A serverless computing framework based on Kubernetes for applications in the IoT domain

Aswin Rajeev

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Future Networked Systems)

Supervisor: Dr. Stefan Weber

August 2021

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Aswin Rajeev

August 31, 2021

Permission to Lend and/or Copy

A serverless computing framework based on Kubernetes for applications in the IoT domain

Aswin Rajeev, Master of Science in Computer Science

University of Dublin, Trinity College, 2021

Supervisor: Dr. Stefan Weber

The enormous growth of the Internet of Things (IoT) has led to an exponential increase in the processing of data at the edge. Serverless computing, sometimes also referred to as Function-as-a-Service (FaaS) is a technology in its nascent stages that removes the necessity of having to have active infrastructure all the time. It uses ephemeral containers for providing stateless and real-time applications without the need to maintain infrastructure. Public cloud providers have serverless platforms, such as Amazon's Lambda, Google's Cloud Function, and Microsoft's Azure Function. While they have advantages in terms of improved development speed and reduced cost, they result in a vendor lock-in with the cloud provider. To get over this, open-source serverless computing frameworks have been introduced but these have not been tested in a constrained environment such as that of an IoT setting.

This research focuses on building a prototype serverless framework based on Kubernetes and how it needs to adapt to payloads for applications in the IoT domain. The prototype uses Docker images containing functions to be executed and spins them up as Kubernetes pods based on IoT event triggers. Further to the implementation of the framework, the major point of research is to look at how the framework scales depending on the traffic from the IoT devices. The framework is loaded using traffic from Apache JMeter to measure pod creation and execution duration for 1, 20, 50 and 100 pods. Multi-node clusters using 3 Kubernetes nodes and 5 Kubernetes nodes is also evaluated and this shows a decrease in latency for creation and execution of functions.

Acknowledgments

I would like to take this opportunity to express my sincere gratitude to my supervisor Dr. Stefan Weber, for his valuable guidance and motivation throughout the course of this dissertation. I would also like to thank my family for their support and encouragement during my course at Trinity.

Aswin Rajeev

University of Dublin, Trinity College
August 2021

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The very first Internet of Things (IoT) [1] dates back to 1982, when the actual Internet itself was not present. It was made by a graduate student at the Carnegie Mellon University, and it came out of a simple problem that he was facing. He knew that his fellow mates at the University consumed a lot of Coke and there was a high probability that if he wanted it, the machine would either be empty or have Coke that was filled recently, which can be very warm and unpleasant. This idea was taken up by two of his friends and a research engineer at the University and they used a board that could sense the light in front of each Coke bottle to sense its availability and temperature. The whole setup was connected to the ARPANET, which was then serving less than 300 computers. The popularity of this grew exponentially and it even led to the creation of a similar setup for M&Ms.

Figure 1.1: The Coke vending machine at Carnegie Mellon University

The Internet of Things is growing at a tremendous pace and is expected to reach over 26 billion connected devices at the start of 2021 [2]. The large-scale deployment of IoT devices has led to the creation of a lot of traffic and unlike traditional web traffic these have different characteristics. IoT environments have functions that are associated with certain events or triggers. Examples include smart water-level monitoring systems where the level of water can result in a notification being triggered to the authorities or a simple smart light in the kitchen that turns on when the owner issues a command or when the ambient temperature is too low. Whatever the case may be, there is a clear trigger or event that results in the actuation of the IoT device.

Another closely related technology, which is also in its nascent stages is serverless computing [3]. Serverless computing, or more specifically, Function-as-a-Service (FaaS) is a cloud computing paradigm that allows the execution of functions on the cloud without provisioning infrastructure in advance. These functions execute on the fly based on the triggers or events that have been defined. Cloud-based offerings are expected to be the major chunk of the enterprise spending with nearly 67% of spending going to the same. The increased adoption has been due to the flexible pay-as-you-go model that has allowed enterprises and organizations to pay for the resources and not for the hardware.

The Internet of Things and serverless computing are closely tied with each other and as mentioned above a lot of the functions being done in the IoT space is often triggered by a well defined rule or an event. IoT requires an open ecosystem that will allow developers and organizations to take it to its full potential. A completely open-source serverless framework that is dedicated to IoT environments can be a possible solution to this. The probing question is, whether it is possible to create one and scale it up through the use of Kubernetes and Docker containers for the functions.

1.1 Motivation and Aim

The two main technologies that form the foundation of this project are serverless computing and the Internet of Things (IoT). The main motivation behind this project was the problem of vendor lock-in and the lack of configuration available in serverless computing frameworks from the public cloud service providers and the fact that open source serverless computing frameworks are not developed keeping IoT environments in mind. Therefore, the necessity for an easy-to-use serverless computing framework in IoT environments through the use of Kubernetes and Docker containers for the functions fuelled my motivation for this dissertation.

This dissertation will showcase how an IoT device can execute serverless functions through the use of Kubernetes and then scale it for a number of executions. The research aims at:

1. Designing and implementing a serverless framework for applications in the IoT domain.

2. Measuring key metrics of the framework like time taken to create new pods for executing the serverless functions, deleting the pods after execution, latency under load, etc.

3. Comparing some of the existing open source serverless frameworks with the implemented prototype to see shortfalls of existing ones and to find room for improvement for the prototype.

1.2 Dissertation Map

This dissertation is structures into 6 sections, each having its own focus area as described below.

• Chapter 1: Introduction – This chapter gives an introduction about serverless computing and how it ties with the existing IoT infrastructures. In addition to that, the section also outlines the motivation and aim for this project as well.

• Chapter 2: State of the Art – This chapter details the various technologies and trends that are currently being used with relation to this project. It goes over the Internet of

Things (IoT), Industry 4.0, Kubernetes and serverless computing based on Kubernetes. Furthermore, it has an overview of the previous work that has been undertaken in this area.

• Chapter 3: Problem Statement – This chapter briefs about the problem statement that has been dealt with through this dissertation along with its scope and the challenges faced during the implementation.

• Chapter 4: Design and Technical Implementation – This chapter has two main sub-sections, one which details about the design of the system including the technology stack used, the architecture and other intricacies. The next sub-section details the actual implementation showing how the framework was setup through the use of a Kubernetes client, a local cluster and an MQTT server.

• Chapter 5: Results and Evaluation – This chapter details the results achieved during the evaluation of the framework in different scenarios and a discussion about them. In addition to that, there is also a comparison of the framework with some of the existing frameworks to get an idea of how well the framework performs.

• Chapter 6: Conclusion and Future Work – Lastly, this chapter gives an insight into the conclusions made from this project as well as the avenues where it can be further extended to.

Chapter 2

State of the Art

This section provides a detailed description of the current developments in the Internet of Things industry and the serverless computing space. It will pave way to the problem statement of this dissertation.

2.1 Internet of Things

The network of connected devices that includes a plethora of 'things' make up what is known as the Internet of Things (IoT). These devices can be anything from sensors, to wearables, to home appliances and even vehicles. The characteristics that set them apart from the usual device is that they are mostly present in resource constrained environments. Another thing to note is the IoT device deployments are always very dense in nature. Their deployments are expected to exceed over 26 billion connected devices at the start of 2021. They cross over several domains which includes but not limited to agriculture, healthcare, autonomous vehicles, smart home, etc. Common examples of communication between IoT devices can be doctors using next-gen pacemakers to keep track of the patient's heart rate and alter it based on the patient's activity levels or smart vehicles transmitting traffic information among one another to get a bigger picture of the congestion and traffic situation. This kind of data transmission can lead to a lot of data being generated. Certain characteristics about IoT devices set them apart from other devices and these will be discussed in detail below.

Resource and Power constraints – In terms of hardware capabilities, IoT devices have very limited memory and processing abilities. Since only very specific and low-level computations are expected to be performed in them, they have such a design. They are also either connected to a constant power supply in case of stationary devices or otherwise powered by battery if they are mobile in nature. In this scenario, the power is a limiting factor and has to be taken into consideration when performing computations. Such devices also have long sleep cycles to save power when they're not in use.

Diversity and dynamic in nature – In an IoT environment, the devices have very diverse capabilities in terms of their hardware and performance. This also leads to the network interactions being between different kinds of devices. In the case of IoT devices in a mobile network, the network is very dynamic in nature and keeps altering. This also affects the way in which the devices interact with each other.

2.1.1 IoT and Home Automation

The idea of an IoT device was to have a so-called 'thing' to be connected to Internet that allows in monitoring of the thing or issue some basic commands. The key technology in the creation of smart homes is a network of interconnected 'things' that include sensors, actuators, and other devices. The tremendous growth in wireless networks have enabled the creation of a globally accessible network of things that allow them to gain greater functionalities. This has led to the increased adoption of IoT devices and the concept itself in a wide range of fields such as medicine, the automotive industry, homes and more. The figure below highlights a basic smart home system based on IoT. It uses sensors connected to the cloud that send readings to the Internet and has an event-management system that can trigger it if the readings go beyond or below a threshold. This can be something like setting the light to turn on when the ambient temperature is too low or something like switching on the air conditioning based on voice input.

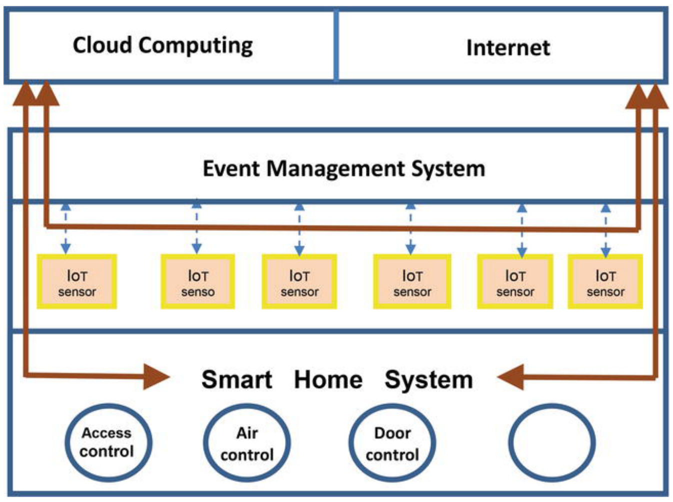


Figure 2.1: IoT Home Automation Workflow

The figure above [4] shows a high-level flow of a home automation system. Home automation systems have several advantages and that is also one of the reasons for their increasing popularity. They have the ability to be controlled and monitored over a lot of different mediums which include any kind of smart devices, most of the modern handheld devices, and computers. A lot of the security automation systems provide additional awareness and security to the homes and capability to monitor 24x7. They also provide benefits in terms of controlling temperature, lighting and heating which in turn helps in saving money to the users.

There has been a tremendous amount of IoT systems that have been initiated by scholars and there is a lot of academic work going on in this space as given in [5] [6] [7] [8]. In the following parts, we will take a look at some of the common wireless technologies and protocols that are currently in use as well as some that were popular few years ago.

Global System for Mobile Communication (GSM) – It is a set of standards developed for the second-generation of mobile networks. It operates under a very large area and uses the time division multiple access (TDMA) technology. It has a data transmission rate of up to 9.6Kbps. Improvements to the 2G standard, also known as 2.5G brought up the peak data rates up to 43.2Kbps.

Third Generation Cellular Network (3G) – This was an enhancement to the 2G networks by the addition of radio network controllers and Node-Bs (or base stations). Since the 3G networks are based on the Internet Protocol (IP), they provided a breakthrough in terms of delivering multimedia services. The peak data rate supported by the 3G networks were 2 Mbps. 3G networks also supported quality of service and further improvements to it led to the launch of 3.5G which offered higher peak uplink and downlink rates.

Long-Term Evolution (LTE) – The fourth generation of mobile networks offered a big jump in capabilities starting with the peak data rates which reached 100Mbps in downlink and 50Mbps in uplink. It has very low latency when switching from idle to active mode, of the order of under 100 milliseconds. It offers backward compatibility with 2G and 3G networks. The main technology in use here is Orthogonal Frequency Division Multiple Access (OFDMA).

Fifth Generation (5G) - The next big step in the world of Radio Access Networks (RAN) is the 5G networks that is aimed at bringing very high data rates and extremely low latency. The architecture of 5G [9] networks will be heterogenous consisting of macro cells, small-cells and also device-to-device networks. They will all have varying degrees of quality of service. The table below gives a high-level overview of some of the important characteristics of these mobile communication standards.

Generations	**2G**	**3G**	**4G**	**5G**
Start/ Deployment	1990 - 2004	2004 - 2010	2010 - Present	Present
Data rates	9.6 Kbps, 43.2 Kbps (2.5G)	2 Mbps	100Mbps	1 Gbps and higher
Multiplexing Variant	Time Division Multiple Access	Code Division Multiple Access	Orthogonal Frequency Division Multiple Access	Orthogonal Frequency Division Multiple Access
Services offered	SMS, Voice	High quality audio, video and multimedia services	Low latency information access, wearables	Low latency information access, M2M systems, IoT, wearables, AI-based predictions

Table 2.1: Comparison of 2G, 3G, 4G & 5G

Bluetooth – Bluetooth is a wireless technology meant for short-ranged data exchange. It can be done for fixed as well as mobile devices but to a limited distance. It works in the ISM bands between 2.402GHz and 2.4GHz making using of ultrahigh frequency radio waves. Bluetooth is extremely low cost, easy to be configured and installed and the current standards have speeds of up to 2Mbit/s. The disadvantage is that it is limited in range and only meant for environments like within the same house or room.

Zigbee – Zigbee [10] is a communication protocol that is mainly used for Personal Area Networks or PANs. Zigbee is also an IEEE 802.15.4 specification. It is targeted at devices requiring low power and having low bandwidth requirements. It is aimed to be simpler to use than Bluetooth and Wi-Fi. The cons with Zigbee are that it has a high cost of maintenance and also low data rates and stability.

Wi-Fi – Wi-Fi is a wireless protocol based on IEEE 802.11. It is a very common requirement these days and has led to it being very inexpensive. Due to its extreme popularity and low price it has led to several electronics manufacturers releasing their own do-it-yourself style kits that allow users to develop smart solutions from scratch that are Wi-Fi enabled. Unlike Zigbee and Bluetooth, Wi-Fi consumes almost 10 times more power and a lot of routers have the maximum number of concurrent device connections capped at 30. There is also a limitation in terms of the range and is usually limited within the same floor of the building unless there are repeaters that are placed to improve the range across the building. Being an open standard, Wi-Fi has been prone to a lot of security issues. The speed of Wi-Fi is also slower than wired networks. The advantage with Wi-Fi is that it is very easy to setup and expand. Since it is an IEEE standard, any manufacturer can bring up their own variant of Wi-Fi or just use the existing advancements with ease. It is become a necessity with mobile devices, smart devices, home automation systems, all having Wi-Fi support out of the box.

This section will look at a host of low-cost hardware components that have increased the popularity of home automation projects and its adoption. Arduino is an open-source, low-cost microcontroller unit with a large community backing it. Due to the same reason, there are a lot of projects which uses Arduino for home automation and there are guides and tutorials on how to do these as well. It is very easy to program in Arduino and it requires only very limited programming knowledge. It is however limited in functionality, so boards like ESP8266 and Raspberry Pi are better alternatives.

ESP8266 boards and ESP32 boards are comparatively more powerful than an Arduino and they have been used in several home automation projects as well [Add reference here]. These boards are also cheap, and they come with Wi-Fi enabled. The ESP32 board also has Bluetooth enabled and offers better hardware capabilities. Raspberry Pi on the other hand is a much more powerful MCU and it can run a complete desktop operating system on it without a problem. Due to the same reason and also its cheap price, they have been used to provide extensive capabilities in several home automation projects. Just like the Arduino and the ESP boards, most of the projects on the Raspberry Pi are open-source and are publicly available for the user to play around with and tweak to their requirements.

Specification	**Arduino UNO Wi-Fi V2**	**ESP8266**	**ESP32**	**Raspberry Pi 4 B**
SOC/ MCU	ATMega 4809	Xtensa Single Core 32-bit L106	Xtensa Dual Core 32-Bit LX6	Broadcom BCM2711 Quad Core
RAM	6KB	160KB	512KB	1GB/ 2GB/ 4GB
Wi-Fi	Enabled	Enabled	Enabled	Enabled
Bluetooth	No	No	Yes	Yes

Table 2.2: Comparison of Arduino UNO, ESP32, ESP8266 & Raspberry Pi 4B

2.1.2 Industry 4.0

Industry 4.0 [11] is the currently ongoing Industrial Revolution which seeks to automate a lot of the conventional manufacturing and industrial practices through the efficient use of the Internet of Things. Below is a list of the previous Industrial Revolutions.

