Aalto University
School of Science
Master's Programme in Security and Mobile Computing (NordSecMob)

Manoj Kumar

Serverless computing for the Internet of Things

Master's Thesis
Espoo, Aug 10, 2018

Supervisors:	Mario Di Francesco, PhD, Aalto University
	Satish Narayana Srirama, PhD, University of Tartu
Advisor:	Mario Di Francesco, PhD, Aalto University

Aalto University
School of Science
Master's Programme in Security and Mobile Computing (NordSecMob)

ABSTRACT OF MASTER'S THESIS

Author:	Manoj Kumar		
Title: Serverless computing for the Internet of Things			
Date:	Aug 10, 2018	**Pages:**	64
Major:	Security and Mobile Computing	**Code:**	T3011
Supervisors:	Mario Di Francesco, PhD Satish Narayana Srirama, PhD		
Advisor:	Mario Di Francesco, PhD		

Cloud-based services have evolved significantly over the years. Cloud computing models such as IaaS, PaaS and SaaS are serving as an alternative to traditional in-house infrastructure-based approach. Furthermore, serverless computing is a cloud computing model for ephemeral, stateless and event-driven applications that scale up and down instantly. In contrast to the infinite resources of cloud computing, the Internet of Things is the network of resource-constrained, heterogeneous and intelligent devices that generate a significant amount of data. Due to the resource-constrained nature of IoT devices, cloud resources are used to process data generated by IoT devices. However, data processing in the cloud also has few limitations such as latency and privacy concerns. These limitations arise a requirement of local processing of data generated by IoT devices. A serverless platform can be deployed on a cluster of IoT devices using software containers to enable local processing of the sensor data. This work proposes a hybrid multi-layered architecture that not only establishes the possibility of local processing of sensor data but also considers the issues such as heterogeneity, resource constraint nature of IoT devices. We use software containers, and multi-layered architecture to provide the high availability and fault tolerance in our proposed solution.

Keywords:	IoT, Serverless computing, Fog computing, Docker
Language:	English

Acknowledgements

I wish to express my gratitude to my supervisors Mario Di Francesco and Satish Narayana Srirama for their valuable inputs, feedbacks and engagement throughout this master thesis.

Thank you!

Espoo, Aug 10, 2018

Manoj Kumar

Abbreviations and Acronyms

API	Application Programming Interface
ARM	Advanced RISC Machine
CLI	Command-line Interface
CISC	Complex Instruction Set Computing
DBaaS	Database as a Service
DNSRR	DNS Round Robin
DNS	Domain Name System
IoT	Internet of Things
IBM	International Business Machines Corporation
ISA	Instruction Set Architecture
JSON	JavaScript Object Notations
REST	Representational State Transfer
RISC	Reduced Instruction Set Computing
SDK	Software Development Kit
STDIN	Standard Input
STDOUT	Standard Output
STDERR	Standard Error
TLS	Transport Layer Security
WWW	World Wide Web

Contents

Chapter 1

Introduction

The history of the Internet begins with the development of ARPANET in the 1960s by the US Department of defence [1]. A standardized protocol suite for the Internet was introduced in the 1970s in the form of TCP/IP which was followed by the world wide web (WWW) in 1980s [2]. By the end of 1990s, it became accessible to the common population. With over 7.5 billion user's across the globe and a growth rate of over 1000 percent from 2000 to 2018, the Internet is now in use more than ever [3]. The Internet has revolutionised the involvement of technology in day to day life. From smart-phones to smart-watches, weather to agriculture, smart cars to smart homes, the Internet is visibly influencing our way of life.

In addition to ordinary users, Internet accessibility has also significantly changed information technology industry and services. Cloud computing is one of such services which is gaining wide popularity. Cloud computing is a model where computing services such as storage, compute servers and database are provided to end users over the Internet by cloud providers. Users are charged per usage for availing these services. This billing model is similar to utilities such as electricity, gas, and water. Cloud computing not only significantly reduced the potential investment for the users but has also reduced the time to access infrastructure and services [4]. As a consequences, businesses of different sizes are moving from traditional in-house data centres to cloud-based services. Netflix is a prime example of the migration of an overgrowing business to the cloud. They moved all of their in-house services to Amazon Web Services in January 2016 to overcome the problem of vertical scaling of their vast databases at their static data centres [5]. Indeed, features such as auto-scaling, redundancy, low initial investment and operational cost metered service make cloud-based models a leading option for organisations with large and increasing data volumes.

Similar to cloud-based services, Internet of Things (IoT) and related applications have gained massive popularity in recent times. IoT network comprises smart and connected objects called “things”. Things communicate with each other without any human intervention and generate a significant amount of data. Things are heterogeneous and build a dynamic infrastructure. IoT devices have already outreached the human population and are expected to cross the number of 500 billion by 2030[6]. IoT devices differ in architecture, sensing capabilities and other aspects such as memory and power. IoT network is dynamic as devices are deployed in a frequently changing environment. Furthermore, IoT devices are resources constrained as they are often deployed with limited memory, power and processing capability [7].

1.1 Problem statement

Cloud computing and IoT devices are characterized by contrasting characteristics. For instance, IoT devices work with limited capabilities whereas the cloud provides an illusion of infinite resources. Cloud computing provides the required resources to the IoT network. Due to the limitations of IoT devices, generated data is offloaded to cloud-based resources for further processing, and the cloud sends the results back upon processing the data [7]. IoT devices in conjunction with cloud resources perform efficient data processing. However, such solution has the following limitations:

- **High latency:** Offloading [8] a small task to the cloud takes relatively more time than processing it locally at the IoT device.
- **Privacy concerns:** Some tasks need more privacy, which makes it infeasible to offload their processing to the cloud.
- **Support for mobility:** In case of non-stationary sensing devices, it might not be possible to offload processing data to the cloud. In such a scenario, a sensor should be able to process it locally.

1.2 Scope and Goal

This work addresses the issues mentioned above and aims at developing a solution that employs serverless computing in the context of IoT. We expect our solution to satisfy some significant characteristics such as easy deployment, fault tolerance and high availability. We also aim at designing our solution’s architecture considering a heterogeneous IoT network. As part of

this work, we deploy an IoT compatible serverless platform using a multilayer architecture across the edge, fog and cloud computing layers.
We target the following research goals in this work.

- **High availability of the architecture:** The architecture should be highly available as there should be a secondary node available in case of the failure of the primary node.

- **Easy of deployment:** Users should be able to deploy the architecture Heterogeneous IoT devices with sufficient ease.

- **Support for heterogeneity:** Heterogeneity is an essential property of any IoT network. Our solution should allow users to deploy it across IoT devices irrespective of the differences in their hardware platforms.

- **Fault-tolerant architecture:** We aim to develop a fault-tolerant architecture to defend it against unexpected failures.

1.3 Structure of the Thesis

The rest of the thesis is organized as follows. Chapter 2 explains the basics of IoT network, system architecture and Docker container-based virtualisation along with Docker swarm. Chapter 3 introduces serverless computing and compares different open source serverless platform. Chapter 4 describes a multi-layer architecture for serverless computing in IoT and introduces a function execution offloading algorithm. Chapter 5 evaluates the proposed solution of various defined criteria such as ease of deployment, fault tolerance, device heterogeneity, scalability, function runtime heterogeneity, security and privacy, . Finally, chapter 6 concludes this work along with future research directions.

Chapter 2

Background

2.1 Internet of Things

The Internet of Things (IoT) is a network of smart, physical devices such as home appliances, vehicles, wearable devices and sensors. These devices are called *Things* and use the Internet to communicate with each other. Iot devices are smart, interconnected and resource-constrained in nature [9]. IoT devices are heavily populated and expected to reach 26 billion by 2020 [6]. Smart home, agriculture, healthcare, intelligent transportation system are some applications scenario relying on IoT devices [10]. For instance, smart cars perform vehicle to vehicle communication for traffic analysis. In another example, doctors monitor the health of patients by IoT devices. Communications mentioned above require connectivity among devices and results in the generation of a large amount of data. Figure 2.1 shows the communication between an IoT node and a gateway. IoT devices have many characteristics that make them different from traditional computing devices. We discuss these properties in the remaining part of this section.

Heterogeneity: IoT devices vary in the hardware and capabilities. As a result, an IoT network contains various IoT devices to interact with each other.

Resource constraints: IoT devices have limited processing power and memory. They are designed to carry out limited computation.

Power constraints: IoT devices can have either continuous or limited power resources. In case of continuous power, a device is continuously plugged into the power resource. Alternatively, In the mobile environment, IoT devices are non-stationary and have limited batteries as their source of power.

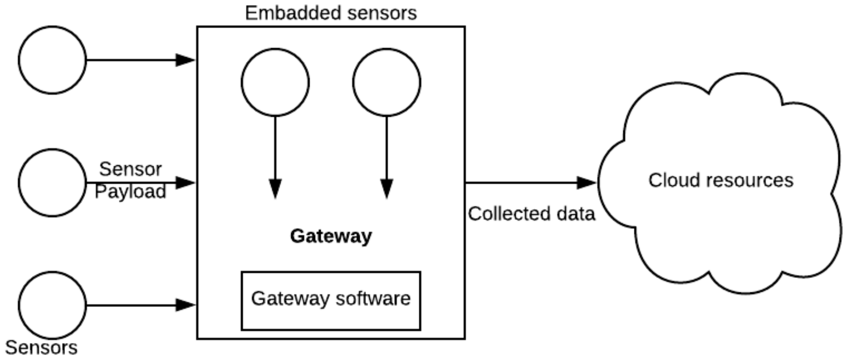

Figure 2.1: IoT network

Dynamic network: IoT devices are mobile, which makes IoT network dynamic and frequently changing.

2.2 Hardware platforms

The central processing unit (CPU) is responsible for the execution of computing tasks. It is also responsible for assigning tasks to other components. Computing devices such as laptops, desktops mobile phones, smart watches and IoT devices use a variety of processors. These processors differ concerning processing power, power consumption, memory availability, instruction sets and many other features. Rest of this section discusses some popular processors such as the ARM, x86 and x64.

