth. *WiFi Explained: The Most Common Wireless LAN Network*. URL: https://www.lifewire.com/wifi-explained-3426413. (accessed: 14.12.2019).

[123] S. Vinoski. “Advanced Message Queuing Protocol”. In: *IEEE Internet Computing* 10.06 (Nov. 2006), pp. 87–89. ISSN: 1941-0131. DOI: 10.1109/MIC.2006.116.

[124] environment & expert systems WAVES - Wireless Acoustics. *Wireless Body Area Networks (WBAN)*. URL: https://www.waves.intec.ugent.be/research/wireless-body-area-networks. (accessed: 20.05.2020).

[125] Chris Woodford. *Bluetooth*. URL: https://www.explainthatstuff.com/howbluetoothworks.html. (accessed: 14.12.2019).

[126] Muneer Bani Yassein, Mohammed Q Shatnawi, et al. “Application layer protocols for the Internet of Things: A survey”. In: *2016 International Conference on Engineering & MIS (ICEMIS)*. IEEE. 2016, pp. 1–4.