Industry 1.0 – Before the current revolution, there were three other Industrial revolutions that took place starting from the 1780s. The First Industrial Revolution was with the invention of water and steam power. This led to a large improvement in the agriculture industry and also the mechanical industry.

Industry 2.0 – The main contributor in the second Industrial Revolution were machines that ran on electrical energy. Unlike steam and water energy, machines running on electrical energy were far more efficient and also easy to use and maintain. Industry 2.0 also witnessed the creation of assembly lines in the industry that led to mass production of goods through these assembly lines. On the managerial side, a lot of process improvements like labor division, manufacturing on the fly and lean management principles improved the quality of output and the process itself.

Industry 3.0 – The advancements in the electronics sector over the hind years of the twentieth century facilitated the commencement of the third industrial revolution, which resulted in the birth of Industry 3.0. Various electronic devices such as transistors and integrated circuits were invented and manufactured, and this led to significant automation of machines, which led to a reduction of effort, improved precision and agility, as well as the complete replacement of the humans in certain areas. The Programmable Logic Controller (PLC), which was initially developed in the 1960s, was a seminal technology that represented the beginning of electronic automation in the workplace. When electronics hardware was incorporated into the production processes, it led to the need of software technologies to ensure that these electronic devices functioned well. This in turn fuelled the growth of the software development industry. The software solutions, in addition to operating the hardware, also aided numerous management operations such as enterprise resource planning (ERP), inventory management, shipping logistics, product flow scheduling and tracking throughout the facility. Electronics and information technology (IT) were used to further automate the entire sector. Since then, the automation processes and software systems have progressed in tandem with the advancements in the electronics and information technology industries. Many manufacturers were driven to relocate to low-cost countries as a result of the push to decrease costs even further. The spread of industrial locations across the globe resulted in the development of supply chains and the widely popular Supply Chain Management concept itself.

Industry 4.0 – In the 1990's, the tremendous growth of the Internet and telecommunications industries changed the way people communicated and shared information. Additionally, it led to paradigm shifts in the manufacturing industry and traditional industrial operations, bringing the physical and virtual worlds closer than ever before. Cyber Physical Systems (CPSs) [12] have erased this line even further, leading to various industry-wide technological changes. These systems enable machines to communicate more intelligently with one another across virtually all physical and geographical boundaries. Industry 4.0 makes machines smarter by utilizing Cyber Physical Systems to share, evaluate, and direct intelligent actions for diverse operations within the industry. These intelligent machines can continuously monitor, detect, and forecast defects in order to recommend preventive and corrective actions. This enables industries to be more prepared and have less downtime. The same dynamic methodology can be applied to various facets of the industry, including logistics, production scheduling, throughput time optimization, quality control, capacity utilization, and efficiency enhancement. Additionally, Cyber Physical Systems enabled total virtualization of the industry, remote monitoring and management thereby providing a new outlook to the production process. It integrates equipment, people, processes, and infrastructure into a unified connected loop, resulting in a highly efficient management system. As the cost-benefit curve of technology continues to steepen, even more disruptive technologies will emerge at ever cheaper prices, redefining the industrial ecosystem. Industry 4.0 is still in its infancy, and industries are still in the process of transitioning to the new platforms. Industries must rapidly adapt new methods in response to these changes so that they can remain profitable. Industry 4.0 is maturing is expected to be there for at least the next 10 years.

2.1.3 Communication Protocols

One of the most crucial aspects of any IoT application is the messaging protocol in use. Devices in IoT systems are constrained in every aspect. They have limitations in terms of computing capabilities, storage, and power. Therefore, the message protocol in use must be lightweight, easy to use, and without much computational overhead. In this project too, there is a lot of communication between the sensors at various different locations, so these requirements apply here as well.

MQTT [13] was the messaging protocol of choice because it offers several advantages over other protocols. This dissertation will introduce MQTT as well as the other protocols in question and provide a comparative analysis between them to show why MQTT has the upper hand. The other protocols under consideration are CoAP, AMQP, and HTTP.

2.1.3.1 MQTT

MQTT expanded as Message Queue Telemetry Transport is a pub-sub (publish-subscribe) messaging protocol. Two entities perform the publish and subscribe operations. These are the clients and brokers. Clients usually subscribe to different kinds of content and they can also publish content to brokers. Brokers are in charge of sending published content to subscribing clients. It can operate in one-to-one, one-to-many, and many-to-many settings and has seen wide deployment in sensor networks, M2M systems, and IoT devices.

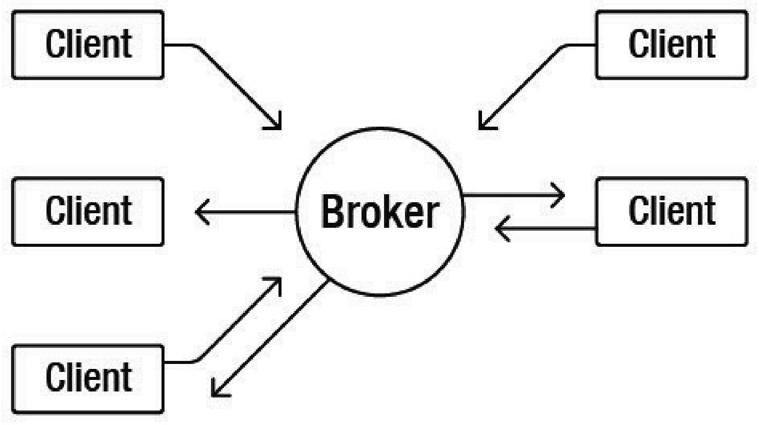


Figure 2.2: MQTT

The two main advantages of MQTT are that it is a very lightweight protocol and it is very straightforward to implement. The reason for the first advantage is because of the thin header structure that MQTT provides. This makes it easy to parse them and due to this, it is especially advantageous for resource-constrained devices, typical of an IoT setting. The size of the header for an MQTT message is just 2 bytes. It is easy to implement due to the fact that reliability considerations are offloaded onto the TCP protocol since MQTT is built on top of it.

MQTT also defines three QoS levels build on top of the underlying transport layer for better reliability.

QoS 0: This translates to data transfer without acknowledgment. At this level, the receiver will receive the message at most once, since there is no retransmission

QoS 1: With QoS 1, it is ensured that the receiver will receive the message at least once. This means that the sender will store the message until an acknowledgment is received. There is a retransmission of the message if no message is received after particular timeouts.

QoS 2: Lastly, we have QoS 2, in which it is ensured that the message is received exactly once. This is done through a two-step acknowledgment procedure, but it requires more states at both the sender and receiver.

MQTT uses TLS for providing security, but this is mostly applicable to non-constrained devices. Much more research and development are required in the area of security for MQTT as there are several scenarios in which security may be compromised.

2.1.3.2 CoAP

At the onset, CoAP expanded as Constrained Application Protocol, which is very different in operation compared to MQTT [14]. The design of the CoAP protocol was motivated by the REST protocol, which was based on HTTP. This allows it to interface through HTTP. REST uses Transmission Control Protocol (TCP) but CoAP uses User Datagram Protocol (UDP). This allows it to have lesser overhead. It has a packet header of size 4 bytes. Using UDP leads to less reliability in general, but CoAP makes use of two different types of messages to provide QoS levels.

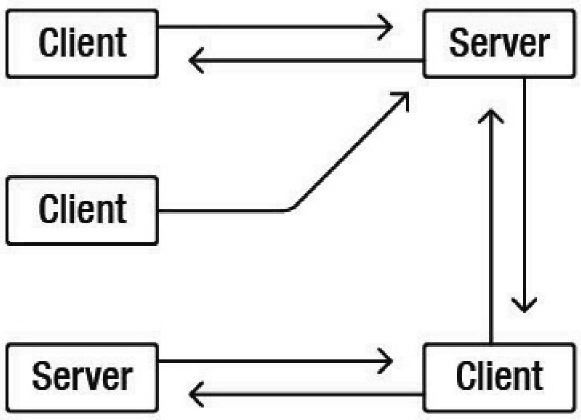


Figure 2.3: CoAP

Confirmable Messages [CON]: In this QoS level, there is a requirement for the receiver to send back an acknowledgment to the sender to confirm that the message has been received.

Non-confirmable Messages [NON]: This is similar to the QoS level 0 present in MQTT, wherein the sender simply sends the message but doesn't make sure that the receiver gets it.

Since CoAP was modeled after the REST protocol, it has the same 4 request types: GET, POST, PUT and DELETE. CoAP offers security through the DTLS protocol.

2.1.3.3 AMQP

AMQP expanded as Advanced Message Queueing Protocol, is another lightweight protocol used predominantly for M2M communications [15]. It was developed at JP Morgan Chase in London. AMQP supports both the request/response messaging model as seen with CoAP and HTTP as well as the publish/subscribe model as seen with MQTT. The header size of AMQP is the largest coming in at 8 bytes. Security in AMQP is provided by either SASL or TLS, or a combination of the two. AMQP provides three QoS levels, which are similar to MQTT. They are:

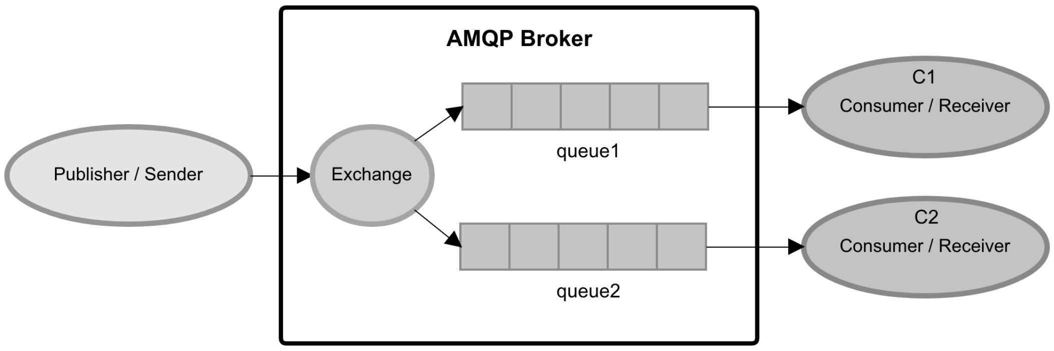


Figure 2.4: AMQP

At most Once: This level is again similar to QoS 0, present in MQTT. It doesn't require the sender to receive any acknowledgment of the message sent to the receiver.

At least once: Here, the sender requires an acknowledgment of the message from the receiver. If the acknowledgment is lost, it leads to a duplicate message being sent to the receiver.

Exactly once: Just like with QoS 2 in MQTT, here the sender and receiver use a two-stage acknowledgment procedure to make sure that the message is received exactly once.

2.1.3.4 HTTP

HTTP, expanded as Hypertext Transfer Protocol, is a messaging protocol, predominantly designed for the web. It uses a request/response model following the RESTful style of architecture. It operates over TCP/IP and provides reliable communication. But the problem arises because resources are accessed based off of IP addresses and URLs. These relationships change dynamically and communication is completed only after several stages of connection establishment and release. For the case of an IoT system, this leads to a very severe overhead in terms of the consumption of network resources.

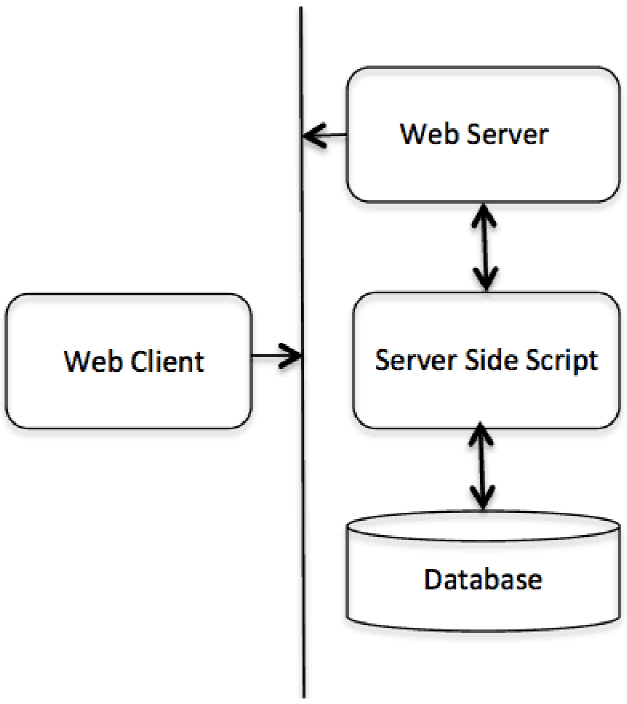


Figure 2.5: HTTP

HTTP does not define a particular size for the header nor does it do for the message size. Rather these responsibilities are offloaded to the web server that implements the protocol or the programming language in which it is implemented. HTTP also does not have any specific QoS agreements and these have to be explicitly supported.

Characteristics	**MQTT**	**CoAP**	**HTTP**	**AMQP**
Messaging Architecture	Client/ Broker	Client/ Server	Client/ Server	Client/ Server
Messaging Pattern	Publish/ Subscribe	Request/ Response	Request/ Response	Request/ Response OR Publish/ Subscribe
Header Size	2 bytes	4 bytes	Not defined	8 bytes
Message Size	Not defined (but small)	Not defined (but small)	Not defined (usually large)	Not defined (but negotiable)
QoS	QoS 0, QoS 1, QoS 2	Confirmable message, Non-confirmable message	TCP reliability	At most once, At least once, Exactly once
Message Type	Binary	Binary	Text	Binary
Security	TLS/SSL	DTLS	TLS/SSL	TLS/SSL & SASL

Table 2.3: Comparison of MQTT, CoAP, HTTP and AMQP

The table above effectively summarises the details mentioned in the previous sections [16]. At a high level we can see that MQTT is the most lightweight among all. It provides security using TLS/SSL and the message size is not defined but is usually small. CoAP has a slightly larger header size and it uses a similar approach to REST APIs. Its security is implemented using DTLS. AMQP has a large header size of 8 bytes and can work in both publish/subscribe model as well as request/response model. HTTP is the most widely used web protocol. It is a text-based protocol whereas all the others are binary. It has as request/response model and the message size is usually large.

2.2 Serverless Computing

With the demand and availability of virtual machines on the rise, cloud computing and in particular Infrastructure-as-a-Service (IaaS), have boomed in the recent past [17]. A huge chunk of enterprises have initiated or are already in the process of migrating their legacy applications to a cloud native tech stack. The percentage of expenditures on cloud products are expected to form 67% of the overall spending of an enterprise, and this itself is a testament to the growth and popularity of these offerings.

One of the main reasons for their increase in popularity is due to their pricing model. Instead of paying a fixed amount for the infrastructure, customers now need to pay only for the resources they use, when they use it. There are no upfront costs to almost all of these platforms. In the IaaS model, infrastructure provisioning and scaling were on the hands of the systems architect and system designers, but according to reports it is seen that most often that not, they over provision resources and end up paying more that what is being utilized.

Serverless computing stemmed from this burden and literally means server-less (or needing less concern for the servers). The earlier burden that came up the IaaS model now was handled by the service providers and the developers or system designers simply had to pay for the amount of resources being used based on events that trigger them. To give a standard definition for serverless computing, we can define it as a platform that lets developers write event-based executable functions and only needs them to pay for them based on the actual execution time and resource usage of these functions. The entire server aspect of these that includes resource allocation and scaling is handled by the platform itself. We will now look at two of the important features of serverless computing.

Pay-as-you-go -– One of the biggest downsides to Infrastructure-as-a-Service was that payment was to be made for the infrastructure that is purchased. This meant that system architects or system designers need to pay upfront for the infrastructure that they think they need. The programs or functions may never take up the entire resources they purchase, or they might need to purchase more if the demand goes up. In the serverless computing paradigm, functions are charged for the execution time only. This means that if a function takes a few milliseconds to be completed, the organization is billed only for the few milliseconds that the function executes. They also have an attractive pricing model with free tiers giving a lot of free requests every month. For organizations that require only execution to be done occasionally, this is extremely lucrative.