2.2.1 Advanced RISC machine

ARM (Advanced RISC machine) processors belong to the family of RISC (Reduced instruction set computers). RISC-based processors are developed to output high processing speed with less number of instruction sets [11]. In contrary to CISC (complex instruction set computers), RISC machines consume less power as they remove unnecessary instructions from the instruction

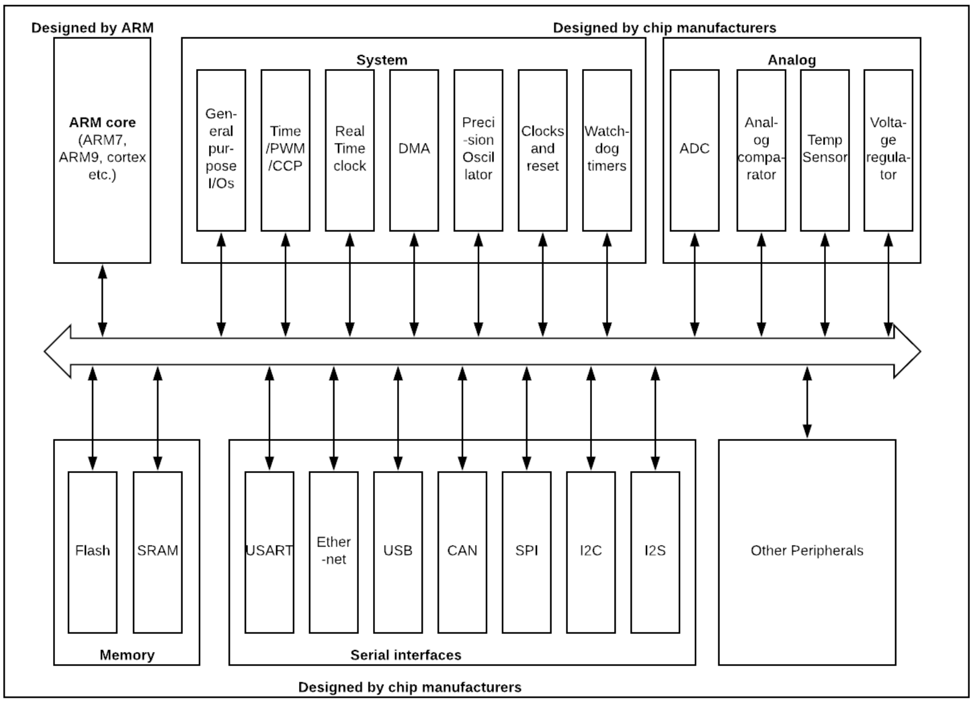

Figure 2.2: ARM simplified block diagram system on chip

set.

ARM processor before version 8 supports 32 bits architecture whereas version 8 and later supports both 32 and 64-bit architecture. ARM processors are ideal for smaller devices such as mobile phones, IoT and other smart devices as they require fewer transistors. ARM processors also support single-cycle process execution and hardware-based virtualisation.

2.2.2 x86 platform

X86 processors are based on CISC which allows them to maintain more special purpose registers instead of general purpose registers. X86 processors support 32-bit instruction set registers in addition to the backward compatibility with 16 and 8-bit registers. X86 processors were initially developed using 16-bit Intel 8086 and 8088 microprocessors later upgraded to the 32-bit, 80386 processors.

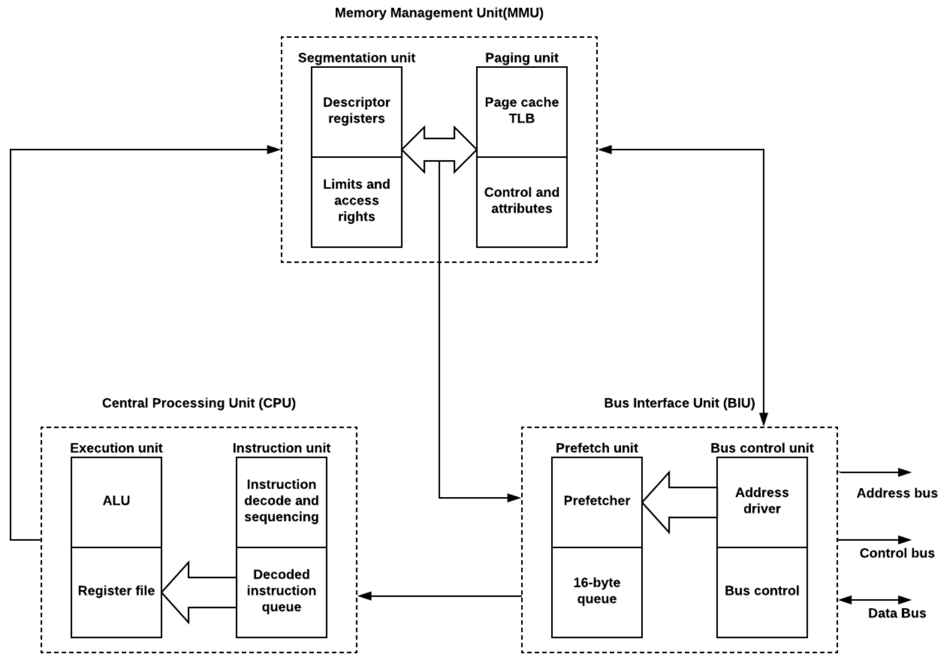

Figure 2.3: Intel 80386 internal architecture

2.2.3 x64 platform

x64 is the upgraded 64-bit version of x86 32 bit instruction set architecture (ISA). AMD initially developed a 64-bit architecture called x86_64 later renamed to AMD64. Similarly, Intel named its implementation IA-32e and later renamed it to EMT64. x64 processors offer larger memory space than 32-bit processors and allow computer programs to store more in-memory data. Table 2.2.3 demonstrate key differences between ARM and x64/86 processors.

	ARM	**x86/64**
Speed	Slower than x86/64	Fast
Power consumption	Low	High
Instruction set	RISC	CISC
Architecture support	32 and 64-bit	32 and 64-bit
Devices	Raspberry Pi, Mobile phones	Intel/AMD computers

Table 2.1: ARM vs x86/64

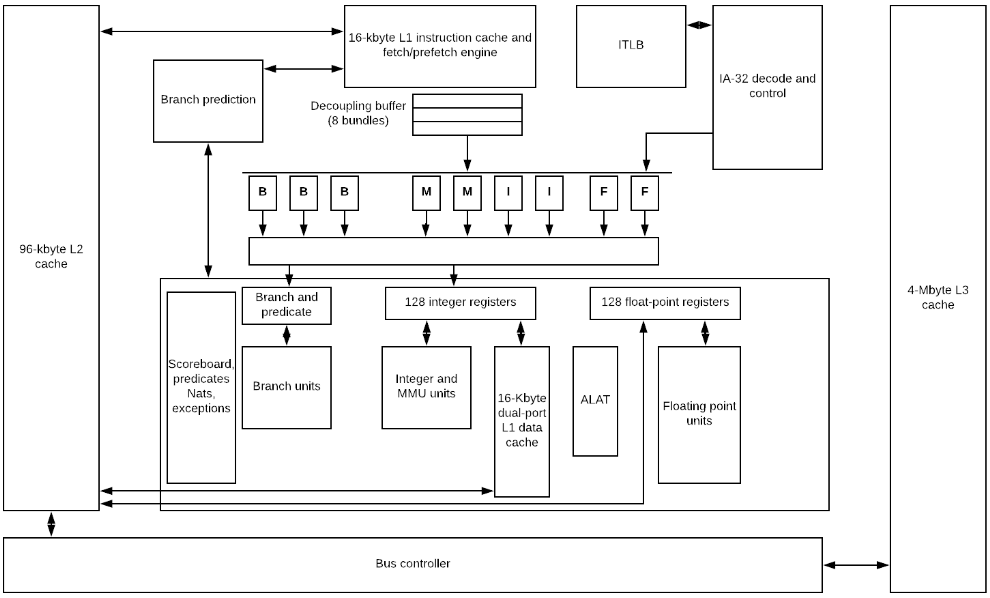

Figure 2.4: x64 architecture

2.3 Software containers and Docker

Docker is a container management platform that allows developers to create, run and deploy their applications easily using Linux containers[1].

2.3.1 Container-based virtualization

A container is a runtime instance of an image that shares the kernel of the host machine and runs natively on the operating system [12]. In contrast, a virtual machine uses a hypervisor to run a complete guest operating system on top of the host operating system. Containers are more flexible, scalable, lightweight and portable in comparison to the traditional virtual machines. One machine can run more than one containers. Linux *namespaces* ensure the isolation of containers' resources from each other. For instance, files, ports and memory allocated to one container are kept isolated from the rest of the containers. It is also possible to define a limit on the resource allocation to the container. Isolation within multiple containers provides security and efficiency. Figure 2.5 displays the architectural differences between a container

[1] https://linuxcontainers.org/lxc/introduction/

and a virtual machine. A container is initiated using an executable image that contains the necessary code, runtime environment, files, environment variables and other dependencies.

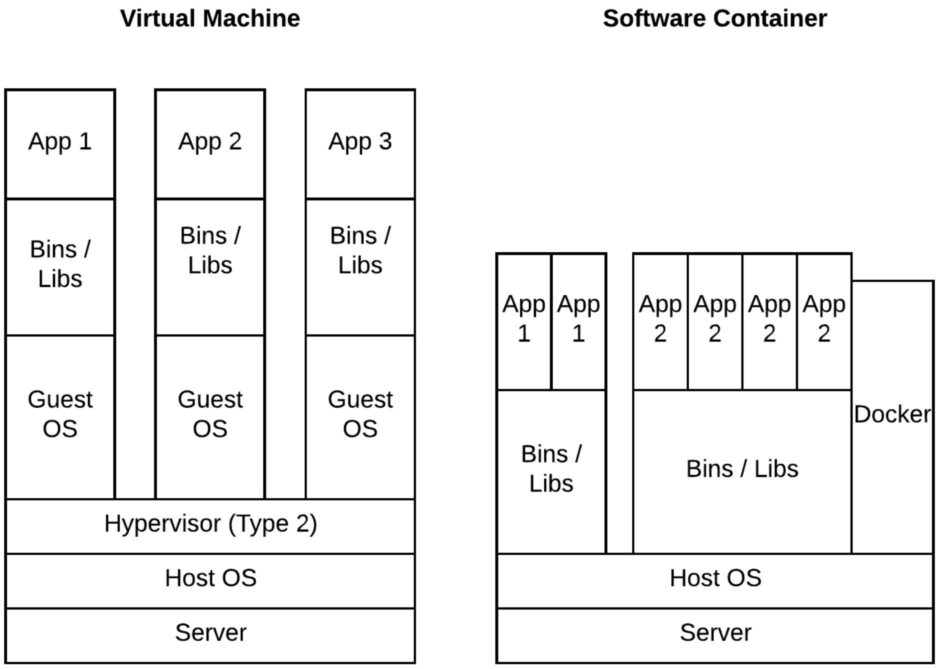

Figure 2.5: Architecture: container vs virtual machine

Software containers help developers deploying the desired execution environment swiftly at any scale as they run on the host kernel as an isolated process and can start almost instantly by using low system resources.

2.3.2 Docker containers

Docker containers are open standard Linux-based containers that can be deployed over a majority of operating systems [13]. Docker containers are initialized from images and allocated system resources. A container is instantiated using writable layers over original images. Similarly, changes to the containers are also added in the form of another layer without disturbing existing layers. Docker image files are read-only and allow other images to use them without any conflicts. Each of the Docker containers maintains its resources such as memory, processes, ports, namespaces and filesystem. They use exposed ports to communicate to external hosts. Docker Engine API provides users with CLI commands to deploy and manage Docker containers.

2.3.3 Dockerfile

Dockerfile is a text file containing a set of commands used to generate a Docker container image. Listing 2.1 shows a sample Dockerfile.

```
#Base Image
FROM ubuntu:16.04

MAINTAINER Manoj Kumar

LABEL version="1.0"

# prerquisites
RUN apt-get update && apt-get install -y apt-utils && apt-get
   install -y curl

ENV DEBIAN_FRONTEND=noninteractive

COPY application.apk /go/src/project/application.apk

WORKDIR WORKDIR /go/src/project/

#Java8 installation
ARG JAVA_URL=http://download.oracle.com/otn-pub/java/jdk/8u131-
   b11/d54c1d3a095b4ff2b6607d096fa80163/jdk-8u131-linux-x64.tar.
   gz
ARG JAVA_DIR=/usr/local/java
ARG JAVA_VER=jdk1.8.0_131

RUN mkdir -p $JAVA_DIR && cd $JAVA_DIR \
  && (curl -Lsb "oraclelicense=accept-securebackup-cookie"
     $JAVA_URL | tar zx) \
  && update-alternatives --install "/usr/bin/java" "java" "
     $JAVA_DIR/$JAVA_VER/bin/java" 1 \
  && update-alternatives --install "/usr/bin/javac" "javac" "
     $JAVA_DIR/$JAVA_VER/bin/javac" 1

EXPOSE 4723

COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

Listing 2.1: Sample Dockerfile

The commands in the dockerfile are explained below.

- **FROM** is used to set a base image for the Docker container image. It is also possible to specify through its release tag. Dockerfiles can also be initiated using a scratch image. Scratch is the minimal image that serve as starting point for building containers.
- **LABEL** is responsible for appending extra description to the image. Labels are specified in the metadata of the image.
- **RUN** executes the commands for the installation of software and scripts execution.
- **COPY** copies the file from one folder to another by overwriting the file in the second folder if already exists.
- **ADD** automatically uploads, uncompresses and places the obtained file to the specified path.
- **EXPOSE** allows inbound traffic to the container over the specifies ports. This command enables outside users to access services on container over the exposed ports.
- **ENTRYPOINT and CMD** are often used in conjunction. Use of *ENTRYPOINT alone makes the container executable. However, using CMD allows users to execute a command on the environment defined in ENTRYPOINT. The listing shows commands demonstrating aforementioned.*
- **USER** *allows to specify a username at the time of the execution of a command. Using the username with correct permissions is important.*
- **WORKDIR** *sets the working directory for the defined instructions' execution.*
- **ENV** *command is used to define environment variables while building and executing the image.*

It is recommended to have one Dockerfile per folder for the better organisation. Docker images can be made more efficient by avoiding unnecessary packages. One of the ideal practices is to run only one application per container. Docker container allows us to run more than one applications per container at the cost of simplicity. Docker images can be stored and distributed using online registry services such as Docker Hub[2], Docker store[3], quay[4], Amazon EC2 container registry. It is also possible to have a registry hosted locally.

2.3.4 Container orchestration

Orchestration is the set of techniques to automate the deployment, management and scaling of containers. Orchestration allows developers to provision and instantiates new containers. It is also responsible for maintaining the expected state of the system by initiating a new container upon discovering a failure. Finally, it maintains connectivity among containers and exposes the running services to external hosts. One of the main responsibilities of the orchestration framework is to scale up or down the containers [14]. Some of the available orchestration frameworks are Docker swarm, Kubernetes, Google container engine, Cloud Foundry's Diego and Amazon ECS. However, this work only focuses on the Docker swarm as we use only Docker swarm to our proposed solution.

Docker swarm

Docker swarm is an orchestration platform that allows users to manage a cluster of the system running Docker platform. These containers are referred to as nodes and can either be physical or virtual. The Docker engine CLI is used to initiate and manage swarms that eliminate the need for any external orchestration software. Docker swarm supports auto-scaling as Docker manager allocate/remove tasks to/from worker nodes. It provides redundancy and performs failover in case of a node failure. A swarm can contain two kinds of nodes, managers and workers. Docker uses features of Linux kernel such as *namespaces* and *cgroups*. *Namespaces* make sure that processes running inside one container do not see the processes running inside the host or other containers running inside the same host. None of the containers can access the network ports or sockets used by another container. Containers use their network ports to communicate with external hosts and other containers. Similar to the physical hosts, containers are connected using a bridge interface. In addition to namespaces-based security, Docker also uses *cgroups*

[2] https://hub.docker.com

[3] https://store.docker.com

[4] https://quay.io

(control groups) to enforce security features of Linux kernel. *cgroups* ensure the fair allocation of resources among containers. *cgroups* provide security against the denial of service attacks stopping the ill distribution of host resources among containers. *cgroups* offer effective resource management in a multi-tenant environment such as Docker.

Nodes in a Docker swarm uses TLS for authentication and authorization with swarm manager. The communication among nodes is also encrypted. Docker has a built-in public key infrastructure to enforce security in the swarm [15]. Swarm manager generates a self-sign certificate along with a key pair to initiate secure communication with other nodes. Docker also provides users with an option of using external certificate authority. Manager nodes generate manager token and worker token. The token includes a random secret and digest of root CA's certificate. Every joining node validate the root CA certificate using the digest. Similarly, swarm manager ensures the authenticity of joining node using the random secret.

- **Swarm Manager** is responsible for allocating tasks to the other nodes. It maintains the cluster state, schedule services and serves as an endpoint API for swarm users [16]. Users execute Docker management commands on swarm manager only. Swarm manager also keeps track of the resources of other nodes and allocate tasks accordingly. The manager follows many strategies to allocate resources to the nodes. The *emptiest node* strategy allows swarm manager to fill the least utilised node first whereas using *Global*, manager allocates one instance of the container at least once. There can be more than one manager within a Docker swarm. It is ideal to have an odd number of swarm managers to take advantage of the fault-tolerant nature of the Docker swarm. The maximum number of swarm manager allowed per swarm by Docker is seven. However, only one of the manager can act as the swarm leader. The leader is elected from manager nodes using the Raft consensus algorithm(described later). Adding more manager nodes does not necessarily mean high scalability and better performance. Manager node can have one of three below-mentioned status.

    - The Leader status indicates that the manager node is looking after all swarm management and orchestration decisions.
    - The reachable nodes(including manager) remain available for the selection process and take part in Raft consensus quorum. Reachable manager nodes can be elected as the new leader if the original leader is unavailable.

  - The unavailable status shows that the manager node is isolated and can not interact with other manager nodes. This node can not take part in the Raft consensus quorum. Once the only available manager node is unresponsive, worker nodes should either join a new manager, or one of the worker nodes should be promoted as the new manager.

- **Worker nodes** are authorised by the manager to join the swarm. It is not possible to perform any Docker CLI actions on worker nodes individually. In a Docker swarm environment, worker nodes require at least one manager node to function. Swarm manager nodes are also worker nodes by default. For instance, a manager node in a single node swarm also acts as a worker. Worker nodes provide resources to swarm and take no part in decision making. The Docker agent running on worker nodes reports the status of the tasks assigned to them by swarm manager. It is also possible to promote a worker node as a swarm manager and vice versa. Worker nodes maintain three availability status based on the availability of the resources.

  - **Active** indicates that a manager node can assign more tasks to it.
  - **Pause** means that it can not accept any more tasks from the manager. However, it continues executing already assigned tasks.
  - **Drain** shows that the manager can not assign anymore task to this worker node. Furthermore, Scheduler needs to shift already running tasks to some other working node with active status as this node can not process any ongoing tasks as well.

Raft consensus algorithm for Docker swarm

The Raft consensus algorithm is used to implement fault tolerance within the network. A consensus is achieved in a multi-host system when all available hosts agree on specific values. A consensus decision is performed based on the accepted value. In case of Docker swarm, the manager maintains a consistent internal state using the Raft algorithm. It also ensures that managers are performing orchestration tasks [17]. Raft consensus decision elects a new leader from available managers in case the current leader becomes unavailable. For an N manager node system, Raft allows the system to function until (N-1)/2 failures occur and decides to receive the majority or quorum of (N/2)+1 manager nodes on absolute value. For instance, in a five node robust system, it is critical to have three available nodes for the system to process new tasks. Swarm leader logs all changes and activities of the swarm. These logs are then replicated to other managers. It ensures that all managers are always ready to fill the leader's position. Each manager has the similar but encrypted logs.

Docker swarm Traffic management