Elasticity – The fact that developers don't need to worry to about how many CPUs, or memory their functions require give them a lot of flexibility to actually focus on the things that matters. In this case, the business logic and their requirements. The cloud service providers make the decisions on how many CPUs are required to execute the developers' programs and how much memory is to be allocated. The CSPs also take care of a host of other aspects like reliability, monitoring, logging, routing maintenance and upgrades to their platforms.

With recent advancements[18] in container technologies and VMs, serverless computing fits the bill to be obvious next step. The original idea of not requiring to have servers, can be traced back to the age of peer-to-peer networks or software, but the actual serverless computing paradigm was first unveiled by Amazon at their re:Invent conference. At that time, it was the only available serverless platform, but fast-forward it to today, there are multiple cloud service providers like Google, Microsoft, IBM and also open-source serverless computing frameworks as well as academics coming out with their offerings. All of these advancements have decreased the development and deployment times and reduced the resource footprint of the applications itself.

Platform-as-a-Service is being is another cloud computing paradigm that is being offered by the likes of Google, Heroku and Cloud Foundry. Serverless computing platforms can be thought of as a development of this paradigm where the former lets the user deploy applications on to a hosting environment that is offered by the provider using the provider-specific tools and programming languages and styles. The consumer does not manage or control the underlying cloud infrastructure, which includes the network, servers, operating systems, or storage, but he or she does have control over the installed apps and, in some cases, the configuration of the application hosting environment.

With serverless computing growing and improving upon core computing concepts like publish-subscribe messaging systems, event-based computing, distributed systems and more, it is definitely here to stay. In the next section, we will walk through the three most popular serverless computing platforms in the market today from Amazon, Google and Microsoft.

2.2.1 Amazon vs Microsoft vs Google

Each of these three tech giants have their very own serverless computing platform. The first one was Amazon's Lambda, which came out in 2015. The next one in line was Microsoft's Azure Functions in 2016 and lastly, we have Google's Cloud Functions which came out in 2018. These three offerings have changed the serverless space by leaps and bounds thanks to the Function-as-a-Service (FaaS) paradigm introduced by them [19].

2.2.1.1 Comparison

Price - First off, we will start by looking at the pricing model for the three services [20] [21] [22]. The biggest advantage with FaaS platforms is that they are extremely cheap. While this being said, the pricing scenario can dramatically change if the scale goes up. All the platforms compute the pricing based on the number of requests and resources used. Below is a small summary of the three platforms and their pricing models.

Service	**Free Requests/ Month**	**Compute duration (gigabyte-seconds)**	**Cost per additional 1000K requests**
Lambda	Client/ Broker	Client/ Server	Client/ Server
Azure Functions	Publish/ Subscribe	Request/ Response	Request/ Response
Google Cloud Functions	2 bytes	4 bytes	Not defined

Table 2.4: Pricing Model Comparison

In the above price comparison, it is pretty evident that the pricing is almost near identical in all three service providers. It is also interesting to note that these statistics are for the free tier of each of these platforms.

Languages Supported – The number of programming languages have considerably increased compared to when it was just C/C++ and Java. With varied strengths and weaknesses for every language, developers have taken a keen interest in experimenting and Cloud Service Providers have also started supporting more and more languages on their platforms. Below is a comparison of the languages supported by these three public Cloud Service Providers.

Service	**Languages Supported**
Lambda	Java, Python, Node.js, C#, PowerShell, Ruby
Azure Functions	Java, Python, Node.js, C#, F#, PowerShell, TypeScript
Google Cloud Functions	Java, Python, Node.js, C#, F#, Go, Visual Basic, Ruby

Table 2.5: Languages Supported

Time of Execution – While serverless functions are aimed to be executed in seconds and mostly are stateless and ephemeral, there might be cases where machine learning models are to be executed and inferences be made. In such scenarios, the functions might take minutes or even hours to complete. The table below shows the execution timeouts that are supported in these platforms.

Service	**Execution Timeouts**
Lambda	15 minutes
Azure Functions	5 to 30 minutes
Google Cloud Functions	9 minutes

Table 2.6: Execution Timeouts

Memory Configuration – Memory is a huge factor when scaling up serverless functions. There is a fine line between paying too much for unused memory and having to deal with slow execution or timeouts due to lack of memory. The table below shows the memory configurations available in the three platforms. It is evidently clear that Azure Functions lead the way here with higher memory configurations available.

Service	**Memory Configurations**
Lambda	0.128GB – 10.24GB
Azure Functions	0.128GB – 14GB
Google Cloud Functions	0.128GB – 4GB

Table 2.7: Memory Configurations

Another point of evaluation is the cold start latency. Cold start is usually referred to as the time taken for a function to execute after the event has been received. None of the Cloud Service Providers have revealed their information regarding the cold start latency but there are certain statistics that are available after tests done by analysts. According to these results, Lambda functions have a cold start latency of less than a second, while Google Cloud Functions have a cold start latency between half a second and 2 seconds. Azure Functions, however, has the largest cold start latency at greater than 5 seconds. As mentioned before, these are not figures published by the Cloud Service Providers themselves, but approximations from third-party tests.

2.3 Abstracting Hardware through Virtualization

One of the main core concepts behind serverless computing is that of virtualization. The concept has its roots at IBM and started in the late 1900s. It was done initially on the M44/44X computer system at the Thomas J. Watson Research Center in New York. The naming scheme of the system refers to the IBM 7044 systems and the 44X refers to the 7044 virtual machines that were run on it. From there, virtualization has grown by leaps and bounds. It allows developers to make use of system resources in an efficient manner and it reduces cost of operation. The virtual instances run completely isolated from each other and this also allows in rapid scalability. There are two broad categories of virtualization that are popular in the market these days. They are hypervisor-based virtualization, which uses a hypervisor and container-based virtualization, that uses containers. Both of these will be detailed below and we will be using container-based virtualization in this dissertation through the use of Docker and Kubernetes.

Hypervisor-based Virtualization – A hypervisor is a software which works like an emulator. It runs on a system, which is referred to as the host system, and allows to logically abstract itself to the other operating systems executed on it, as if they were running using the actual hardware resources themselves. With the help of a hypervisor, the host system is now able to run multiple operating systems on it by sharing its resources. This was a breakthrough technology and it allowed servers to run multiple operating systems where they actually had to run one operating system on a physical server prior to it.

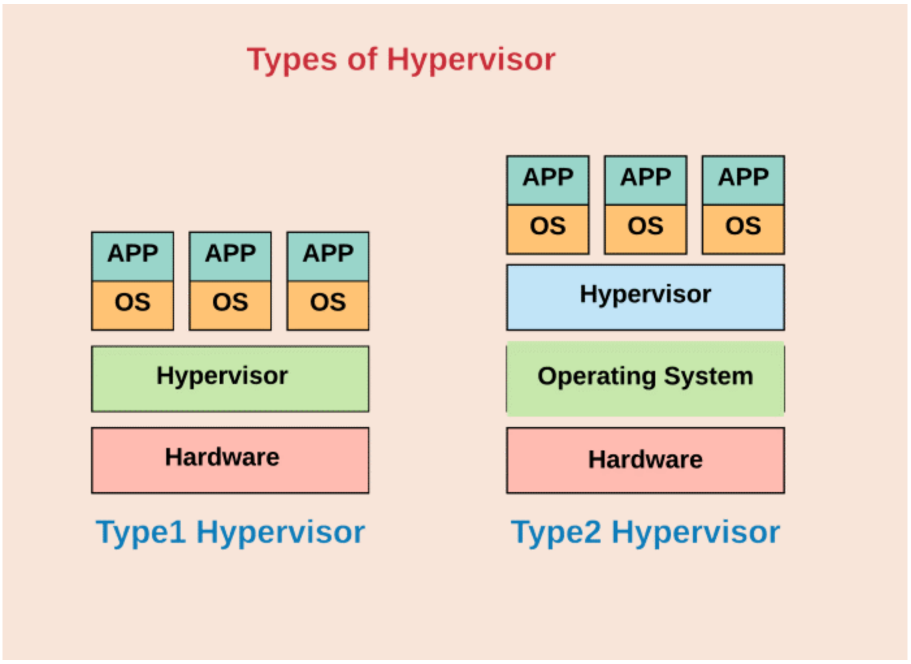


Figure 2.6: Type 1 and Type 2 Hypervisors

This is the core technology concept that is used in cloud computing platforms to virtualize resources and provide them as services to the customers. There are two kinds of hypervisors. Type 1 hypervisors run directly on the hardware of the host system. Type 2 hypervisors [23], however, are installed as an application on the host system. They don't provide as much performance compared to Type 1 hypervisors, since it runs directly on the hardware of the host system.

Container-based virtualization – This is an alternative method to virtualization through the use of containers. It is also known as operating system level virtualization or OS-level virtualization. Containers create what is known as user spaces on top of the same OS kernel. These user spaces run isolated from one another. As shown in the image below, you can see that virtualization through a hypervisor is much more resource intensive as it requires full blown operating systems to be installed on each virtual machine. Containers on the other hand run isolated from one another and use the same kernel of the operating system that the host machine is using.

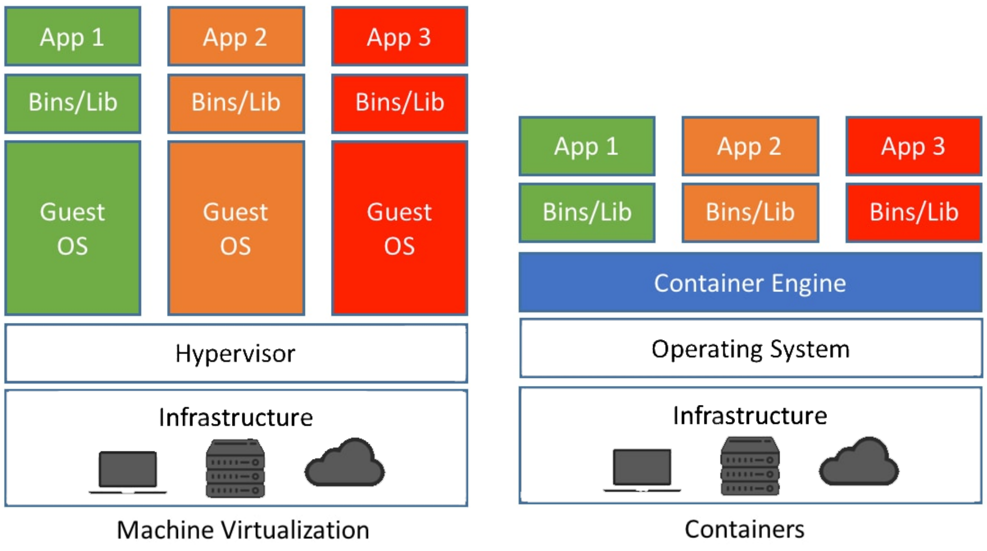


Figure 2.7: Virtual Machines vs Containers

Container-based virtualization has a lot of advantages when compared to hypervisor-based virtualization. One of the biggest advantages is speed. Since containers don't require entire operating systems to be installed, they just need the runtime libraries and the code of whatever is needed to be executed packaged together. This makes it extremely lightweight also. There are Linux-based containers of BusyBox and Alpine Linux coming at 1.24 megabytes and 3.54 megabytes.

Since there is no operating system installed and has only the runtime libraries and the code to be executed, containers spin up very quickly. Another huge advantage with container-based virtualization is that resources are not locked to any container when executing. The containers can leave resources to other containers when not in use. This is not the case with virtual machines, where the resources are bound to a particular virtual machine. The only con for the container-based virtualization is that since every container is sharing the same kernel, there is slightly lesser isolation in comparison to the virtual machines [24].

2.4 Docker & Containers

As mentioned on the Docker website, it is a completely open-source tool that is used for building, shipping and deploying containers. The Linux kernel offers the ability to isolate resources and this is implemented through concepts like namespaces and cgroups. Docker makes use of these concepts present in Linux to operate. Unlike traditional applications, containers can be deployed at a very faster rate. The Dockerfile allows developers to code up infrastructure elements like network, memory and CPU. The process can also be done using the command line as well. What this does is that it helps developers to manage infrastructure resources like how they would manage a normal application.

At the end of the day, the main pain point that Docker tried to tackle is the time to market for an application. It has been able to do so and a plethora of enterprise organizations are using Docker to run their applications in isolated containers. The success of Docker lies in the fact that it can run almost any application in an isolated container and also allows running multiple containers, each again isolated from one another. Due to the isolated nature of the execution of these containers and the security features provided by Docker, they can all be run on a single host. Coupled with the microservices architectures, Docker[25] makes it possible to deploy multiple instances of different sub-components of an application, without any constraints of the programming language, host or framework used. Due to the fact that the containers themselves are very lightweight, it promotes easy distribution of an application.

2.4.1 Docker Architecture

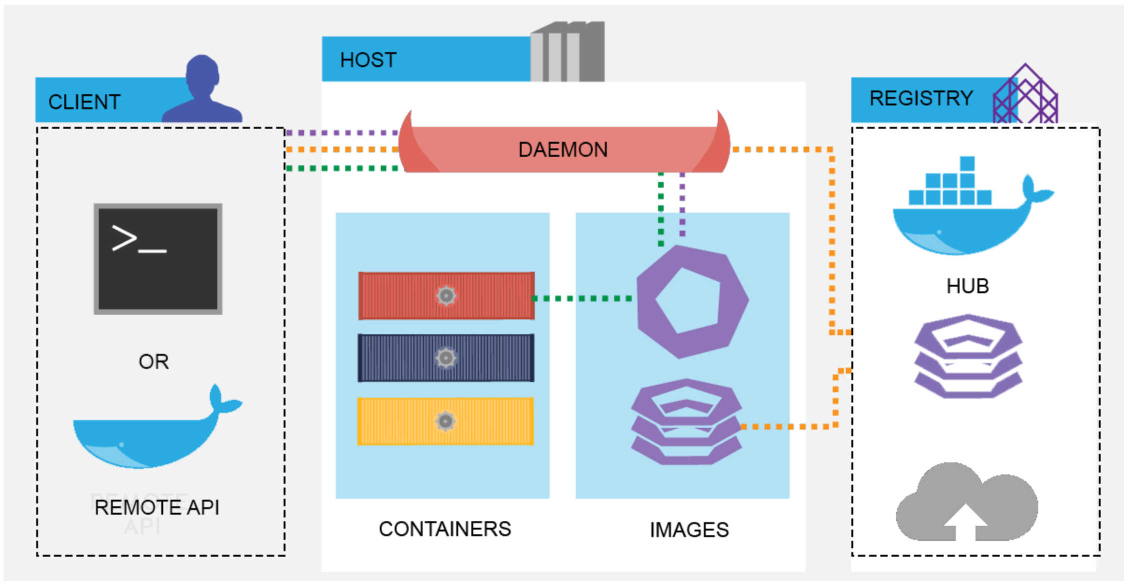


Figure 2.8: Docker Architecture

The above figure illustrates the architecture of Docker [26]. The main component of the Docker architecture is the Docker Engine which takes on the responsibility of running Docker containers as well as creating them. They follow a client-server architecture as shown above. Users or clients interact with the Docker engine through the Docker client or the command line. Below we describe each of these components.

Docker client – The Docker client is the primary point of contact between the users and the Docker platform itself. Users can interact with the Docker client using a command line interface and use Docker specific commands. These commands are then sent to the Docker platform, in particular to the Docker daemon which runs the commands. The Docker client facilitates all communication between the user and the daemon through a Representational State Transfer (REST) API. It is also possible to have the client and the daemon on different hosts, as the communication is through a REST API.

Docker daemon – The main point to note with the Docker daemon is that users cannot interact with it directly. The daemon runs on the host machine and manages all the Docker containers present in the system. The daemon runs on the host and waits for the requests that come as API calls from the docker client.

Docker images - These contain the code that pertains to the creation of execution of the container itself. Docker's foundation lies with containers. The image consists of runtime libraries and other binaries used to build and run the programs. Docker has a wide community that lets developers download images from other developers or also create their very own Docker image. Docker uses a file system known as UnionFS, which builds the Docker image in layers. These layers are represented as steps in the Dockerfile, which is a stepwise declaration of what is needed for the program to run. The biggest advantage of Docker images is that whenever there is a change in the image, Docker only rebuilds the layer that pertains to the change and adds it to the already existing layers. Docker keeps the layers as a cache so that each time a change is made, the rest of the layers are taken from the cache and the new layer is just added on top of the existing image.