Network traffic in the Docker swarms consists of *control and management plane traffic* and *application data plane traffic. control and management plane traffic* is always encrypted and includes swarm management messages. Whereas, *application data plane traffic* is the traffic coming from external hosts [18]. It also includes traffic generated by existing containers in the swarm. Both, the *Control and management plane* and *application data plane traffic* use the same interface. However, it is also possible in Docker version 17.06 onwards to use separate interfaces for each kind of traffic. Furthermore, Docker swarm utilizes the network concepts below to handle the network traffic of the system.

- **Overlay networks** use Docker overlay network driver. It manages the interaction among the swarm nodes. Users can attach services to overlay networks for service to service communication. Overlay network defines the scopes named as the swarm, local, host and global. For instance, an overlay network with a scope of swarm allows connections from all nodes participating in the swarm. It is also possible to customize overlay networks such as a setting user-defined subnet and gateway. Service discovery in the overlay network allows Docker to manage the communication between the external service client and individual Docker swarm nodes. Service discovery enables the abstraction of swarm nodes details from the external client. Service discovery can be performed by using virtual IP (VIP) or customised request-based load-balancing at Layer 7 using DNS round robin (DNSRR). Docker allows users to configure those as mentioned above for each service individually.

    - Docker assigns a virtual IP to the services as it is attached over the network. External clients then use the assigned virtual IP to communicate to the service. Docker maintains the information of workers running the service along with their interaction with the external client.
    - In DNS round robin (DNSRR), Docker creates DNS entries for the services. The client connects to one of the obtained IP addresses upon the resolving DNS query. It allows users to use the customised load balancer.

- **Ingress network** is also an overlay network. It is used to handle internal load balancing for the deployed services. It is created automatically at the time of initialisation of a Docker swarm. It is also possible to customise it for Docker version 17.06 onwards.

- **docker_gwbridge** connects the overlay networks to the individual docker's private network. It is also created automatically when a node joins the swarm.

Chapter 3

Serverless Computing

3.1 Evolution of Cloud Computing

Cloud computing is a paradigm where computing services such as storage, compute servers and databases are provided to end users over the Internet by cloud providers. The providers charge users per usages for availing the services. This billing model is similar to the day to day services such as electricity, gas, and water. Cloud computing environment consists of a cloud provider that provides on-demand, highly scalable resources over the Internet to the customers. These resources can be an application, a database or simple virtual machine with no operating system installed. The provider is responsible for the most of the infrastructure management. However, end-user is still responsible for few tasks such as selection of operating system, capacity, and program execution. The provider can swiftly deploy and scale the resources while achieving multi-tenancy. Whereas, the end-user can achieve significantly reduced setup cost with fewer management efforts. The evolutions of cloud computing can be divided into following models.

- **Infrastructure as a Service (IaaS)** is the model of cloud computing, where providers provide the infrastructure such as servers, database, and data centre space by usage. Providers of this model mostly use commodity hardware distributed across the Internet. Using this model, an end user does not have to worry about setup, time-consuming procurements, and scaling of infrastructure. Amazon EC2[1], RackSpace[2], and GoGrid[3] are some example of the IaaS model of cloud computing.

[1] https://aws.amazon.com/ec2/

[2] https://www.rackspace.com/

[3] https://www.datapipe.com/gogrid

- **Platform as a Service (PaaS)** consists of the provider that provides ready computing platform along with the solution stack to the end-user and saves setup time. The customer gets the benefit of fast development and deployment of the applications with less interaction with the middleware. The customer also does not have to worry about software, provisioning, and hosting. Google app Engine[4], Hadoop[5], and Heroku[6] are the examples of PaaS where end users can directly deploy their applications and instantly make them available for users.

- **Software as a Service (SaaS)** is a cloud computing model where end users avail the web-based applications deployed on the cloud servers without any deployment and management worries. These cloud-hosted applications are accessed using a web browser. End users get the benefit of a fast start, on demand and location independent access and dynamic scaling. However, this approach also comes with the downside of less control available to the end-user. Facebook and Gmail are some of the popular SaaS-based services.

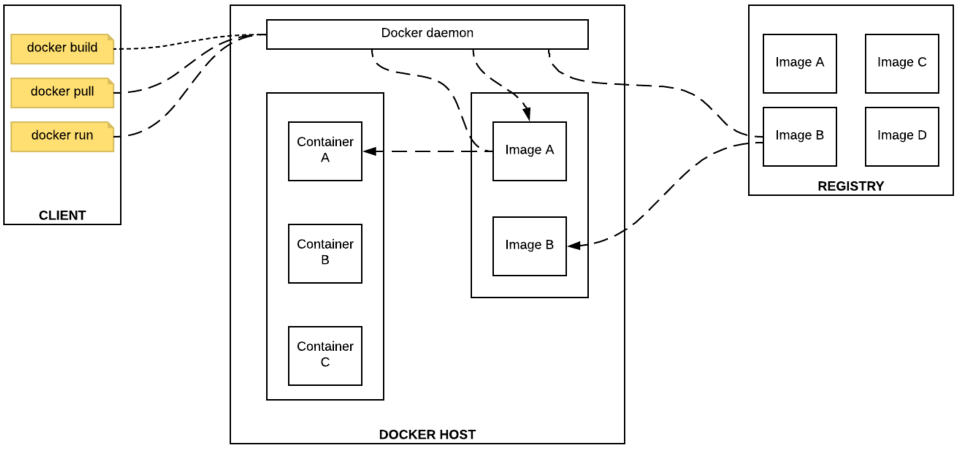

Figure 3.1: Docker architecture

- **Container** is a lightweight solution where virtualisation is achieved using the host kernel without needing a hypervisor. Containers give an advantage of reduced runtime and memory overhead as they run as a

[4] https://cloud.google.com/appengine/
[5] http://hadoop.apache.org/
[6] https://devcenter.heroku.com/

process in the host operating system. Containers are typically stateless and ephemeral. They take less time in starting the service and often give the near-native performance. Docker[7] and Kubernetes[8] are the examples of current container-based solutions. In Docker-container environment, Dockerfile contains the sequence of commands responsible for building the container image and executes these commands in the sequence. It is also feasible to use multiple containers as a cluster. We have discussed Docker containers in detail in Chapter 2. Figure 3.1 shows the architecture of the Docker engine containing a client, Docker host, Docker engine and image registry.

- **Serverless computing** Serverless computing is the latest cloud computing model specifically built for ephemeral, stateless and event-driven applications. The serverless computing model is based on on-demand horizontal scaling approach as hosted applications are required to scale up and down instantly. It also assimilates the "pay as you go approach" of cloud computing since users are billed for the actual usage at a millisecond granularity. A more formal definition of the serverless computing is “Serverless architectures refer to applications that significantly depend on third-party services (knows as Backend as a Service or ‘BaaS’) or on custom code that’s run in ephemeral containers (Function as a Service or ‘FaaS’.)” [19]
  Serverless computing addresses present issues in cloud computing models such as relatively high setup cost, user-end management, inefficient use of system resources and auto-scaling. The model is designed to support minimum user management efforts and event-driven architecture. The serverless logic is also known as function. Most of the available serverless projects use ephemeral containers to run these functions as a stateless service based on defined triggers and rules. Serverless functions are not limited to any specific programing language or libraries and provide flexibility of using multiple programming languages to write a function. It is also possible to wrap the function inside a container which allows the use of any possible programming language even if the serverless framework does not directly support it. A developer is only required for deploying the code to the provider’s infrastructure whereas the provider is responsible for the management, auto-scaling and execution of the function. Auto-scaling is horizontal and allows developers to handle a large burst of requests without any manual intervention.

[7] https://www.docker.com/

[8] https://kubernetes.io/

Apache OpenWhisk, OpenFaaS, Kubeless, iron.io, AWS Lambda and Fission are some available serverless project [20]. AWS Lambda is part of Amazon Web Services whereas Apache OpenWhisk, OpenFaaS, Kubeless, iron.io and Fission are available as open-source projects. The rest of the chapter reviews the aforementioned open-source projects in detail.

3.2 Apache OpenWhisk

Apache OpenWhisk is an open-source cloud platform which supports distributed and event-driven execution model of serverless computing [21]. IBM originally initiated the project and later continued as an open-source project under Apache license. The user writes a function which triggers in response to certain requests such as an HTTP request or feed based on predefined rules. It supports JavaScript, Swift, Python, PHP, Java and Golang as runtime programming languages. Additionally, programmers are allowed to develop in any language and run it inside a Docker container.

Figure 3.2 explains the high-level architecture of Apache OpenWhisk and related components are explained below.

- **Action** is the execution logic written in any available programming language or binary code embedded inside the docker container. An action can be invoked manually from OpenWhisk API, CLI, or iOS SDK. OpenWhisk supports chaining of multiple actions where the output of the first action in sequence serves as input to the next action. Actions take input in JSON format and produce output in the same format.

- **Event** is a change in OpenWhisk environment, which may or may not lead to trigger an action. For instance, smoke detector reading, a commit to Git repository, writing data to the database are some examples of OpenWhisk events.

- **Rule** defines the conditions required to deploy and execute an action. They establish the relationship between an action and a trigger by defining which action is executed in response to a certain trigger.

- **Triggers** are the events generated by some event sources such as IoT devices or web application. Triggers are designed as the channel for events. These events are responsible for the deployment and execution of functions based on certain rules.

- **Packages** are a shared collection of actions and triggers. They help programmers in integrating additional services to the event source code.

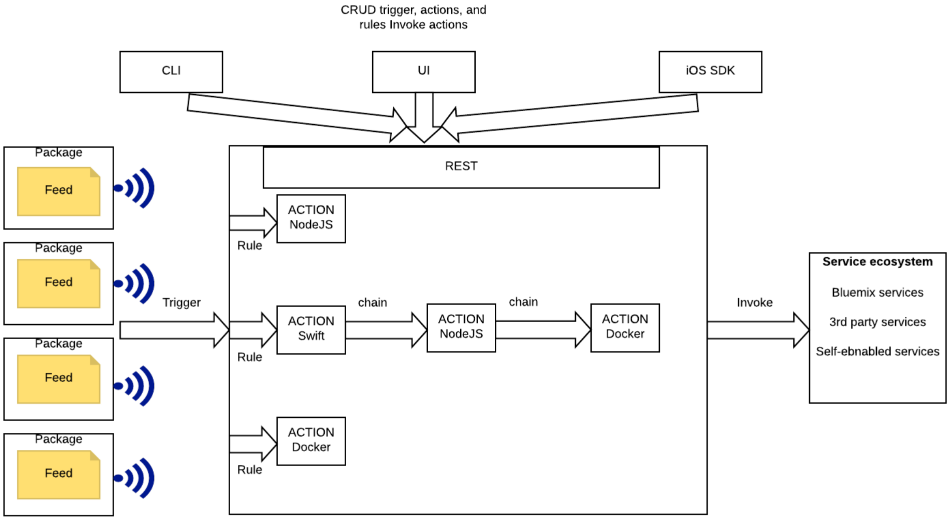

Figure 3.2: OpenWhisk high-level overview

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
def main(dict):
    fromaddr = ‘sender@gmail.com’
    toaddr = ‘receiver@outlook.com’
    msg = MIMEMultipart()
    msg[‘From’] = fromaddr
    msg[‘To’] = toaddr
    msg[‘Subject’] = ‘Apache OpenWhisk’
    body = ‘TEST’
    msg.attach(MIMEText(body, ‘plain’))
    server = smtplib.SMTP(‘smtp.gmail.com’, 587)
    server.ehlo()
    server.starttls()
    server.login(fromaddr, ‘password’)
    text = msg.as_string()
    server.sendmail(fromaddr, toaddr, text)
    return {"Status", "Success"}
```

Listing 3.1: OpenWhisk python function example

Listing 3.1 shows OpenWhisk function written in python triggering an email in response. The code returns success once after successful delivery of the email.

3.2.1 OpenWhisk internal architecture

The internal architecture of OpenWhisk shown in Figure 3.3, includes components such as ngnix, CouchDB, controller, Kafka and Docker container-based invoker. We discuss the roles of components as mentioned earlier next.

- **Nginx** is an HTTP-based reverse proxy server that receives commands coming from OpenWhisk API. The API is RESTful in design and sends HTTP requests to nginx server. It can be considered as the entry point of OpenWhisk architecture. The nginx server forwards the received valid HTTP request to the controller.

- **Kafka** is an open-source Apache framework responsible for providing high throughput and low latency reliable communication for real-time data feeds. It uses messages buses to communicate to controller and invokers. Kafka buffers the communication between controller and invokers to make sure the availability of the communication messages in the possibility of a system crash. The controller shares the messages and required parameters with Kafka. Upon getting confirmation from Kafka, the user is sent a notification. Invokers then pick a task from the Kafka placed via the message bus.

- **CouchDB**[9] is an Apache open-source non-relational database which stores data in JSON format. It responds to the authentication and authorisation request of the controller by verifying the privileges of the user. It also stores the result from invokers before returning it to the controller. CouchDB contains whisks database which contains actions and their meta-data.

- **Controller** is a REST API implemented in scala. The controller receives requests from nginx and serves as an interface for user's actions. The controller API uses Akka[10] and Spray[11]. Akka is used to build concurrent and distributed event-driven applications for scala and java whereas, Spray provides REST and HTTP support to Akka-based applications. The controller analyses the user's requests and contacts CouchDB to authenticate and authorise the request. Upon getting a

[10] https://akka.io/

[11] http://spray.io/introduction/what-is-spray/

positive response from CouchDB, the controller loads task from the Whisks database and fetch the information about invokers. It then places the task on Kafka addressing to the selected invoker.

- **Invokers** are responsible for executing the function after fetching the Kafka message from message bus. Invokers execute actions in an isolated environment inside a Docker container. The container is destroyed after publishing the result to the database.

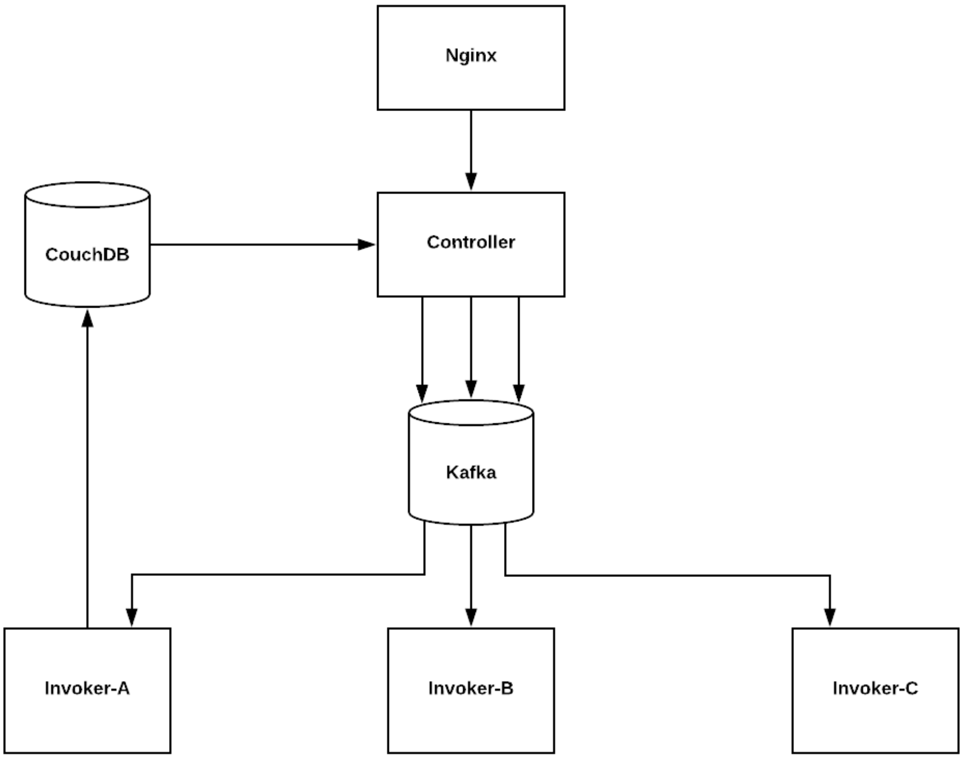

Figure 3.3: OpenWhisk: flow of processing

3.2.2 Setup and Triggers

OpenWhisk can be set up and use in two ways.

- **Local setup** requires users to clone the OpenWhisk's Github repository[21] to the local server. Upon cloning, users can install the project locally and requires Docker as a pre-requisite. Local installation supports

CouchDB and cloudant as databases. Cloudant is an IBM proprietary NoSQL database and provided as a database as a service(DBaaS) by IBM cloud. In contrast, CouchDB is an open-source NoSQL databased and can be installed locally. OpenWhisk components are available to install in both centralised and distributed manner. In a centralised installation, all components are installed on the same machine, whereas distributed installation allows components to be installed on different machines.

- **IBM cloud-based setup** is a paid service provided by IBM cloud as IBM functions. IBM cloud function uses the only cloudant as a database.

Furthermore, a command line utility is available for users to deploy, modify and execute functions. The utility for the local setup can be cloned and installed form the Github project[22]. IBM cloud's command line utility also provide similar functionality and can be download from IBM cloud webpage[12].
OpenWhisk supports the following triggers in the IBM cloud [23] environment.

- **Cloudant-based Trigger :-** This trigger can be defined based on any modifications made to the cloudant database.
- **Customized Trigger :-** OpenWhisk allows developers to customise the defined triggers. An example of a customised trigger is a trigger based on a POST request.
- **GitHub Trigger:-** Any changes to the Git repository can also generate an action trigger.
- **MessageHub Trigger:-** These triggers invoke an action when a new message is written to the queue.
- **Mobile Push Trigger:-** A push notification to the mobile application defines this trigger.
- **Periodic Trigger:-** Using this trigger, one can define desired date and time to execute an action.

[12] https://console.bluemix.net/docs/cli

3.3 OpenFaaS

OpenFaaS is an open-source project based on the function as a service (FaaS) model of the serverless framework. It runs containers in the background. In addition to Docker containers, OpenFaaS also supports Kubernetes.

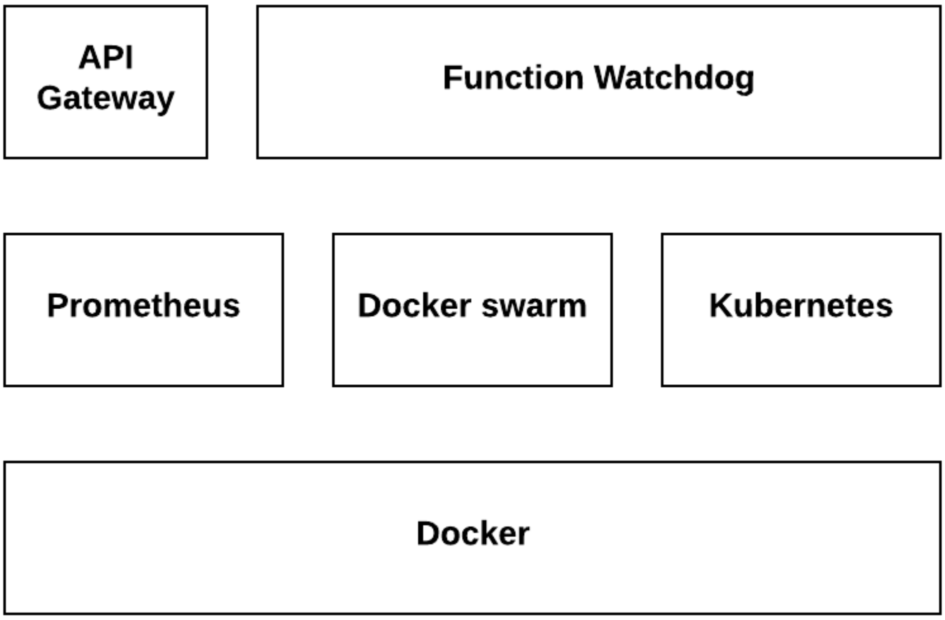

Figure 3.4: OpenFaaS components

3.3.1 Architectural Overview

Figure 3.4 shows next mentioned components of OpenFaaS architecture [24].