Dockerfile - Dockerfile is a simple text file containing the necessary content to create a Docker image. The Dockerfile is then used by the developer to create the image by using simple Docker commands. To build an image, we use the docker build command. If there are any files to be used within the Docker image, we place them in the same directory as the Dockerfile. Docker iteratively goes through each step in the Dockerfile and checks whether any images in the cache can be used to replace the layers to be created. The steps in a Docker file start with the 'FROM' instruction, which chooses the base image. For the scope of this dissertation, we have used Alpine Linux, which is a lightweight Linux distribution that weighs under 6 megabytes. Once the base image is ready, then the following steps, add the other dependencies or instructions to add files, move files or remove files or execute commands.

Docker registry - The images created by developers are stored in what is known as a Docker registry. There are two kinds of registries, public and private. The most common registry in use today is the Docker Hub which is a part of Docker itself and is a place where developers can create and push their images to. Docker Hub itself allows the creation of public and private repositories to store the images, wherein private repositories don't allow unauthorized people to access the images. For this dissertation, we used a private Docker registry that is linked to the Kubernetes cluster in the localhost and not in Docker Hub. This allowed easy communication between the cluster and the registry thereby reducing the latency to pull down the image to execute the functions.

Docker containers - Lastly, we have the most important part of the Docker architecture and that is the Docker container. The Docker container is the running instance of the images we mentioned above. Once we create a Docker image from the Dockerfile and push it to a Docker registry, we need to execute them. This is done through the Docker client. In my case, I used Docker desktop, which allows execution of commands from the command line. It also has a simple Graphical User Interface (GUI) which can be used to run the containers. After pulling down the image from the registry, we simply need to click the play button next to the image on the GUI. The Docker client can be used to run, stop, delete containers. Multiple instances of the same image can be run on the same host and these will run without any interference with one another. Docker containers themselves are stateless, and if there needs to be some sort of persistence, we need to link it with a persistent storage.

2.4.2 Orchestrating Containers

Due to the increased popularity in deploying applications through containers, it became extremely vital to manage the whole process. Docker made it very easy to deploy applications, package and transport them with a very tiny footprint. Distributed systems also embraced the introduction of containers with multiple containers interacting among each other spanning different geographical locations. This was one of the strongest reasons for needing a container management tool. The container management tool, or also commonly referred to as the container orchestration tool, does all the functions throughout the lifecycle of a container. This includes starting containers, stopping containers, scaling them based on the traffic, patching them with updates, provisioning hosts, open up ports on the containers to allow outside network access and more. The most popular container orchestration framework in the market right now is Kubernetes and there is also a tool by Docker itself known as Docker Swarm. This dissertation used the Kubernetes orchestration tool to manage the containers and also execute the serverless functions on the fly due to the increased support available online and its increased popularity.

2.5 Microservices

With top technology companies embracing the microservices architecture, applications are being continuously migrated or created using the same. In the traditional monolithic architecture, applications and its associated libraries and components were compiled into a single unit and scaling it meant that the entire single unit needed to be deployed multiple times. In the microservices architecture, the application itself is broken down into independent components that are small [27]. They communicate with each other and are deployed as services. The communication is usually done over HTTP. The figure below compares a traditional monolithic application architecture with that of a modern application that uses the microservices architecture.

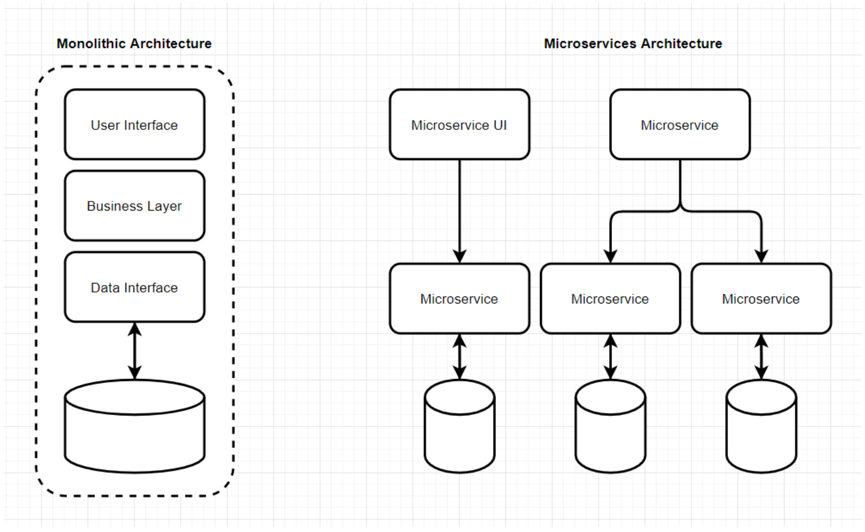


Figure 2.9: Monolithic Architecture vs Microservices Architecture

The monolithic application follows a very stringent way of including all the application components. What this means is that it doesn't consider the varying system resource requirements of the different components that make up the application. A huge disadvantage for a monolithic application is the size of the codebase as well as the difficulty in understanding it. Due to the extremely large size in the codebase, the application build time will also be greater. Making small changes will be a hassle with a large application as well since it needs to be completely rebuilt and redeployed. Lastly, another disadvantage that is still causing issues with major enterprises is that of technology lock-in. A common example is that of banking companies where their entire application is built on a legacy technology stack in a monolithic fashion and since it needs to be resilient against cyber-attacks, they need to migrate to a newer technology stack. This will require a considerable amount of time and effort from the company to completely modernize the legacy application to a microservices architecture.

To build and scale large enterprise applications efficiently, the microservices model has been proposed as the way to go. The biggest advantages of using this architecture are as follows:

Language independent – Each microservices in an application can use any language that best suits the requirement and the developer proficiency.

Easy to scale and deploy – Since each component is independent, they can be scaled independently as well. If there's a particular service that is handling payments and there is a huge spike in purchases on a website, they can simply deploy more instances of the payment service and not the entire application.

No single point of failure – Since it is a distributed deployment and the application is made up of independent, individual services, there is no single point of failure and the whole application needn't be taken down in case there is an outage.

In the section above, we saw what the advantages of using a microservices architecture are, but there are still some challenges with this model. As we mentioned about not having a single point of failure, it also makes it difficult to isolate the error. This can definitely be solved using a uniform logging mechanism, like using the ELK stack, which will help in isolating the erroneous service. Another disadvantage is that of network communication, which is slower than in memory calls. It also creates that additional dependence of having a stable network connection for the application to function properly. In case of a large enterprise application, there will be a number of services running in parallel, and the more the number of services the higher the chances for errors. While it makes it easy for testing each service individually, the integration tests get complex as the application grows in size.

2.6 Kubernetes

Google pioneered the art of running their applications in containers and nearly every application at Google were run in containers. To manage these containers, Google initially used a system called Borg, which later became the foundation for the Kubernetes project. As the usage of containers started growing, Google partnered with the Linux Foundation and released Kubernetes v1.0 in 2015. The entire Project Borg was written in C++, but the rewritten Kubernetes is written completely in Go, which is more performant and modern. Currently, Kubernetes is managed by the Cloud Native Computing Foundation (CNCF) and it is open-source. In the subsequent sections, we will go over the different components that make up the architecture of Kubernetes [28].

2.6.1 Kubernetes Architecture

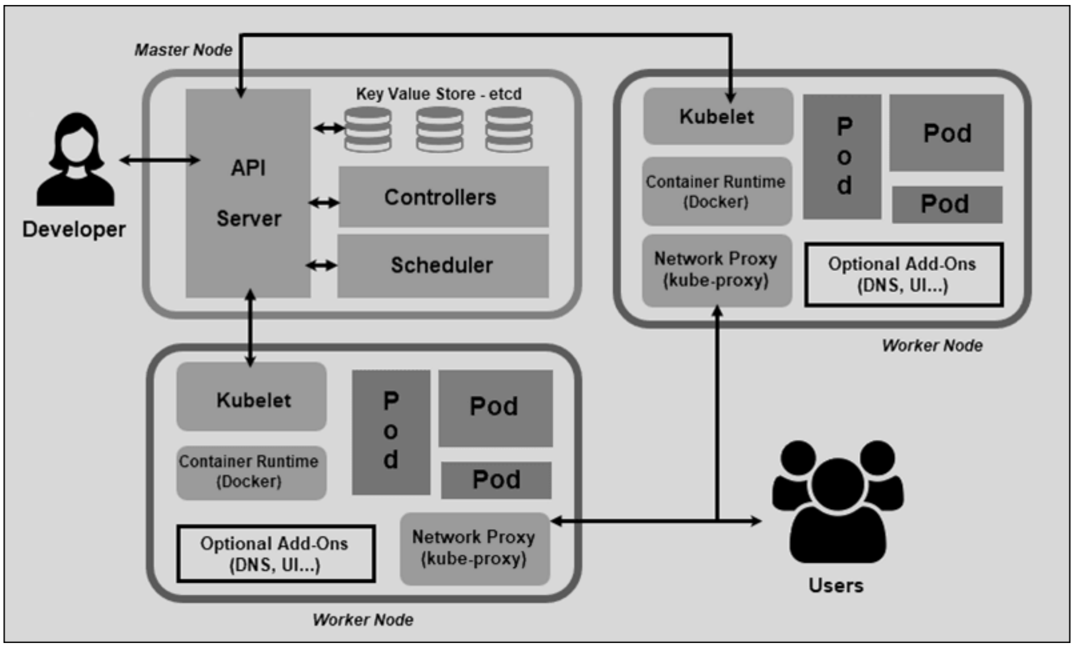


Figure 2.10: Kubernetes Architecture

1. Pods - The smallest deployable unit in Kubernetes is the pod. They can contain one or more containers and each of these containers in the same pod share the same resources like port range, IP addresses and storage. The containers are co-scheduled and co-located thus making them tightly coupled. Each time a pod is created, a unique IP address is assigned to it by Kubernetes. The multiple containers within the pod, communication through the localhost. Inter-pod communication is done through the unique IP addresses that are assigned at the time of their creation. It is important to note that pods lack any kind of durability, and developers need to keep them stateless. Another point to note is that Kubernetes orchestrates and schedules pods themselves and not the containers within it. Therefore, the application will need to be packaged as a pod by the developer.

2. Master and nodes – The naming conventions of Kubernetes has changed a little bit from the time it started. The broad categorization of components in Kubernetes is that of the master and the node. The nodes were referred to as minions before and they are also referred to as worked nodes sometimes. The control plane of the Kubernetes cluster is the master and it is usually deployed on a highly available cloud or on-prem system, either a virtual machine or a barebone system. All the work on a Kubernetes cluster is done on nodes and not on the master. These worker

nodes are managed, scheduled and orchestrated by the master. The master itself is made of up several different components and they are listed down in detail below.

(a) *API Server* – One of the most important components of the master is the API server. It is the frontend of the Kubernetes cluster with which the developers interact and all communications to the cluster are done. In Kubernetes, there is an API versioning system and they are all made to scale horizontally. The communication between the master and worker nodes are taken care of by the API server. Developers can interact with the cluster through a command-line tool such as kubectl or kubeadm, and they in turn use the API server to facilitate the communication.

(b) *Controller Manager* – There are several controllers in the Kubernetes ecosystem. This includes the service account controller, node controller, endpoint controller and the replication controller. They are control loops which run on the master node. Their function is to check the current state of the cluster through the kube api server and then compare it with the desired state to which the cluster should be moved and moves them. A common use case is that of the replication controller that looks at the number of pods running in a node and then adds more pods if the number doesn't match what is mentioned in the configuration. To check the availability of nodes we use the node controller. For joining together pods with services, we make use of the endpoint controller. The service account controller creates API access tokens and default accounts.

(c) *Scheduler* – The main task of a scheduler is to assign pods to appropriate nodes. This is not done on a random fashion but by considering several factors like the resource consumption and the node's availability for new pods. The scheduler has the right to remove pods from a node if it leads to jeopardizing the state of the cluster.

(d) *etcd* – etcd is the only data store present in the Kubernetes cluster. This is a very highly available data store that stores data as key-value pairs. It is inherently very lightweight and also distributed. It uses the Raft Consensus algorithm to maintain strong consistency and availability. It is an open-source project in itself and was a part of the now discontinued CoreOS.

(e) *Cloud Controller Manager* – Just like the controller manager the cloud controller manager provides a control loop to execute cloud-specific code within the Kubernetes cluster. This abstraction was added in Kubernetes v1.6 to provide an independent lifecycle for the cloud components. This further allowed popular cloud service providers to create their very own cloud controller manager implementations including the likes of Amazon, Google, Microsoft, Oracle and more.

(f) *Add-ons* – These are pods and/or services that add on to the functionality of the Kubernetes cluster. There is an add-on manager that manages and creates add-ons for the cluster. The simplest example of an add-on is a web-based UI dashboard that is used to view and manage the different objects in Kubernetes.

Every node in a Kubernetes cluster has what is known as a node component. These components are responsible for providing a runtime environment for the pods that execute within the node. The node components in a Kubernetes node are the kube-proxy and the kubelet.

(g) *kube-proxy* – It is one of the most vital components of a node. It is a reverse network proxy that provides capabilities like load balancing and also bridges containers to services. As mentioned above, it runs on every node. Since they are very simple implementations, they can forward only UDP or TCP traffic or do a simple round robin load balancing across different backends.

(h) *kubelet* – The Kubelet is by far the most vital component present in a node. It makes use of the PodSpec configuration document, that is written usually in YAML, but also possible in JSON, to describe the pods. The kubelet process can create, monitor and destroy containers that are running on any node. The PodSpec configuration is provided to the kubelet by the kube api-server through an HTTP endpoint or can be provided as a file path. The kubelet process then runs periodic checks to this endpoint or file path. One of the main responsibilities of the kubelet is to perform tasks as mentioned by the master.

3. Replication Contrller - The resiliency and fault tolerance for the application is provided by the replication controller that keeps a track of the number of instances of pods to be run in a node at the same time. The different instances of the pod are known as replicas. There are certain rules that advocate how many number of pods are to be run at the same time and this is present in a pod template. This is then used by the replication controller to keep the number of pods running as mentioned in it. The replication controller can delete or create replicas as and when needed to keep the cluster state in sync with the pod template. There is a possibility to create pods without attaching it to the replication controller but this is not recommended as pods can stop or get killed unexpectedly and if there is no replication controller

attachment, it might not get restarted or instantiated appropriately. When deleting the replication controller, it does not kill off any of the pods attached to it.

4. Services - Services make sure that the pods are accessible to the users and provide functionality. Unlike the replication controller, services don't need to worry about the number of instances running to provide better fault tolerance and resilience. Like mentioned above, every pod is assigned an IP address, but since the replication controller kills or creates pods based on the pod template, it will be difficult to access these pods based on their changing pod IPs. To solve this problem, Kubernetes introduced the concept of services which act as a layer of abstraction over a set of pods. They have policies that define how to access the pods and these are usually done through what is known as a label selector. Without the label selector, the service can be mapped using a DNS name or a specific IP address as well. Therefore, there is no need to get constrained by the label selector itself. These services themselves are RESTful and they have an IP address. At the time of creation, the service object also creates an endpoint object that hosts the pod details using the label selector which can be found in the service template. The endpoint object automatically updates itself each time when a pod or pods linked to a particular service undergoes any change.

2.6.2 Serverless Frameworks Based on Kubernetes

Open-source platforms have given developers the liberty to plug and play multiple services within their application to give robustness and reduce the time of development. Being open-source there is also a large community backing such software or frameworks. Even though they come with their advantages, they have their own disadvantages as well. Open-source software has a learning curve to it. Not everyone can easily start setting it up on their system or integrate it with an existing project without properly getting to know the framework and all its prerequisites. Another disadvantage is the technical support available. It is agreed that if there are a lot of people using the framework or software, that it will be robust, because the more people use it the more issues are reported and the faster, they are fixed. Again, there is the problem of who prioritises these fixes, since one person's priority might not the be the other person's priority. Kubernetes is an open-source container orchestration framework and it is managed by Google and the Cloud Native Computing Foundation. While we discussed about frameworks like Lambda, Azure Functions and Google Cloud Functions, there are also open-source serverless frameworks that are based on Kubernetes to provide serverless capabilities. Most of these frameworks use the Kubernetes APIs to create, deploy and execute serverless functions. The most common implementation is through the use of a Custom Resource Definition (CRD) [29] which creates a deployment or pod for the function execution. Below is an overview of some of the most common open-source serverless frameworks based on Kubernetes.

2.6.2.1 Kubeless

Kubeless [30] is an open-source serverless computing framework provided by Bitnami. It is available under the Apache 2.0 license. Kubeless relies on Kubernetes to perform autoscaling, monitoring, routing, etc. There are three different trigger mechanisms available in Kubeless. They are pub-sub based triggering, scheduled triggering and HTTP-based triggering.

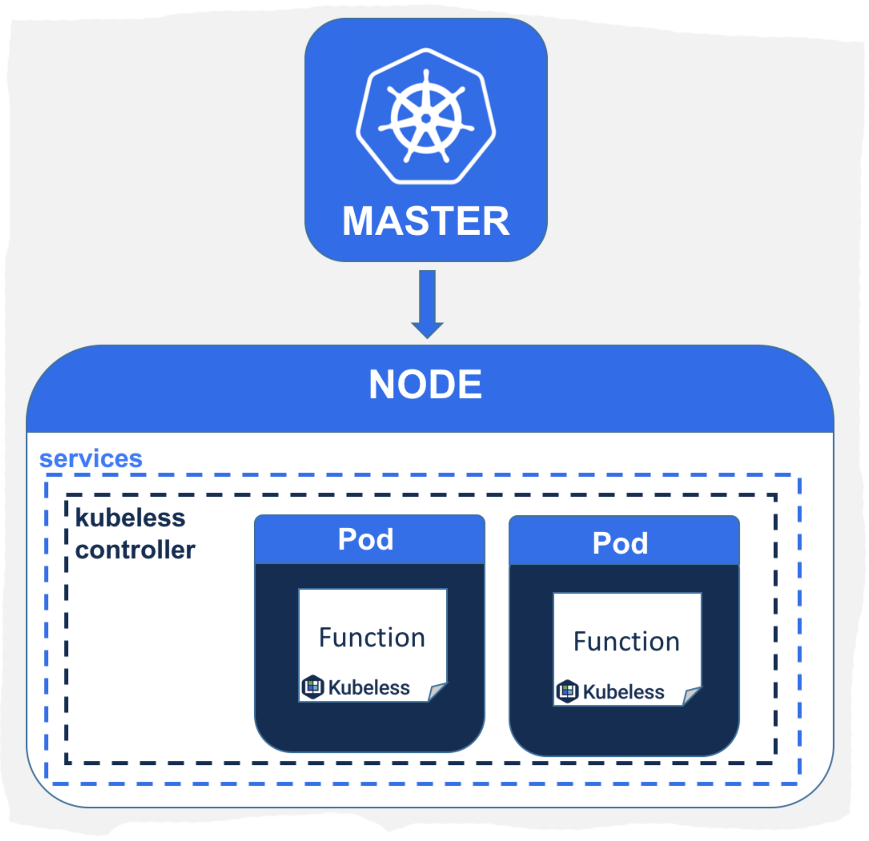


Figure 2.11: Kubeless Architecture

Kubeless makes use of CRDs in Kubernetes which creates Kubernetes resources, which in turn are the functions to be executed. A Kubernetes controller, which watches over resources, watches over this CRD and execute based on triggers. A Kubernetes deployment or pod is created when the function is triggered and a Kubernetes service exposes these functions to the outside world. Kubeless supports a host of language runtimes. This includes Python, PHP, Go, .NET, Ruby, Ballerina, Node.js. Kubeless has a very intuitive user interface called the kubeless-ui which is very similar to what you get with the AWS Lambda console. It lets you mange and deploy serverless functions with ease.

As it is a Kubernetes native serverless framework, it uses Kubernetes to do most of the heavy lifting. In terms of autoscaling, Kubeless uses the Horizontal Pod Autoscaler available in Kubernetes. The Horizontal Pod Autoscaler checks for CPU utilization and other metrics to scale up or down. As mentioned above interactions with Kubeless can be performed with an intuitive UI or there is the command line interface (CLI) for those who like to directly type in to a terminal.

2.6.2.2 Nuclio

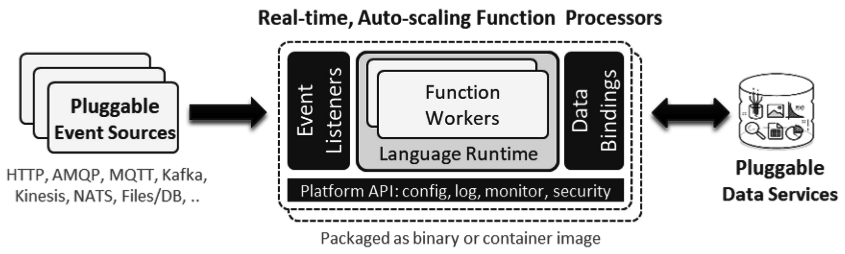

Real-time, Auto-scaling Function Processors
Pluggable Event Sources
HTTP, AMQP, MQTT, Kafka, Kinesis, NATS, Files/DB, ..
Event Listeners
Function Workers
Language Runtime
Data Bindings
Platform API: config, log, monitor, security
Packaged as binary or container image
Pluggable Data Services

Figure 2.12: Nuclio Architecture

Nuclio has a core component known as the function processor. This is like the operating system for the functions and feeds them with the events from multiple pluggable sources like HTTP, messaging queues and event streams. The two main components within the architecture are the event listener interface and the worker processes. The event listener is what listens to the external events through the different sources mentioned above. It can be even configured to allow different timeouts for queuing up of events. The worker processes can be configured as a static parameter that will let a certain number of function executions to happen every time. There is a data binding interface which takes care of making available external data to the system. This interface takes care of the necessary connection, security and caching of external data.

Nuclio is built with speed in mind and they also boast to the extend saying that writing functions on bare metal hardware will be slower compared to their function processor. The function processor has the ability to process over 400,000 function invocations a second when using the Go language or up to 40,000 events if using Node.js or Python [31].

2.6.2.3 Knative

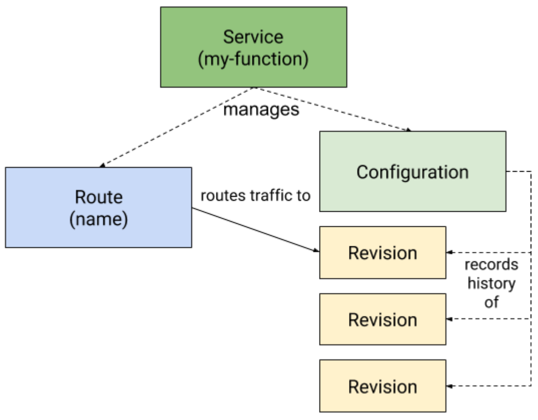


Figure 2.13: Knative Architecture

Knative [32] itself is a combination of a lot of open-source projects from the Cloud Native Computing Foundation (CNCF) [33]. Knative has an API called kn which is exposed to CRDs and operators from Kubernetes. This is then used to deploy the developers' applications or functions. Kubernetes creates the resources for these in the background and this can be anything from pods, deployments, services, etc. It should be noted that these resources are not created immediately but rather a virtual endpoint is exposed. When the event is triggered after hitting this virtual endpoint, only then is the actual resource created and executed.

The main component of KNative are the function pods that contain the actual function itself and a queue proxy. All the incoming requests hit the queue proxy which is then forwarded to the function container. The queue proxy manages the queuing mechanism as well as timeout of the incoming requests. Since there is an additional overhead of actually sending the requests through a queue and then to the function, it has slightly lower throughput compared to the other serverless computing frameworks. Knative, however, has one interesting feature known as the panic mode. This is an autoscaling mechanism that has better responsiveness to spikes in traffic and can scale up the pod instances by up to ten times or as mentioned in the configuration.

2.6.2.4 OpenWhisk

IBM developed OpenWhisk [34] initially and then open-sourced it by sending it over to the Apache Incubator. It is also the backbone of the current serverless computing platform present in IBM Cloud called Bluemix. The three main primitives in the OpenWhisk architecture are actions, triggers and rules. Actions are basically the serverless functions that get executed. They are stateless according to OpenWhisk. Triggers are events that cause these functions to be executed. Triggers can originate from different sources like we've seen on other open-source serverless platforms. In other words, based on the trigger, an action is performed. Rules are primitives that chain together actions and triggers. When there is a trigger, if a particular rule is present to perform an action, it will be performed.

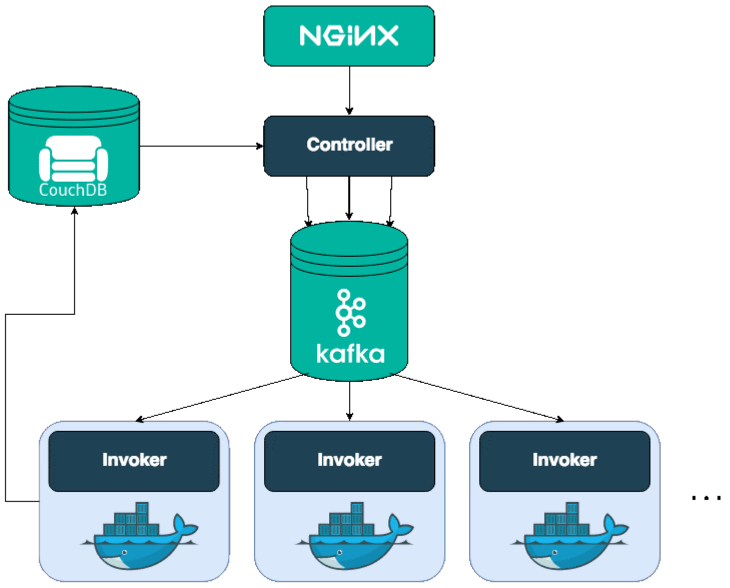


Figure 2.14: OpenWhisk Architecture

The above figure describes all the major components in the OpenWhisk architecture. It has nginx, a controller, Kafka, CouchDB and invokers. The nginx server is a proxy that forwards requests to the controller. Controller is nothing but an API gateway that routes requests to the actions required by the user. Authentication is performed by CouchDB. To complete the flow, once the request is received by the nginx, it forwards the request to the controller, which authenticates it with CouchDB and then if authenticated it is forwarded to the invoker. The invokers use Apache Kafka to communicate with each other. As soon as a message is picked up by the invoker from the controller through Kafka, it spawns a Docker container with the function code injected into it. Not mentioned in the architecture above is Consul, which is a distributed key-value store, similar to etcd in Kubernetes, that is used to maintain the state of OpenWhisk. There are three different kinds of containers that can be spawned by the invoker. These can be hot containers, which are those that are already present and running, warm containers, those which are present but paused, and cold containers, which are not created yet. This decision is made by the invoker. Unlike a lot of the other serverless platforms seen above, OpenWhisk does not rely on Kubernetes to manage the autoscaling aspects of the serverless platform. It is managed by the controller itself. There is also a wide range of languages supported by OpenWhisk and this includes Java, PHP, Python, Swift and JavaScript for the functions. They also have one of the best documentations among the four we have compared and it is easy to set up and get started.

2.7 IoT Traffic Characteristics

IoT devices have certain common characteristics for the traffic that is generated by them. These characteristics are detailed below with a brief description about each.

Small payload – The payloads generated are small in size, since IoT devices are resource constrained in terms of power, processing capabilities and they're battery powered, they always generate smaller payloads. This also ensures that they're not consuming much network traffic and a large number of devices can concurrently operate.

High quantity – In IoT environments, most often than not, the amount of traffic generated is very high. This can be sensor readings in short intervals, live meter values being sent to an edge processing device, etc at a very high frequency.

Heterogeneity – In a high majority of IoT environments, there are many different kinds of devices that produce different kinds of data. Take the example of a smart home with multiple IoT devices continuously generating data. The data being generated will differ for each device.

Redundant data – The data sent within a payload is mostly redundant information. This can be a certain sensor information that is continuously being relayed. Since a lot of IoT applications are surrounded around monitoring and actuating, there is a lot of redundant information being generated.

Chapter 3

Problem Statement

We saw how the Internet of Things ties in with serverless computing and how there is a need for a scalable, serverless solution. In the IoT setting, a lot of the functions that are executed occur based on well defined triggers or events. For example, this can be a smart light that turns on when the ambient temperature goes down or a river level monitoring system that alerts the necessary team when the levels are dangerously high. Whatever the setting may be, we can see that there is a clear requirement of a serverless solution that scales well.

3.1 Problem Description

This leads us to the description of the problem. We know that there is a need for a scalable serverless solution for the IoT environments. But why go for a new framework when there are already solutions like Amazon Lambda, Microsoft Azure Functions and Google Cloud Functions. They are serverless computing platforms that are offered by the top three cloud service providers. There are several reasons as to why we shouldn't opt for a public cloud service provider for this case. They are detailed below.

1. There may be several organizations that deal with critical data or data that comes under regulatory bodies. For these organizations, there might be a need to process all the data within their systems and not send them outside to third-party providers. Some organizations might have a lot of infrastructure lying around and they wouldn't want to pay a third-party. For both these cases, we cannot go for a public cloud service provider.

2. Cloud service providers don't have much constraints on the language to be used or the framework which executes the code. But they do have certain formats in which the functions need to be written. This can lead to the problem of vendor lock-in [35] down the line.

3.2 Proposal

The proposed solution looks at the possibility of using Kubernetes to implement a serverless solution through the use of a lightweight client to communicate with the cluster and Docker containers to execute the serverless functions. As the framework is targeted at IoT environments the communication protocol used is MQTT to cater to lightweight payloads. MQTT is a widely deployed communication protocol in wireless sensor networks, machine to machine systems as well as IoT environments. It is lightweight, thanks to the lean header size and can easily integrate with multiple languages and frameworks. The question is to see if there is the possibility of creating such a framework and if so, evaluate whether it can scale up to loads that are similar to the traffic in IoT networks.

The current landscape of serverless frameworks include three top platforms by the public Cloud Service Providers (CSP), Amazon, Microsoft and Google. There are also other open-source serverless solutions as seen in the previous chapters that offer capabilities without having to pay for the services but they require some level of development expertise and are not really tested against IoT environments or not suited for it at all.

3.3 Challenges & Summary

There were quite a few challenging situations encountered during the course of this dissertation. The pandemic didn't help things either. They are highlighted below.

Max pods in Kubernetes – The Kubernetes documentation states that only 110 pods can be supported by a single node at the same time. But for an IoT environment, there may be a requirement for a lot more than this. It was a challenge to figure out if there was a possibility to tweak this, and after mailing some of the developers, posting in forums and going through a lot of content on the Internet, it was finally implemented.

Traffic generation and scaling up – It is never easy to demo how a framework or tool can scale up if the right kind of hardware is not available. For the scope of this prototype all the testing and evaluation was done on a local machine.

Docker images – For this dissertation, the ask was to create functions that require less space and are stateless in nature. There were two options in Alpine Linux and BusyBox to be used as base images to build the functions but the choice was made to use Alpine Linux since it supported installation of additional libraries, whereas BusyBox was more like a bundled image and didn't allow installation of additional dependencies beyond the ones that were already present.

Learning Curve – There was a steep learning curve since the project involved a myriad of technologies, platforms and tools to be used. The technologies included C/C++ for programming in ESP32, Python to program the MQTT publisher and subscriber, YAML for writing configuration files for the Kubernetes cluster, UNIX commands to access the command line for Docker and Kubernetes, etc.

Chapter 4

Design & Technical Implementation

This chapter will delve deeper into the technical aspects of the project including the design choices that were made and the overall system architecture. The overall system architecture will then be split up into several sub-components that make up the system, each going in depth into their individual implementation details.

4.1 System Architecture

This section will show in brief the different components that make up the system and also the tools used.