- **Function WatchDog** is an HTTP server written in Go programming language. It converts Dockerfiles into serverless functions. It serves as an entry point for the HTTP requests. Additionally, it interacts with the processes and caller by sending and receiving standard input and output. Standard input(STDIN), output(STDOUT) and error(STDERR) are three major data streams in Linux-based operating systems. STDIN helps the process to read information from user whereas STDOUT allows a process to write the information back to the user. STDERR helps a process to write error information.

- **API Gateway** is responsible for the auto-scaling by contacting Docker Swarm or kubernetes[13] components. It also has a user interface which allows a user to invoke their functions from the browsers. Developers use an API gateway to push and deploy the functions.

- **Prometheus**[14] keeps track of cloud-native metrics and monitors the environment. The cloud-native approach focuses on designing, building and deploying highly scalable and fast application by using the advantages of cloud computing model such as infrastructure as a service(IaaS). Prometheus reports the cloud-native metrics to API gateway.

- **Swarm and Kubernetes** are container orchestration solution for deployed functions and used for auto-scaling in the OpenFaaS environment.

OpenFaaS deploys its components using the Docker container. It is evident from the listing 3.2 that critical components such as function watchdog, Prometheus, and API gateway run as a container-based service using Docker.

```
CONTAINER ID        COMMAND
265d8d765ada        "/bin/alertmanager..."
740ed1f0cecf        "fwatchdog"
5aa428c77b35        "fwatchdog"
9f5022f23ceb        "fwatchdog"
0292d506b5f1        "fwatchdog"
33bec77614db        "fwatchdog"
be894bf951df        "fwatchdog"
de3aaa490ad2        "fwatchdog"
7d4d2593f036        "/usr/bin/fwatchdog"
9124556b624c        "/bin/prometheus␣-..."
832e8ebeb17b        "./gateway"
```

Listing 3.2: OpenFaaS deployement using Docker swarm

3.3.2 OpenFaaS Setup

Running OpenFaaS functions requires Docker community edition 17.05 as pre-requisite. OpenFaaS can be cloned from the project's git repository [24]. Docker is required to be initialized in the swarm mode. The user can deploy functions using both the CLI and the user interface on localhost over port 8080. However, CLI requires a separate installation and can be installed from the github[15]. OpenFaaS support multiple languages such as Go, Ruby,

[13] https://kubernetes.io

[15] https://github.com/openfaas/faas-cli

Python, and Node.js as it's function runtime environments. Additionally, the user can write a function in a different language and wrap it inside the Dockerfile. FaaS CLI creates three files named as *handler.py*, *requirement.txt* and *<function-name>.yml.*

Handler.py contains the actual code, whereas *requirements.txt* consists required external modules to run the function. *<function-name>.yml* includes metadata about the function. It includes details of the remote gateway, function and function's language, path of the handler, timeouts and Docker image. Detailed setup and installation process can be referred from [25]. Listing 3.3 is an example of an OpenFaaS function to check if a word exists in a given webpage.

```python
import requests
import json

def handle(req):
    result = {"found": False}
    json_req = json.loads(req)
    r = requests.get(json_req["url"])
    if json_req["term"] in r.text:
        result = {"found": True}
    print json.dumps(result)
```

Listing 3.3: OpenFaaS python function

We can pass the arguments in JSON format and fetch the result using curl command. For listing 3.3, we can use command mentioned in listing 3.4 to return the result. As we can see, the term "Botanical Gardens" is present at the University of Tartu's webpage on Wikipedia.

```
curl localhost:8080/function/hello-python--data-binary
'{"url": "https://en.wikipedia.org/wiki
/University_of_Tartu","term": "Botanical␣fGardens"}'
{"found": true}
```

Listing 3.4: OpenFaaS python function execution

OpenFaaS auto-scaling

The openfaas architecture includes function auto-scaling by scaling up or down the function deployment on-demand. OpenFaaS accomplishes the auto-scaling using Prometheus matrices. The *AlertManager* generates an alert to API Gateway once Prometheus trigger a defined usage alert matric. The communication between *alert manager* and *API gateway* takes place using */system/alert* route. Listing 3.5 shows the alert manager rule for Docker swarm [26]. OpenFaaS has a defined value of min and max replicas. By default, the minimum possible number of replicas is 1 whereas, one can launch maximum 20 replicas while scaling up. It is also possible to disable auto-scaling by setting same value for min and max replicas or setting *com.openfaas.scale.factor=0*. The scaling factor defines the number of replicas initiated upon the alarm is generated and is set to 20% by default.

```
groups:
- name: prometheus/alert.rules
  rules:
  - alert: service_down
    expr: up == 0
  - alert: APIHighInvocationRate
    expr: sum(rate(gateway_function_invocation_total
    {code="200"}[10s])) BY (function_name) > 5
    for: 5s
    labels:
      service: gateway
      severity: major
      value: '{{$value}}'
    annotations:
      description: High invocation total on
      {{ $labels.instance }}
      summary: High invocation total on
      {{ $labels.instance }}
```

Listing 3.5: Alert manager for Docker swarm

3.4 IronFunctions

IronFunctions[16] is an open source platform for serverless computing initiated by iron.io[17]. The project is written in Go, whereas users can define functions in any programming language. Additionally, IronFunctions also supports AWS lambda functions. It is feasible to use both Docker and Kubernetes to deploy IronFunctions' components. A command line interface is also available to build and deploy functions using the command line. Listing 3.6 is an example of IronFunctions' serverless function written in Go.

Functions are written in order to parse the standard input, perform an action and respond/update. Developers are responsible to define environment variables such as *REQUEST_ URL*, *ROUTE*, *METHOD*, *HEADER_ X*, *X*. *REQUEST_ URL* is the full URL of the request. *ROUTE* is the matched

[16] https://github.com/iron-io/functions

[17] https://www.iron.io/

route of the deployed function. A method can be any method such as GET, POST called for an HTTP request. *HEADER_X* is used for the HTTP header of the request where *X* denotes the header name. *X* can be any variable or configuration value.

```go
package main
import (
    "encoding/json"
    "fmt"
    "os"
)
type Person struct {
    Name string
}
func main() {
    p := &Person{Name: "Bob"}
    json.NewDecoder(os.Stdin).Decode(p)
    fmt.Printf("Hey%v!", p.Name)
}
```

Listing 3.6: IronFuntions example

IronFunctions platform function also supports logging. Logs can be configured using the programming language used to write function and viewed as standard errors(STDERR). Serverless functions can be created and managed using following commands.

- **Init** takes the written function file as an input and generates a *func.yaml* file in the project folder. *Init* is also used to create the function using Dockerfile as an input.
- **Bump** perform an increment to the version number of a configuration file.
- **Build** is used to build an image from your function/Dockerfile.
- **Run** tests the function before creating an image.
- **Push** writes the function's image to DockerHub.
- **App** creates an API for the deployed function. Moreover, a route can be created for the function API to make the deployed function reachable. A route allows a user to define a path in the application that maps to a function.
- **Deploy** IronFunctions also supports bulk deploy using deploy command where are all function present in the directory are scanned. Upon scanning, functions are rebuilt and pushed to the Docker Hub registry.

Commands mentioned in listing 3.7 shows the usages of different CLI commands.

```
# Mention Docker hub Username while creating func.yaml
fn init USERNAME/myfunc
# Build created function
fn build
# Test the function
# - in, eg: ‘cat myfunc.payload.json | fn run’
fn run
# Build and push upon successful testing
fn build && fn push
# Create app for your deployed function
fn apps create funcapp
# Create a route for deployed function
fn routes create funcapp /myfunc
# Update the function
fn bump && fn build && fn push
# Update the route
fn routes update myfunc /myfunc
```

Listing 3.7: IronFuntions CLI commands

IronFunctions support two type of authentication.

- **Service level authentication** authenticates requests coming from clients. *JWT_AUTH_KEY* variable is responsible for enforcing service level authentication.
- **Route level authentication** is responsible for performing authentication for a request made to a specific route.

3.5 Kubeless

Kubeless is a serverless platform built on the top of Kubernetes. Kubeless supports multiple runtime environments including Python, Node.js, Ruby and PHP. It also provides users with a CLI to execute and manage serverless functions. The CLI handles HTTP requests and Kubernetes kubectl[18] commands. Kubeless is written in Go and uses Kubernetes features such as autoscaling, monitoring and API routing. The executable piece of code is called function, which also contains dependencies and runtime environment's metadata. Functions are represented using a custom resource definition (CRD)

[18] https://kubernetes.io/docs/reference/kubectl/overview/

feature of Kubernetes. The platform uses Kubernetes pods to run different runtime environments. A Kubernetes pod is a group of one or more containers that share system resources. A pod also includes the specifications about the execution of containers. The dependencies are loaded using *init* containers. The function is exposed to an external network using the ingress route. Using different CDRs for different functions allows Kubeless to maintain the disjunction among deployed functions.

```
def func(event, context):
  print event
  return event['data']
```

Listing 3.8: Kubeless function example

Listing 3.8 is an example of the kubeless function with HTTP trigger. Event parameters have the information about the event source whereas the context parameter contains information about the function itself. Kubeless provides different methods to support the life cycle of a function.

- **Deploy** is used to deploy the function over the runtime framework. Functions can be invoked directly or using triggers.
- **Execute** provides us with the feasibility of invoking a function directly.
- **Get** command is used to extract the function's metadata.
- **Update** is used to update the changes to the function and its metadata.
- **Delete** allows developers to remove the deployed function along with its metadata.

Kubeless also provides some helping functions such as *List* and *Logs*. An example of function deployment using CLI is given in Listing 3.9.

```
kubeless function deploy get-python --runtime \
                        python2.7 --from-file \
                        func.py --handler func.foobar
```

Listing 3.9: Kubeless function deployment

Triggers are used to execute the deployed function as soon as a certain event arises. Any single trigger can be used for multiple functions and managed using methods such as *Create*, *Update*, *Delete* and *List*. Kubeless allows developers to define HTTP triggers, Cronjob triggers and Kafka triggers for their functions.