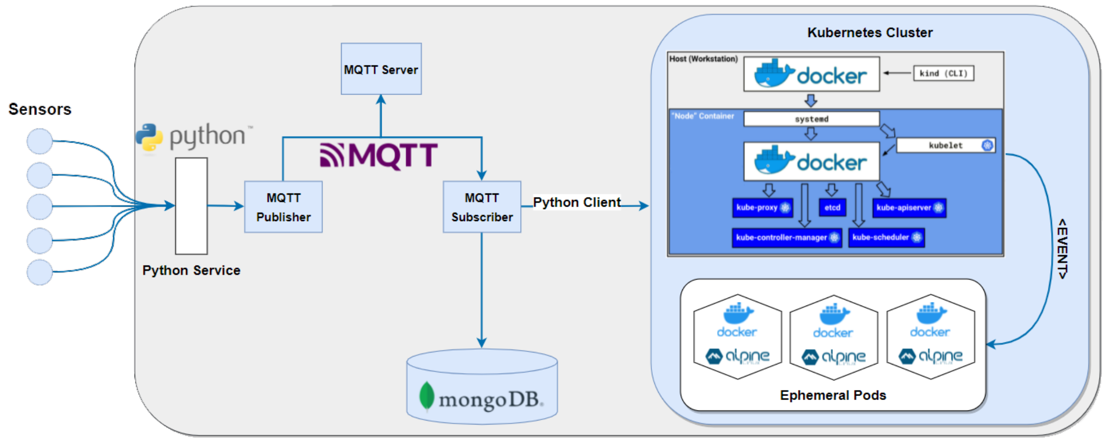
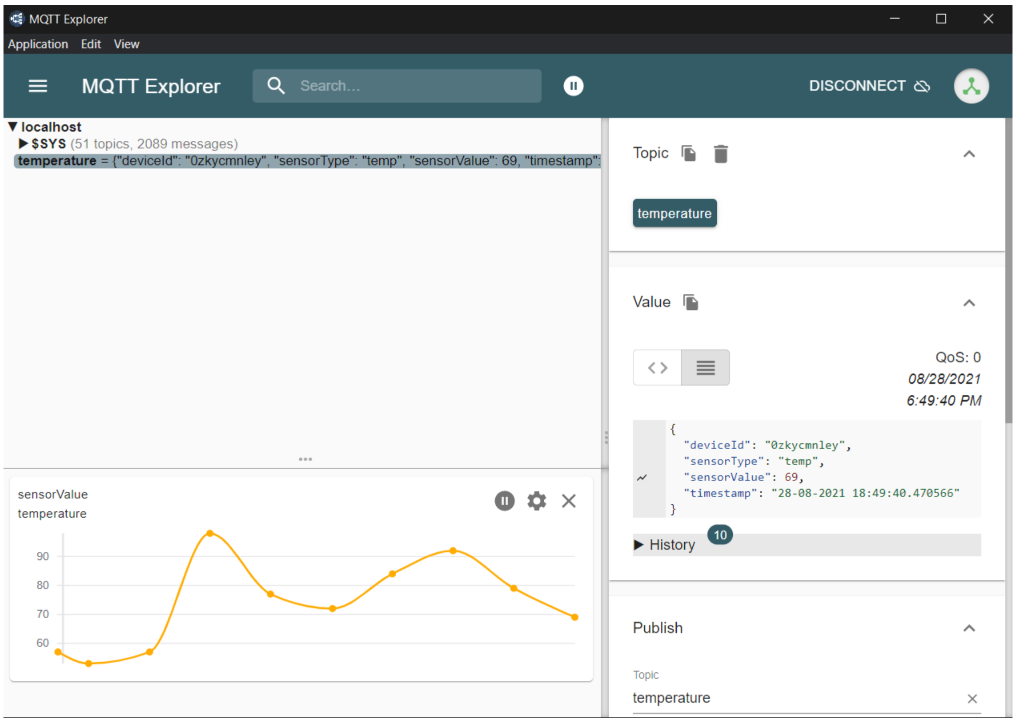


Figure 4.1: System Architecture

4.1.1 MQTT – Communication between sensor and client

The MQTT protocol was chosen because it is lightweight with a header size of just 2 bytes and is widely deployed in IoT, M2M and wireless sensor network systems. Several other protocols also exist, but since the IoT environment requires a considerably lightweight protocol, the decision was made to go with MQTT. Another important thing to note is that MQTT is well integrated with the public cloud service providers, therefore, it will allow for easy integration with them as well for future improvements.



```
$SYS (51 topics, 2089 messages)
temperature = {"deviceId": "0zkycmnley", "sensorType": "temp", "sensorValue": 69, "timestamp":
```

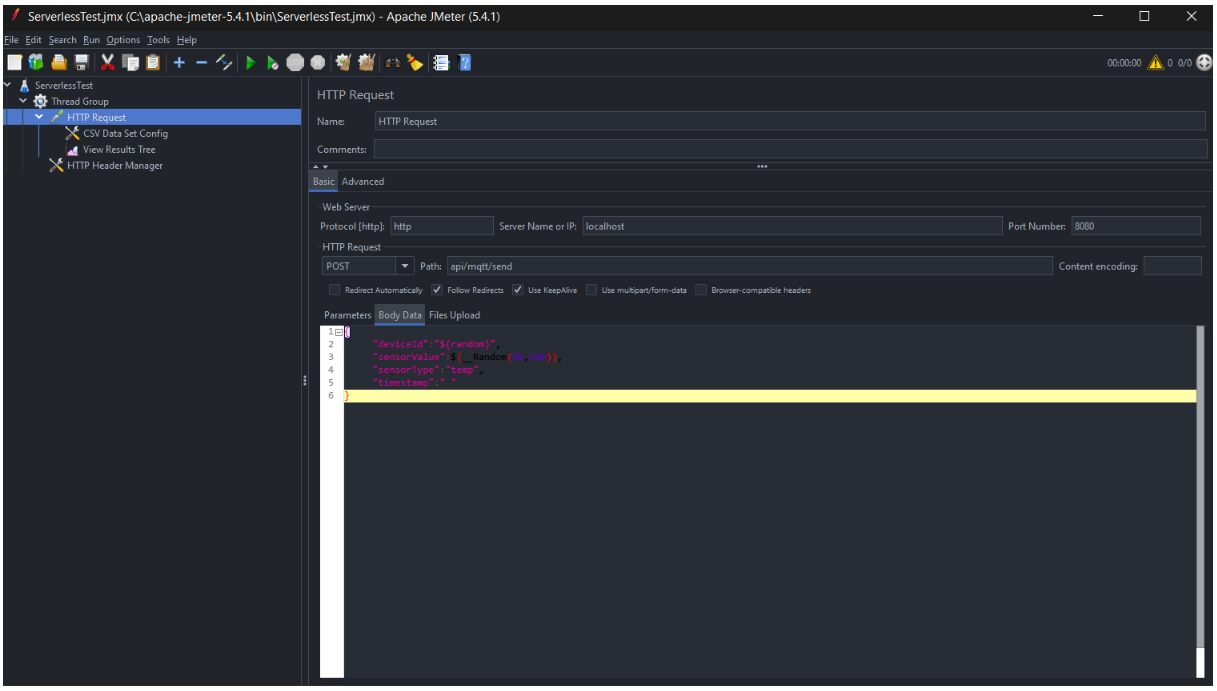
QoS: 0
08/28/2021
6:49:40 PM

```
{
  "deviceId": "0zkycmnley",
  "sensorType": "temp",
  "sensorValue": 69,
  "timestamp": "28-08-2021 18:49:40.470566"
}
```

Figure 4.2: MQTT Explorer

The local MQTT server was set up using Mosquitto. Mosquitto is a lightweight implementation of the MQTT protocol. It also has a bundled C and C++ library to interact with the server and issue commands. The server runs on the default port of 1883 and this is used by the Python services to communicate with the server. To visualize the messages being received at the topic, we used an open-source tool known as MQTT explorer. This is shown in the figure above.

Since we wanted to emulate multiple sensors sending readings simultaneously, we used Apache JMeter to load the framework with multiple readings. Apache JMeter is a testing framework that allows to generate traffic with multiple conditions. It uses multiple threads to concurrently send requests based on the configuration provided. In our tests we tried multiple scenarios using a ramp up period which determines how many requests are to be sent with the number of threads mentioned. For example, if we have mentioned the number of threads as 10 and the ramp-up period as 1 second, then 10 requests will be sent within 1 second from JMeter to the framework.


ServerlessTest.jmx (C:\apache-jmeter-5.4.1\bin\ServerlessTest.jmx) - Apache JMeter (5.4.1)
File Edit Search Run Options Tools Help
ServerlessTest
Thread Group
HTTP Request
CSV Data Set Config
View Results Tree
HTTP Header Manager
Basic Advanced
Parameters Body Data Files Upload

HTTP Request

Name: HTTP Request

Comments:

Web Server

Protocol [http]: http | Server Name or IP: localhost | Port Number: 8080

HTTP Request

POST | Path: api/mqtt/send | Content encoding:

☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

```
{
    "deviceId":"${random}",
    "sensorValue":${__Random(50,100)},
    "sensorType":"temp",
    "timestamp":" "
}
```

Figure 4.3: Test plan in Apache JMeter

4.1.2 Python Services – Publish and Subscribe

The Python service has two sub-components. One is the publisher that has an exposed endpoint to receive the readings from the sensors and send it to the MQTT server and the other one is the subscriber that receives these messages and acts based on the values received. The Paho MQTT client for Python was used to connect to the MQTT server.

Publisher – The Python publisher has two major functionalities. The first one is to connect to the local MQTT server on port 1883. The second is to expose an endpoint which the sensors can have access to receive the readings. It is important to note that this part of the functionality is only required if there is no dedicated MQTT server available. If a dedicated MQTT server is available, then the publisher itself can be removed, since the sensors will be sending their readings directly to the MQTT server and the subscriber which is listening in on the topic will receive the reading.

Subscriber – The subscriber does most of the heavy lifting in the framework. It has three major functionalities. The first function is to connect to the MQTT server and listen in for messages being received at topics. The second function is to parse these messages, read their contents and see if they need the execution of a serverless function. If it does, the subscriber connects with the Kubernetes cluster using the Kubernetes Python client and then instructs the cluster to create a pod and execute it. The last function is to save the readings that have been received into a MongoDB database.

4.1.3 Local Cluster and Kubernetes Client

The local Kubernetes cluster is created using kind as well as minikube. Both of these can be used interchangeably except for the case when we need a local cluster with multiple nodes. In that case, we have to go for kind, since minikube doesn't allow the creation of multiple nodes on the same machine. Below, we will go over the architecture of kind and minikube in a little more detail.

4.1.3.1 Kind

Kind is an abbreviation for Kubernetes in Docker and it is a tool that is used to create local Kubernetes clusters. Kind uses Docker mages to create nodes in the Kubernetes cluster. Each of the Kubernetes cluster functionalities are implemented as packages written in the Go language. Kind was initially started as a way of testing cluster functionalities before sending them to production but it has grown to be a versatile solution for smaller prototypes. Kind makes it very easy to create clusters through the use of Docker containers and it integrates well with other tooling. There is extensive documentation on how to get started with it and based on forums and activity surrounding it, Kind is very stable with great features in terms of error checking. This is largely due to the fact that it was initially set up as a way to test Kubernetes clusters. The figure below illustrates the architecture of Kind [36] and it also shows the several components that you see in a typical Kubernetes cluster.

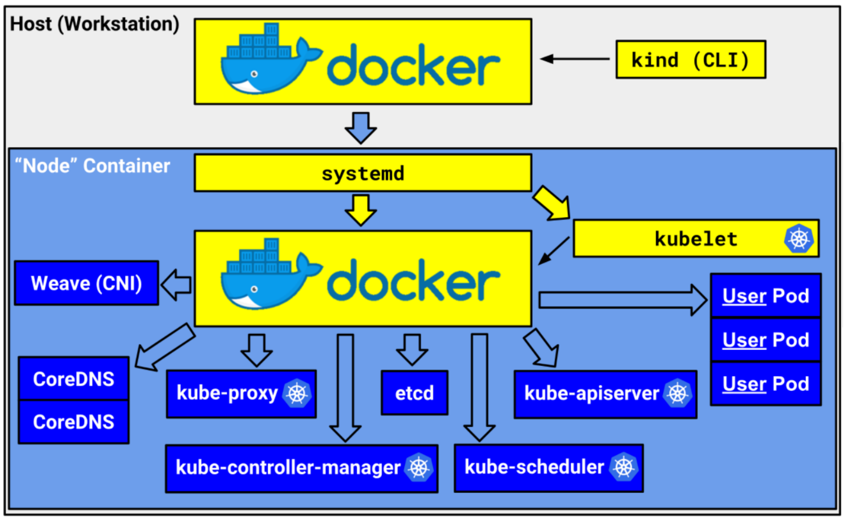


Figure 4.4: Kind Architecture

Clusters – As mentioned previously, all Kubernetes functionalities are packaged as go packages in Kind. The entire logic for clusters are found under the pkg folder, in a folder named cluster. Every cluster has a unique identification key, which is usually a docker object label key in the format of the cluster name followed by an ID. This is then set as the value of each node in a Kubernetes cluster, which in turn is a simple Docker container. The very important kube-config file is bind-mounted to a temporary directory and it is available for detection by the kind tools to be used for the containers.

Images – Running Kubernetes in a container was a problem for minikube in the initial days and they had to take help from the people at Kind who were kind enough to share their implementation style. Now, minikube also supports the same way of implementing Kubernetes through Docker containers. The base image for Kubernetes in Kind is a simple one that contains the utilities like systemd, mount, certificates, etc. Kubernetes is then actually installed over this layer.

Cluster Creation – The nodes in Kind run as Docker containers. During the initial boot, they get into a paused state. The entry point then waits for the SIGUSR1. Before starting systemd and all the other components, we can use the docker exec command to inspect and make changes to the container. After every component on the node is sufficiently ready, the entry point is instructed to actually boot the node. Once docker services are initialized on the node, kubeadm service is executed to initialize the node. After kubeadm also has initialized and booted up, we then export the kubeconfig file and then we can start testing our Kubernetes cluster.

Cluster Deletion – To delete a cluster, all we have to do is use the docker labels that identify that particular container and then delete it using the kind delete command. To get to know the nodes that are running, we need to use the kubectl command to get a list of all nodes and then delete the one that you want.

Kubernetes has client libraries in multiple languages that allow programmatic communication with the Kubernetes cluster. In this project, as we are using Python, we have used the Python client library to communicate with the cluster. On receiving messages that require the execution of a function, it is this client library the communicates with the cluster.

4.1.3.2 Configuring maxPods in Kubernetes

There was an initial challenge to create more pods on the Kubernetes cluster than what was actually mentioned on the documentation. The official documentation mentions 110 pods as the maximum number of nodes that can be supported by a single node. Since 110 sounded like a random number, we thought it could be configured and after searching up community forums and also several other documentations online, it was found that there are several organizations that run up to 500 pods in a single node in a production environment.

This is not officially supported by Kubernetes yet and it had to be done through a workaround. As part of this testing, the initial step was to try and execute more than 110 pods in my system. Just as they mentioned, after 103 pods (with the addition of the 7 system pods that Kubernetes runs), all the other pods went into a waiting state.

Even killing off the existing pods didn't start the ones in queue, but simply restarted the pods that were killed. The next step was to alter the kubelet config file that was present in each node. For a single node cluster, since there was only one file to be altered, we used the Lens IDE for Kubernetes to open a shell into the cluster itself.

The following command was executed to get the location of the kubelet config file.

```
sudo systemctl status kubelet
```

Once the location of the kubelet configuration file was identified, we simply had to change the maxPods property that dictates the maximum number of pods that can be executed. The location of the kubelet configuration file is /var/lib/kubelet/config.yaml. To test if the change was successful, we tried creating more than 110 pods, and this time it worked and we were able to successfully run 145 pods before the system shut down due to lack of resources. Just before the point of shutting down, both the CPU and memory were at 90+ in terms of utilization.

4.1.4 MongoDB – For storing unstructured data

MongoDB is used in this project as a data store for storing all the sensor readings. We have used a uniform payload across different sensor types and this also comes in handy when executing the serverless functions itself.

Unlike traditional relational databases, MongoDB is a NoSQL database, which means that it does not conform to a Structured Query Language, which is traditionally used for relational databases. It follows eventual consistency and has a rich query language that helps in fast retrieval and processing of queries. The biggest advantage with MongoDB is that it auto scales and also has high availability and performance.

As mentioned above that it is a NoSQL database, we don't need to mention a structure for the database. All the data is stored as documents which are basically in the form of JavaScript Object Notation (JSON) object. MongoDB has powerful replication features that allows the data to be replicated among a primary and multiple secondary nodes. In case of a failure of the primary, there will be a leader election that will decide the next primary. This will ensure that there is minimal to no downtime and no loss in data.

The reason to its high performance is attributed to indexing which allows a document to be indexed by a primary index and other secondary indices. This leads to faster query performance and in turn better performance.

4.1.5 Docker containers - Containing the serverless functions

The serverless functions as part of this project have been housed inside Docker containers that run Alpine Linux. The decision to go for Alpine Linux was mainly due to its small size coming in around 5.34MB. Another alternative is to use Busybox which is even smaller at under 1.4MB.

4.1.5.1 Busybox

Busybox existed way before the popularity of containers skyrocketed and it was intended to have everything that an embedded system requires in a very small footprint. It can be deployed on Linux operating systems as well as any POSIX-based operating systems as its foundation. Busybox is not really an operating system as it doesn't have a kernel on its own. It is a part of a lot of embedded systems and is open-source. Busybox bundles together several of the functionalities in the Linux system and called applets. Even though it doesn't have the exact functionality of a standard Linux distro, it comes in at a size under 2 megabytes. When it comes to deploying them in an embedded system, Busybox come as a single file with symlinks pointing to the executable for all the applications it replaces [37].

4.1.5.2 Alpine Linux

Alpine Linux, on the other hand, is based off of Busybox but has its own kernel that allows it to boot up and initiate a session. It has one of the fastest boot times of any operating system and follows the Simple, Small, Secure (SSS) principles. Alpine Linux uses the musl libc library as its standard libc implementation which is lightweight and minimal. It comes with the Busybox coreutils and its very own package manager called apk. It is secure due to the fact that all user binaries are compiled as Position Independent Executables (PIE) with stack smashing protection [38]. Alpine itself has its share of issues as well. Since it uses the musl libc instead of the standard Linux glibc which a lot of the popular distros use, anything compiled on Alpine Linux won't work on any of these popular distros. Busybox has a hardcoded syslog limit of 256 characters, which has been increased to 1024 in Alpine Linux, but even this is not very high. Messages get truncated after the 1024 character limit on Alpine Linux and it can be hard for developers to view logs of apps.

4.1.5.3 Creating the serverless functions

```
FROM alpine:latest
RUN apk add
RUN apk add ssmtp
ADD ssmtp.conf /etc/ssmtp
ADD sendMail.sh /sendMail.sh
ENTRYPOINT ["/sendMail.sh"]
CMD ["sendMail.sh"]
```

Listing 4.1: Dockerfile used for the serverless function

The snippet shown above is the Dockerfile used for the serverless function to send an email. It uses the Alpine Linux distribution as the base image. On top of this, we have added the ssmtp library to send mails. Once that dependency is installed, we need to create a configuration file that contains the credentials from where the mail will be sent. In case of an actual application, these will be the details of the organization which uses the sensors/meters to alert the consumer, but in this case it is my personal email address and its credentials.

Once we have all these in place, the next step is to add a script, sendMail.sh to the Docker image. This is the actual function that gets executed and it is a one-line function that sends a mail using the ssmtp library passing in the configuration file we created earlier with an alert message. Lastly, we set that as the entry point to the image and it gets executed as soon as the container is deployed and ready.

The reason why Busybox was not used as the base image, even though it was smaller in size was because of the fact that Busybox allowed sending a mail, but the actual process of sending the mail was very slow in comparison to the ssmtp library in Alpine Linux. Also, Busybox does not allow installation of additional libraries, therefore it didn't allow me to install ssmtp on it.

The added overhead of using a larger base image and an additional library would impact a situation where the Docker image is pulled from a remote repository. This would mean that a larger image will need more time to be downloaded to the framework and then be executed. To get over this situation a private Docker registry was used which was linked to the Kubernetes cluster. This allowed the cluster to communicate with the private Docker registry and then get the image from there rather than downloading it.

Chapter 5

Results & Evaluation

This chapter will evaluate the performance of the serverless computing framework that has been designed based on the previous chapters. The initial section will go over the setup that is used to test the system while the subsequent section details the different scenarios that the framework was tested against. The scenarios below show all cases against which the framework was tested.

1-node cluster	**3-node cluster**	**5-node cluster**
1 pod	**1 pod**	**1 pod**
10 pods	**10 pods**	**10 pods**
50 pods	**50 pods**	**50 pods**
100 pods	**100 pods**	**100 pods**

Table 5.1: Evaluation Scenarios

Furthermore, we discuss how the framework fares against some of the common open-source serverless computing frameworks and evaluate them using the same scenarios as the former.

5.1 Setup

All the tests were run on a local system with the following system configuration:
CPU: Intel i5-7300HQ Quad-Core
RAM: 16GB DDR4 2400MHz
GPU: Nvidia GTX 1050 2GB GDDR5
HDD: 1TB 7200RPM
SSD: 512GB (Running the OS and applications)

The system has minikube and kind installed for the creation of the local Kubernetes cluster. It has a dependency to install Docker Desktop to run the nodes as Docker containers. MQTT Explorer is an open-source tool that is used to visualize the messages being sent to the respective MQTT topics. The MQTT server used is Mosquitto and it is run on the same host as well. Lens, which is an IDE for Kubernetes to get in-depth information about the cluster and nodes, is also used. The programs for exposing the endpoint to receive sensor readings and also connect the MQTT publisher and subscriber are written in Python using the PyCharm IDE. Lastly, Apache JMeter is used for generating multiple readings and hitting the framework in an automated fashion.

5.2 Single Node Cluster

The first scenario is done by testing the framework with a single node with the creation of multiple pods. This scenario will serve as the baseline for the rest of our evaluation. The test plan was configured to generate 1 request that crosses the threshold to execute the function invocation every second for the different scenarios. The results obtained for that are shown in the table below.

Number of Pods	**Average Creation + Execution Time**
1 pod	**1.721109152 seconds**
10 pods	**1.680930662 seconds**
50 pods	**1.868855166 seconds**
100 pods	**1.865673363 seconds**

Table 5.2: Pod creation and execution times - Single Node Cluster

It is evident from the results that there is no strict correlation between the time taken for creation and execution with the number of pods to be created. The average deletion time that was taken for the nodes were 17.14656 milliseconds. This baseline case will now be used to evaluate the framework with multiple nodes.

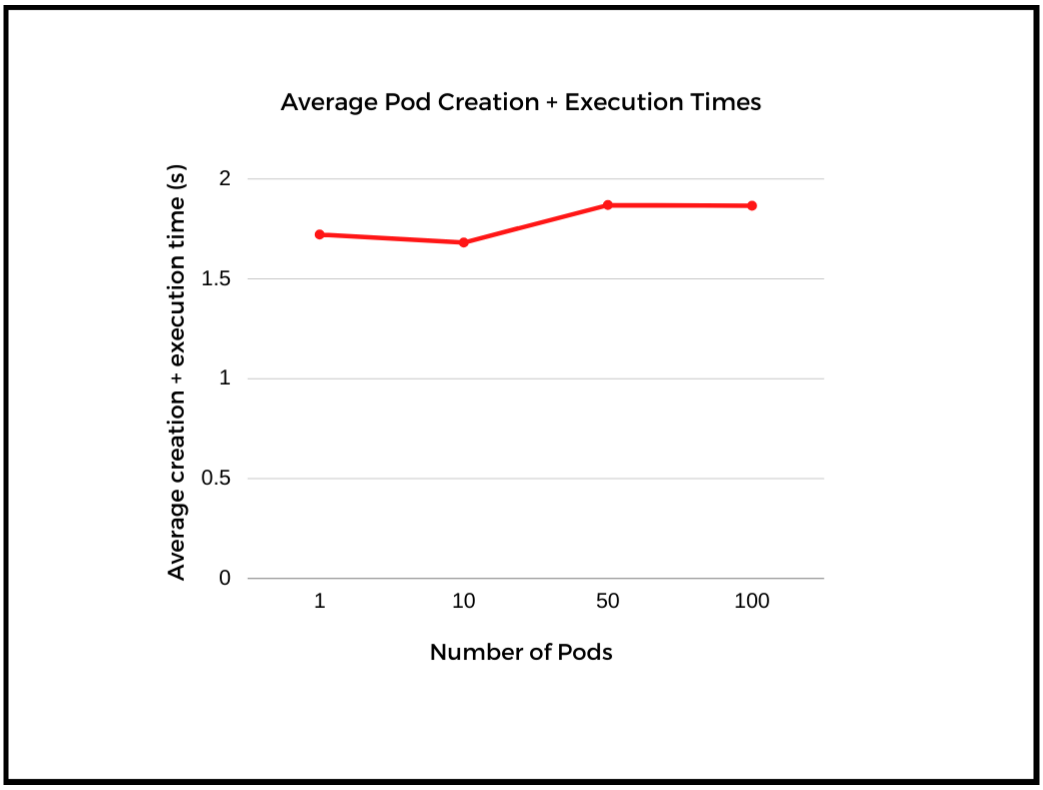


Figure 5.1: Pod Creation and Execution Times on a Single Node Cluster

5.3 Three Node Cluster

The snippet below is the configuration file that is used to create multiple nodes in kind. This is to be used as an argument when creating the cluster.

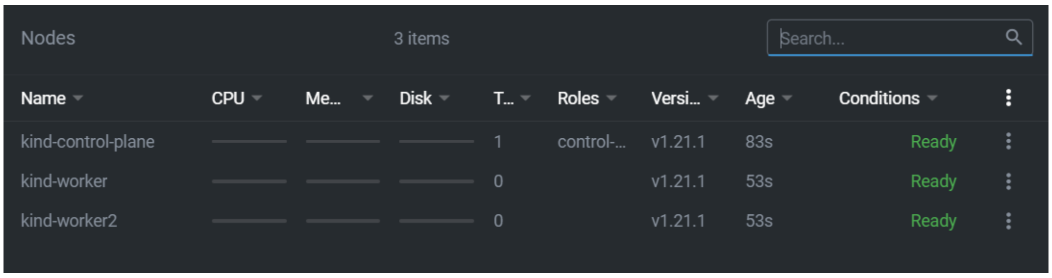
```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
```

Listing 5.1: kind-config file for creating 3 nodes

The command used to create the cluster is given below:

`kind create cluster {config kind-config.yaml`

This will create a local cluster with two worker nodes and one master node. When there is increased load, we will now have multiple worker nodes to perform the operation. In theory, it should have better results compared to a single node cluster. The figure below shows the view from the Lens IDE that shows the information of the three nodes in the cluster.


Nodes 3 items Search...

Name	CPU	Me...	Disk	T...	Roles	Versi...	Age	Conditions
kind-control-plane				1	control-...	v1.21.1	83s	Ready
kind-worker				0		v1.21.1	53s	Ready
kind-worker2				0		v1.21.1	53s	Ready

Figure 5.2: 3-Node Cluster View - Lens IDE

There was a considerable drop in the creation and execution times when the number of nodes were increased from 1 to 3. The average deletion time was almost the same with a 2-millisecond difference at 19.42 milliseconds. There was an average of 0.6425 milliseconds advantage when using 3 nodes against the single node approach.

Number of Pods	**Average Creation + Execution Time**
1 pod	**0.746335029 seconds**
10 pods	**1.192943835 seconds**
50 pods	**1.309501414 seconds**
100 pods	**1.322927063 seconds**

Table 5.3: Pod creation and execution times - Three Node Cluster

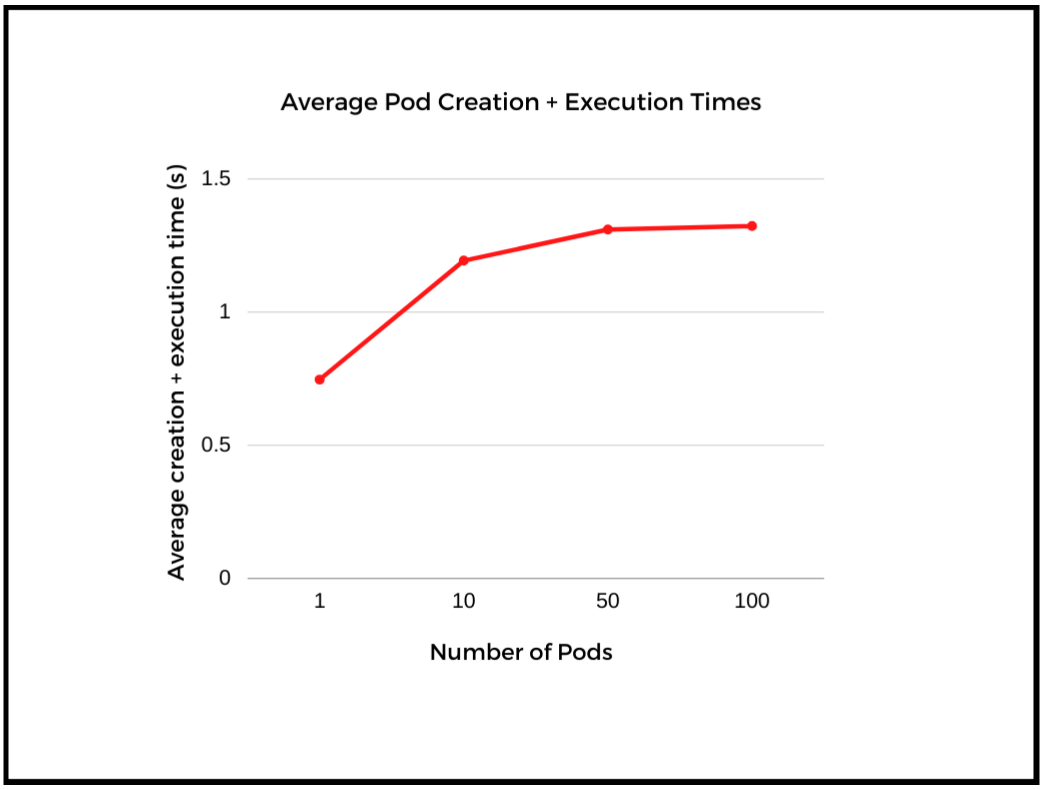


Figure 5.3: Pod Creation and Execution Times on a Three Node Cluster

Now, we will try the same setup but instead of a three-node cluster, we will go ahead with a five-node cluster.