- **HTTP triggers** include a function name that is to be executed upon HTTP request. Hostname option is used for the virtual hostname. A route to the function is defined using a path variable. HTTP triggers also have an option to enable TLS. Listing 3.10 is an example of an HTTP based trigger.

```
apiVersion: kubeless.io/v1beta1
kind: HTTPTrigger
metadata:
  labels:
    created-by: kubeless
  name: get-python
  namespace: default
spec:
  function-name: get-python
  host-name: get-python.192.168.99.100.nip.io
  ingress-enabled: true
  path: func
  tls: false
```

Listing 3.10: Kubeless HTTP trigger

- **Cronjob triggers** Cronjob triggers are schedule-based triggers and need a cronjob in order to execute the function. Cronjobs are based on linux utility cron which allows a piece of code to run at a specific time and date. The configuration includes a function *name* and *schedule* field. An example of cronjob trigger is given in listing 3.11.

```
apiVersion: kubeless.io/v1beta1
kind: CronJobTrigger
metadata:
  labels:
    created-by: kubeless
    function: scheduled-get-python
  name: scheduled-get-python
  namespace: default
spec:
  function-name: scheduled-get-python
  schedule: '* * * * *'
```

Listing 3.11: Kubeless cronjob trigger

- **Kafka triggers** use Kafka topics to invoke a function. Their configuration includes *functionSelector* and *topic* fields. *Topic* includes Kafka topics whereas *functionSelector* fetches the list of functions upon matching the topic condition.

```
apiVersion: kubeless.io/v1beta1
kind: KafkaTrigger
metadata:
  labels:
    created-by: kubeless
  name: s3-python-kafka-trigger
  namespace: default
spec:
  functionSelector:
    matchLabels:
      created-by: kubeless
      topic: s3-python
  topic: s3-python
```

Listing 3.12: Kubeless Kafka trigger

Kubeless uses Prometheus to monitor the function and to generate metrics. Function utilization metrics are gathered and can be displayed using Prometheus user interface. Additionally, Kubeless also supports Grafana[19] to visualize the metrics. In order to support auto-scaling of deployed functions, Kubesless take advantage of *HorizontalPodAutoscaler* of Kubernetes. It is also feasible to auto-scale functions depending upon CPU usage using *–cpu* command. *–cpu* command allows developer to define a CPU usage limit at the time of deployment of the function. Functions can also be build and deployed from Docker registries such as Dockerhub. However, the user can pull an image only from a single registry.

3.6 Fission

Fission is another open-source serverless framework. As a developer, one can use Fission with both Docker and Kubernetes. Fission is developed in Go and supports multiple function deployment environments such as Node.js, Python, Ruby, Go, PHP, Bash and .net. Fission function are deployed upon creating the environment in the desired programming language. The platform supports both synchronous and asynchronous functions. In order to access the deployed function using HTTP requests, a route has to be created for the deployed function. Listing 3.13 is an example of a basic fission function written in Node.js.

```
module.exports = async function(context) {
    return {
```

[19] https://grafana.com/

```
        status: 200,
        body: "Hello␣world!\n"
    };
}
```

Listing 3.13: Fission function example

Fission supports different methods to support the lifecycle of a function.

- **Create** generates the function and its metadata. The executor type and scaling in case of *Newdeploy* executor is defined at the time of creation of the function.
- **Get** is used to retrieve the function code.
- **Update** allows developers to modify the function's code.
- **Test** is used to check the expecting behaviour of a function before deploying it.
- **Log** returns function's log and is used for further troubleshooting.

Fission supports pool-based executers (*Poolmgr*) and new-deployment executor (*Newdeploy*). These executors are defined within the deployment environment and establish the creation of Kubernetes Pods to deploy functions and their capabilities. Pool-based executers create environment pods at the time of creation of the environment. These pods can be generic or specialised at the time of allocation to a particular function. These pods are *warm* or ready to use, hence accelerate the pod allocation and function execution process. If the function execution is finished and allocated pod is idle, the pod is removed after a certain interval of idle duration. This executor type is favourable for the functions with low latency requirement. However, it does not provide the privilege of auto-scaling. The solution to the auto-scaling issue with *Poolmgr* is second executor type termed as *Newdeploy.* It creates pods along with the service execution and takes leverage of *HorizontalPodAutoscaler* feature of Kubernetes to perform horizontal scaling. The requirements are specified while writing the function and are given priority over the general requirement mentioned in the execution environment. *Newdeploy* is ideal for asynchronous functions where minimising the latency is not a primary requirement. It is also feasible to keep the idle pods available so that they can be used to minimise the latency and execution time. However, there is always a trade-off between latency and resource consumption in this scenario.

Fission's serverless functions can be executed using triggers. Fission framework provides three categories of triggers.

- **HTTP Trigger** are used to trigger a function in response to an HTTP request method.
- **Time Trigger** enables developers to use custom periodic triggers to invoke their function.
- **MQ Trigger** MQ triggers are topic based triggers similar to Kafka-based triggers in Kubeless. This trigger uses messages queue topics to execute the function. Fission supports *nats-streaming* and *azure-storage -queue* message queues for MQ triggers. It also allows developers to add dependency packages/libraries with the code. The platform allows developers to customise the horizontal scaling based on the minimum and a maximum number of allowed CPU, scale and memory.

Features/projects	**OpenWhisk**	**OpenFaaS**	**iron.io**	**Kubeless**	**Fission**
Docker support	yes	yes	yes	no	no
Kubernetes support	yes	yes	yes	yes	yes
Arm deployment	no	yes	no	no	no
Default timeouts	60 sec	read - 25 sec write - 25 sec upstream - 20 sec	60 sec maximum	180 sec	user-defined
Support for async functions	yes	yes	yes	yes	yes

Table 3.1: Comparison of serverless platforms

Table 3.1 shows the comparison of discussed serverless platforms. Available platforms are compatible with i386, x86 and x64 architecture. Furthermore, it has been demonstrated the deploy Docker swarm [27] and kubernetes [28] over arm based devices such as Raspberry Pi. From the discussed serverless platforms, only OpenFaaS has expressly provided the possibility with the arm based deployment using Docker swarm [29].

Chapter 4

Design and Implementation

Resource-intensive processes in IoT are offloaded to cloud-based resources for further processing. However, there is always a trade-off between resource and time consumption in such offloading mechanisms. In the case of small computations, it is wise to perform the job locally at the IoT-gateway itself. Local execution of serverless computing functions saves not only computation time but also the network bandwidth. Privacy-sensitive tasks also apply restrictions on the computational offloading. For instance, computation at the public cloud is not an option for some Military Grid Reference System (MGRS)[1] coordinate-based applications. Some computations are both resource-intensive and time-sensitive. It is critical to formulating a solution that provides efficient resource utilisation with minimum possible latency.

This chapter explains the design and implementation of the deployment of OpenFaaS for IoT devices. We present and describe the system architecture which follows by an explanation of a function execution offloading algorithm.

4.1 System Architecture

The designed architecture spans across IoT-gateway, edge layer, fog layer, and cloud layer. The solution is designed by considering requirements that are critical for the deployment of the FaaS-based framework, as follows :

1. The developer knows the structure of the information generated by the IoT device. It is an essential condition as the function written by the developer uses the data as input. For instance, based on the smoke sensor reading, a function can display a warning to command-centre or even contact the nearest fire station.

[1] https://mappingsupport.com/p/coordinates-mgrs-google-maps.html

2. Developers write ephemeral functions or micro-services for the deployed platform. It is essential to respect the resource-constrained nature of IoT devices while writing such functions. Furthermore, the concept of serverless computing is entirely based on small functions that can be executed within a very short span of time.

3. The edge and fog layers have connectivity to the local IoT-gateway. This assumption is crucial for the function execution offloading.

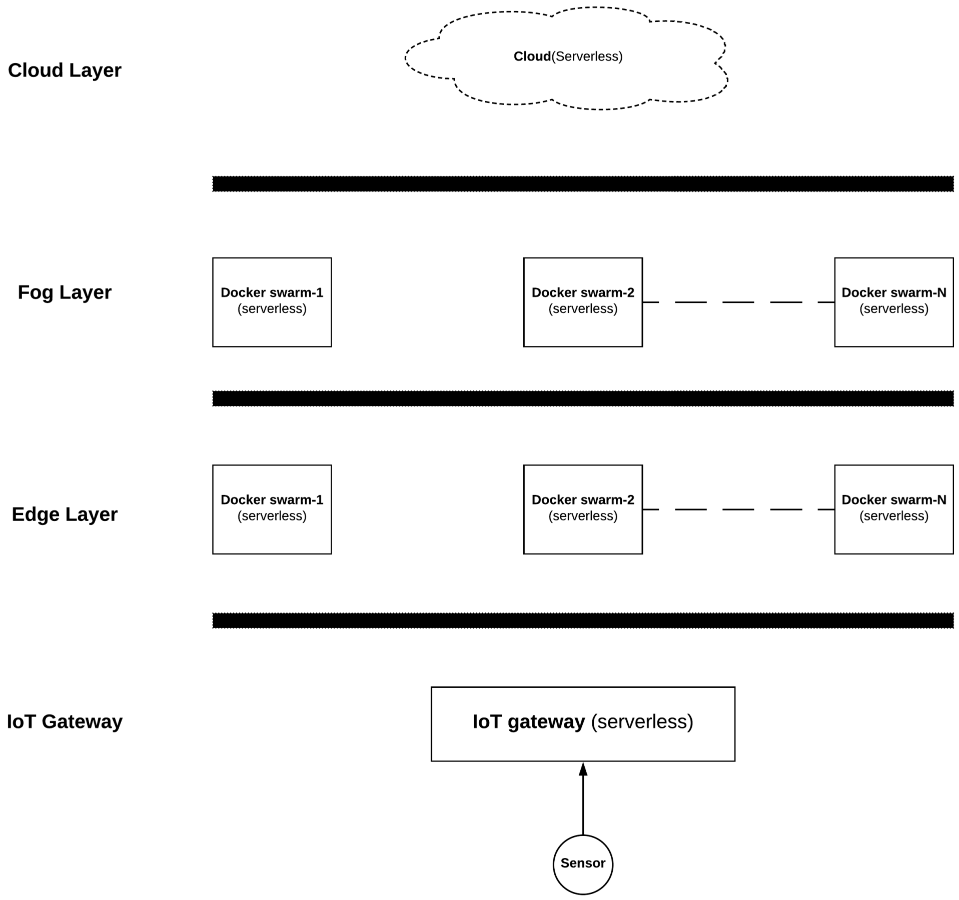
Cloud Layer
Cloud(Serverless)
Fog Layer
Docker swarm-1 (serverless)
Docker swarm-2 (serverless)
Docker swarm-N (serverless)
Edge Layer
Docker swarm-1 (serverless)
Docker swarm-2 (serverless)
Docker swarm-N (serverless)
IoT Gateway
IoT gateway (serverless)
Sensor

Figure 4.1: Multi-layered architecture

The architecture presented in Figure 4.1 can be divided into four layers named as IoT-gateway, edge layer, fog Layer, and cloud layer. Structure and functioning of each of the layers as mentioned earlier are discussed below.