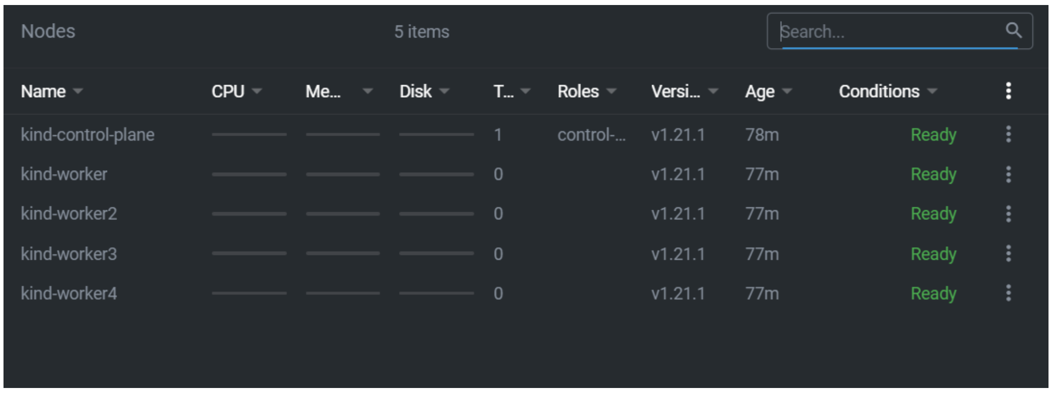
5.4 Five Node Cluster

Similar to the kind-config file that we saw earlier, this one too uses a configuration file with 4 worker nodes and 1 control plane.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
```

Listing 5.2: kind-config file for creating 5 nodes

Once the cluster is up and running, we can visualize whether the cluster is running properly without any issues using the Lens IDE.


Nodes 5 items

Name	CPU	Me...	Disk	T...	Roles	Versi...	Age	Conditions
kind-control-plane				1	control-...	v1.21.1	78m	Ready
kind-worker				0		v1.21.1	77m	Ready
kind-worker2				0		v1.21.1	77m	Ready
kind-worker3				0		v1.21.1	77m	Ready
kind-worker4				0		v1.21.1	77m	Ready

Figure 5.4: 5-Node Cluster View - Lens IDE

With that said, let's execute the tests and see how the results add up for this cluster. If everything goes as per theory, it should have an even better drop in the creation and execution times since there are 4 worker nodes that can handle the incoming requests.

Number of Pods	**Average Creation + Execution Time**
1 pod	**0.953655481 seconds**
10 pods	**0.924593759 seconds**
50 pods	**1.293314652 seconds**
100 pods	**1.489927716 seconds**

Table 5.4: Pod creation and execution times - Five Node Cluster

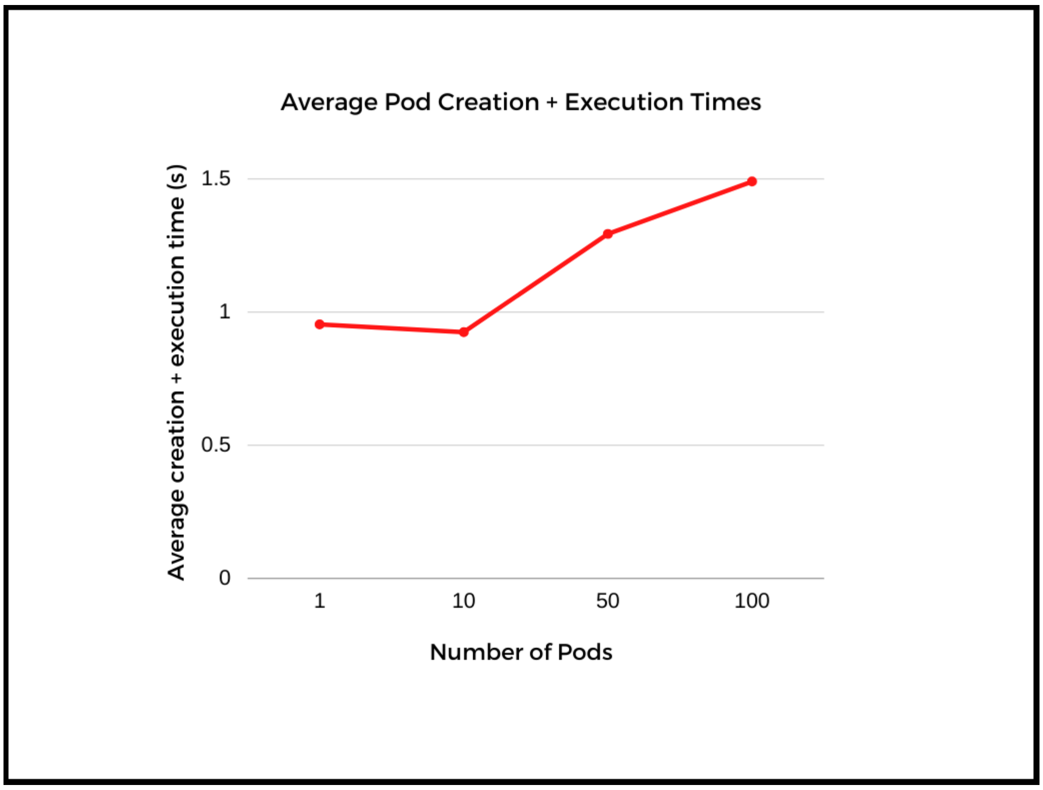


Figure 5.5: Pod Creation and Execution Times on a Five Node Cluster

Even though we see a considerable improvement in case of a smaller number of pods, in the case of 100 pods it is more than the time within a 3-node setup. It is also important to note that other factors like CPU heating, memory consumption and the overall system conditions might have also affected this setup because it was highly resource intensive. The average deletion time didn't see much of a difference and stayed at 19.88 milliseconds.

5.5 Discussion

To put it into perspective, let's take a combined look at the three setups and see how they perform. There is an evident difference between using a single node cluster against a multi-node setup. Though the difference between a 3-node cluster and a 5-node cluster was not extremely noticeable, it has to be taken into consideration that the setup was all performed on the local system which is a laptop. The resource utilization was off the charts when the 3-node and 5-node clusters were running on the laptop, and there is a high possibility that in case of a more powerful machine, the results would've been different.

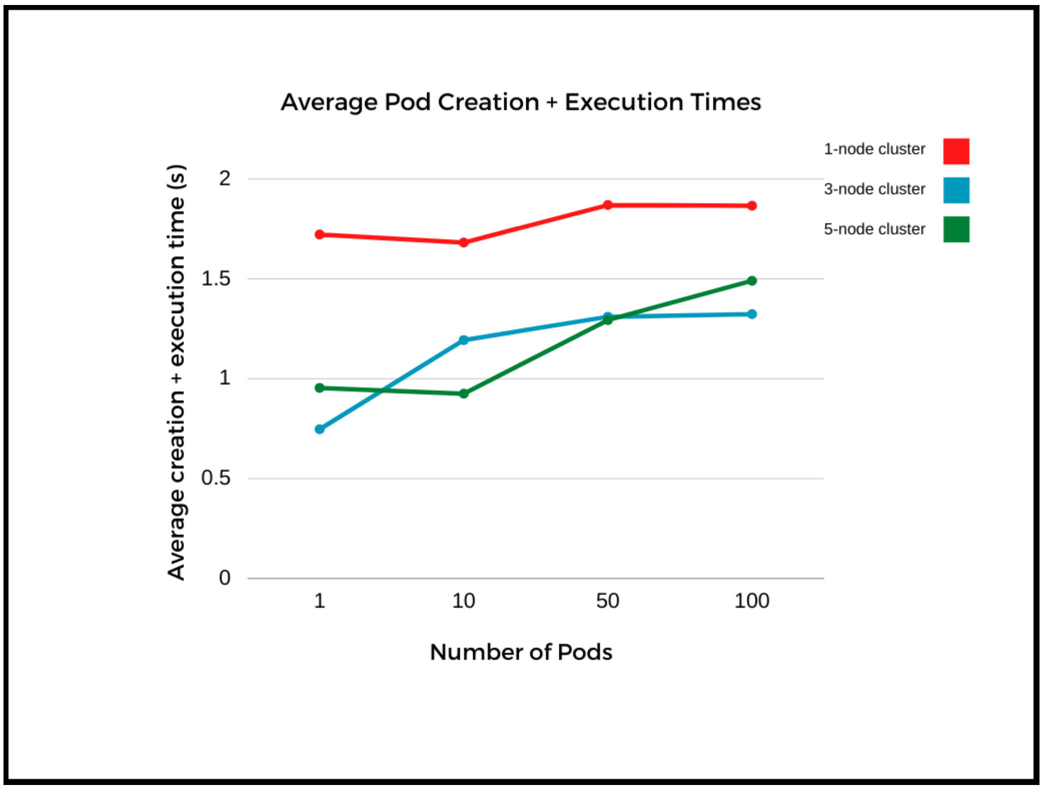


Figure 5.6: Pod Creation and Execution Times - Comparison

Chapter 6

Conclusion and Future Work

This chapter concludes the work presented in this dissertation and is spread across two sections. The first section details the conclusions that were drawn from the dissertation and the second section draws the possible future enhancements and work that could be carried out on this project.

6.1 Conclusion

To conclude this dissertation, it has undoubtedly been fruitful. The initial aim was to look at the possibility of creating a serverless computing framework with barebones Kubernetes and Docker containers to execute the functions. Once it was realized, the research progressed further to implement it for IoT networks. Through this research, the following contributions and benefits have been realized.

- Understand the serverless computing landscape and how the frameworks need to adapt for IoT environments. Use that knowledge to understand how to create lightweight functions with the help of Alpine Linux and Docker containers.
- Extend the capabilities of Kubernetes nodes by tweaking the kubelet configuration file and hence make it possible to be used in environments where a large number of pod executions are to be made. It is worth mentioning that, when executing high number of pods, the hardware should be powerful enough to support it.
- A considerable improvement in the time to create and execute functions were observed by increasing the number of nodes in a cluster. A 3-node cluster and a 5-node cluster performed considerably better than a single node cluster. It should be made mandatory that when using serverless computing frameworks based on Ku-

bernetes, there should be a minimum of 3 nodes to also support for fault tolerance and resiliency.

- Most of the current open-source serverless computing frameworks use Apache Kafka for communications between the events and the serverless functions. Kafka follows a pub-sub messaging pattern. This can be effectively replaced by an MQTT server which also uses the same messaging pattern but is much more lightweight and more suited to IoT environments.

6.2 Future Work

One of the main points of future work is to have a distributed cluster of Kubernetes nodes running the framework and then testing it based on traffic from different sources. It will be interesting to note how the framework performs and how much of a hit the latency takes with the network distribution involved. Since most of the tests were done on a local system with a quad-core processor and 16GB of RAM, it would be interesting to also note how a more powerful setup like server grade hardware will be when it comes to executing these serverless functions.

A dashboard that gives the user fine grained details like the sensors sending readings and their statuses using the MongoDB readings that we are storing is also a future enhancement that could really give it a production grade feature. The dashboard can give the user information like the number of sensors that are active, the ones that got triggered and executed the serverless function, the ones that are not, etc.

Bibliography

[1] P. Suresh, J. V. Daniel, V. Parthasarathy, and R. H. Aswathy, "A state of the art review on the internet of things (iot) history, technology and fields of deployment," in *2014 International Conference on Science Engineering and Management Research (ICSEMR)*, p. 1–8, IEEE, Nov 2014.

[2] Y. Perwej, M. K. Omer, O. E. Sheta, H. A. M. Harb, and M. S. Adrees, "The future of internet of things (iot) and its empowering technology," *International Journal of Engineering Science*, vol. 20192, 2019.

[3] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, p. 44–54, Nov 2019.

[4] M. Domb, *Smart Home Systems Based on Internet of Things.* InTech, 2019.

[5] A.-V. Nedelcu, F. Sandu, M. Machedon-Pisu, M. Alexandru, and P. Ogrutan, "Wireless-based remote monitoring and control of intelligent buildings," in *2009 IEEE International Workshop on Robotic and Sensors Environments*, pp. 47–52, 2009.

[6] V. Puri and A. Nayyar, "Real time smart home automation based on pic microcontroller, bluetooth and android technology," in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 1478–1484, 2016.

[7] S. Zhihua, "Design of smart home system based on zigbee," in *2016 International Conference on Robots Intelligent System (ICRIS)*, pp. 167–170, 2016.

[8] S. Gunputh, A. P. Murdan, and V. Oree, "Design and implementation of a low-cost arduino-based smart home system," in *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*, pp. 1491–1495, 2017.

[9] M. Shafi, A. F. Molisch, P. J. Smith, T. Haustein, P. Zhu, P. De Silva, F. Tufvesson, A. Benjebbour, and G. Wunder, "5g: A tutorial overview of standards, trials,

challenges, deployment, and practice," *IEEE journal on selected areas in communications*, vol. 35, no. 6, pp. 1201–1221, 2017.

[10] S. C. Ergen, "Zigbee/ieee 802.15. 4 summary," *UC Berkeley, September*, vol. 10, no. 17, p. 11, 2004.

[11] D. Gorecky, M. Schmitt, M. Loskyll, and D. Zühlke, "Human-machine-interaction in the industry 4.0 era," in *2014 12th IEEE international conference on industrial informatics (INDIN)*, pp. 289–294, Ieee, 2014.

[12] P. J. Mosterman and J. Zander, "Industry 4.0 as a cyber-physical system study," *Software & Systems Modeling*, vol. 15, no. 1, pp. 17–29, 2016.

[13] T. Yokotani and Y. Sasaki, "Comparison with http and mqtt on required network resources for iot," in *2016 international conference on control, electronics, renewable energy and communications (ICCEREC)*, pp. 1–6, IEEE, 2016.

[14] H. W. van der Westhuizen and G. P. Hancke, "Comparison between coap and mqtt-server to business system level," in *2018 Wireless Advanced (WiAd)*, pp. 1–5, IEEE, 2018.

[15] S. Appel, K. Sachs, and A. Buchmann, "Towards benchmarking of amqp," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pp. 99–100, 2010.

[16] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*, pp. 1–7, IEEE, 2017.

[17] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2658–2659, IEEE, 2017.

[18] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*, pp. 1–20, Springer, 2017.

[19] "Serverless showdown - acloudguru." Available at `https://acloudguru.com/blog/engineering/serverless-showdown-aws-lambda-vs-azure-functions-vs-google-cloud-` (accessed Aug. 30, 2021).

[20] "Aws lambda pricing." Available at `https://aws.amazon.com/lambda/pricing/`, (accessed Aug. 30, 2021).

[21] "Google cloud functions pricing." Available at `https://cloud.google.com/functions/pricing`, (accessed Aug. 30, 2021).

[22] "Microsoft azure functions pricing." Available at `https://azure.microsoft.com/en-us/pricing/details/functions/`, (accessed Aug. 30, 2021).

[23] R. P. Goldberg, "Architectural principles for virtual computer systems," tech. rep., HARVARD UNIV CAMBRIDGE MA DIV OF ENGINEERING AND APPLIED PHYSICS, 1973.

[24] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 171–172, IEEE, 2015.

[25] C. Anderson, "Docker [software engineering]," *Ieee Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[26] "Docker architecture." Available at `https://docs.docker.com/get-started/overview/#docker-architecture`, (accessed Aug. 30, 2021).

[27] A. Cockcroft, "Evolution of business logic from monoliths through microservices to functions," Feb 2017. Available at `shorturl.at/hkzLW`.

[28] "Kubernetes architecture." Available at `https://kubernetes.io/docs/concepts/overview/components/`, (accessed Aug. 30, 2021).

[29] "Custom resource definitions in kubernetes." Available at `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/`, (accessed Aug. 30, 2021).

[30] "Kubeless architecture." Available at `https://kubeless.io/docs/architecture/`, (accessed Aug. 30, 2021).

[31] N. E. Ioini, D. Hästbacka, C. Pahl, and D. Taibi, "Platforms for serverless at the edge: A review," in *European Conference on Service-Oriented and Cloud Computing*, pp. 29–40, Springer, 2020.

[32] N. Kaviani, D. Kalinin, and M. Maximilien, "Towards serverless as commodity: a case of knative," in *Proceedings of the 5th International Workshop on Serverless Computing*, pp. 13–18, 2019.

[33] "Serverless by cncf." Available at `https://github.com/cncf/wg-serverless`, (accessed Aug. 30, 2021).

[34] "Apache openwhisk." Available at `https://openwhisk.apache.org/`, (accessed Aug. 30, 2021).

[35] T. C. i. S. Francisco, "Lambda and serverless is one of the worst forms of proprietary lock-in we've ever seen in the history of humanity." Available at `https://www.theregister.com/2017/11/06/coreos_kubernetes_v_world/`, (accessed Aug. 30, 2021).

[36] "Kind design." Available at `https://kind.sigs.k8s.io/docs/design/initial/`, (accessed Aug. 30, 2021).

[37] G. Sally, "Busybox," in *Pro Linux Embedded Systems*, pp. 293–307, Springer, 2010.

[38] "Alpine linux." Available at `https://alpinelinux.org/about/`, (accessed Aug. 30, 2021).

Appendix

Abbreviation	Expansion
IoT	Internet of Things
CSP	Cloud Service Provider
M2M	Machine to Machine
FaaS	Function-as-a-Service
GSM	Global System for Mobile Communication
TDMA	Time Division Multiple Access
LTE	Long Term Evolution
RAN	Radio Access Networks
PLC	Programmable Logic Controller
ERP	Enterprise Resource Planning
CSP	Cyber Physical Systems
MQTT	Message Queue Telemetry Transport
CoAP	Constrained Application Protocol
AMQP	Advanced Message Queueing Protocol
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
PaaS	Platform-as-a-Service
REST	Representational State Transfer
GUI	Graphical User Interface
CNCF	Cloud Native Computing Foundation
JSON	JavaScript Object Notation
CRD	Custom Resource Definition

Table 1: List of Abbreviations