- **IoT-gateway** is directly connected to the IoT devices and receive the data from them. OpenFaaS is deployed on the IoT-gateway with the help of Docker. We selected OpenFaaS due to the ease of deployment on arm architecture-based devices. Docker is initiated in swarm mode on the gateway to meet the pre-requisites of deploying OpenFaaS platform. Upon successful installation of the OpenFaaS, the required function can be built, pushed and deployed to the gateway. Function runtime environments are already installed with the OpenFaaS deployment in the form of Docker containers. These environments allow us to execute the function without installing the required runtime environments explicitly. The deployed function can be triggered by using both OpenFaaS user interface and the command line interface. However, additional runtime environment's installation is required to trigger the function programmatically.

- **Edge layer** executes the function in case IoT-gateway is not capable of the execution. This situation arises due to insufficient system resources such as power, CPU, and memory. Moreover, certain conditions associated with the function (asynchronous function type) can also result in function execution offloading to upper layers. The fog layer consists of available edge devices in the network offering their resources collectively for the computation of serverless functions. These devices are grouped in clusters using Docker swarm. Swarms not only serve as pre-requisites to the installation of OpenFaaS but also incorporate other features such as fault tolerance and high-availability.

- **Fog layer** receives the function execution when

  1. IoT-gateway and edge layer cannot handle the function execution.
  2. Edge layer is unavailable, or IoT-gateway does not get any response from the edge layer.

  We install docker on available IoT devices at the fog layer and initiates it in swarm mode to deploy OpenFaaS.

- **Cloud layer** is responsible for the function execution in below-mentioned conditions.

  - IoT-gateway, edge and fog layer devices do not have sufficient resources.
  - Edge and fog layer's devices are not available.

- The function is asynchronous as asynchronous functions consume more time and resources

Unlike lower layers, deploying OpenFaaS on the cloud layer does not give us the same benefits. For instance, employing Amazon EC2 instance to deploy OpenFaaS is expensive than using AWS lambda for serverless functions. We use IBM cloud functions to execute our serverless function in the cloud.

4.2 Function execution offloading

IoT-gateway initiates function execution of any synchronous function. There could be situations where IoT-gateway is unable to perform the function execution due to the lack of system resources. It is important to devise an offloading mechanism among the layers as explained in Section 4.1.

Offloading solution

The existing computational offloading solutions are not fitting to our requirement as we do not need to shift the computation but the function execution trigger. Functions are already deployed at each layer of our architecture. Hence, we propose a solution where serverless function execution is offloaded based on the availability of system resources. Our offloading mechanism from Figure 4.3 considers factors such as memory utilisation, remaining power, CPU utilisation and availability of active nodes in Docker swarm.

4.2.1 Terminology

This subsection addresses the terminology used in the Section 4.2.3. We define and explain the terms individually in the rest of this subsection.

- ***exec timeout*** is defined both in the OpenFaaS function meta-data and the OpenFaaS gateway configuration. It represents the function execution timeout value in seconds. The *exec timeout* value differs for each of the layers in our architecture considering their system resources. Table 4.1 shows the *exec timeout* values for all four layers of our architecture.

- ***Function bucket*** stores function name and current date-time value in key-value pair format. The IoT-gateway, edge and fog layers maintain this data structure. OpenFaaS checks the entry of a function in the bucket before executing the function. It does not proceed with the

Layer	**Platform**	***exec timeout***
IoT Gateway	OpenFaaS	20 seconds
Edge Layer	OpenFaaS	30 seconds
Fog Layer	OpenFaaS	45 seconds
Cloud Layer	Apache OpenWhisk	60 seconds

Table 4.1: *exec timeout* values

execution if the function name exists in the respective bucket. The bucket is flushed every 120 minutes. An example of the function bucket is shown in Listing 4.1.

```
{ "id": "func1", "time": "2018-05-27␣11:58:48..." },
{ "id": "func2", "time": "2018-05-27␣11:22:28..." },
{ "id": "func3", "time": "2018-05-27␣11:59:48..." },
{ "id": "func4", "time": "2018-05-27␣12:00:34..." }
```

Listing 4.1: An example of a function bucket

- ***System resources*** include available system memory, available CPU and, remaining power. In context of fog and edge nodes, system resources are simply determined by active nodes in the deployed Docker swarm. Docker determines the active nodes in swarms by tracking actual system resources of all participating nodes.

4.2.2 Communication for the offloading decision

The function execution offloading allows the layers to move the OpenFaaS function execution to the upper layer. Offloading takes place when a particular layer runs low on system resources. The communication among layers is illustrated in Figure 4.2 and works as follows.

1. The lower layer sends a multicast request to the defined address when it realizes about not having sufficient system resources to execute the OpenFaaS function.

```
{ "ip": "10.10.10.*", "active_nodes": "2" },
{ "ip": "10.10.10.*", "active_nodes": "0" },
{ "ip": "10.10.10.*", "active_nodes": "1" },
{ "ip": "10.10.10.*", "active_nodes": "1" }
```

Listing 4.2: Multicast response from upper layer

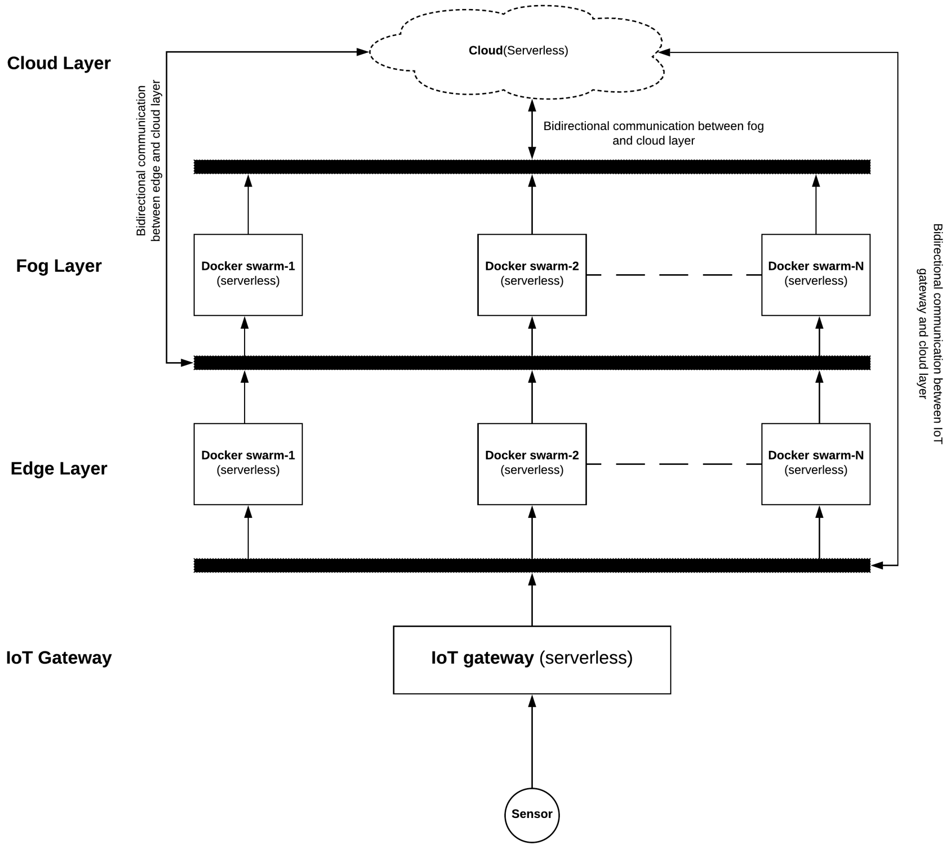
Cloud Layer
Cloud(Serverless)
Bidirectional communication between edge and cloud layer
Bidirectional communication between fog and cloud layer
Fog Layer
Docker swarm-1 (serverless)
Docker swarm-2 (serverless)
Docker swarm-N (serverless)
Bidirectional communication between IoT gateway and cloud layer
Edge Layer
Docker swarm-1 (serverless)
Docker swarm-2 (serverless)
Docker swarm-N (serverless)
IoT Gateway
IoT gateway (serverless)
Sensor

Figure 4.2: Communication for the function execution offloading

2. Docker swarms deployed at the upper layers listen to the multicast request. They respond with their respective IP addresses and the number of available active nodes in the particular swarm as shown in Listing 4.2.

3. Upon getting the multicast response from an upper layer, the lower layer takes following steps.

    - If it does not get any response from upper layers in the defined time limit, the lower layer moves the function to the next layer. For instance, If IoT-gateway does not get a reply from the edge layer, it sends the function execution to the fog layer.
    - When the lower layer gets the multicast reply from the upper layer, it checks the number of active nodes per swarm.

- Function execution is offloaded to layer next to the upper layer if there is no active node available at next upper layer.
- If the number of active nodes is same for all swarms, one of the available swarm is randomly selected, and function execution is delivered to that swarm.
- If none of the above two conditions exists, the execution is offloaded to the swarm with the highest number of active nodes.

4.2.3 Offloading algorithm explanation

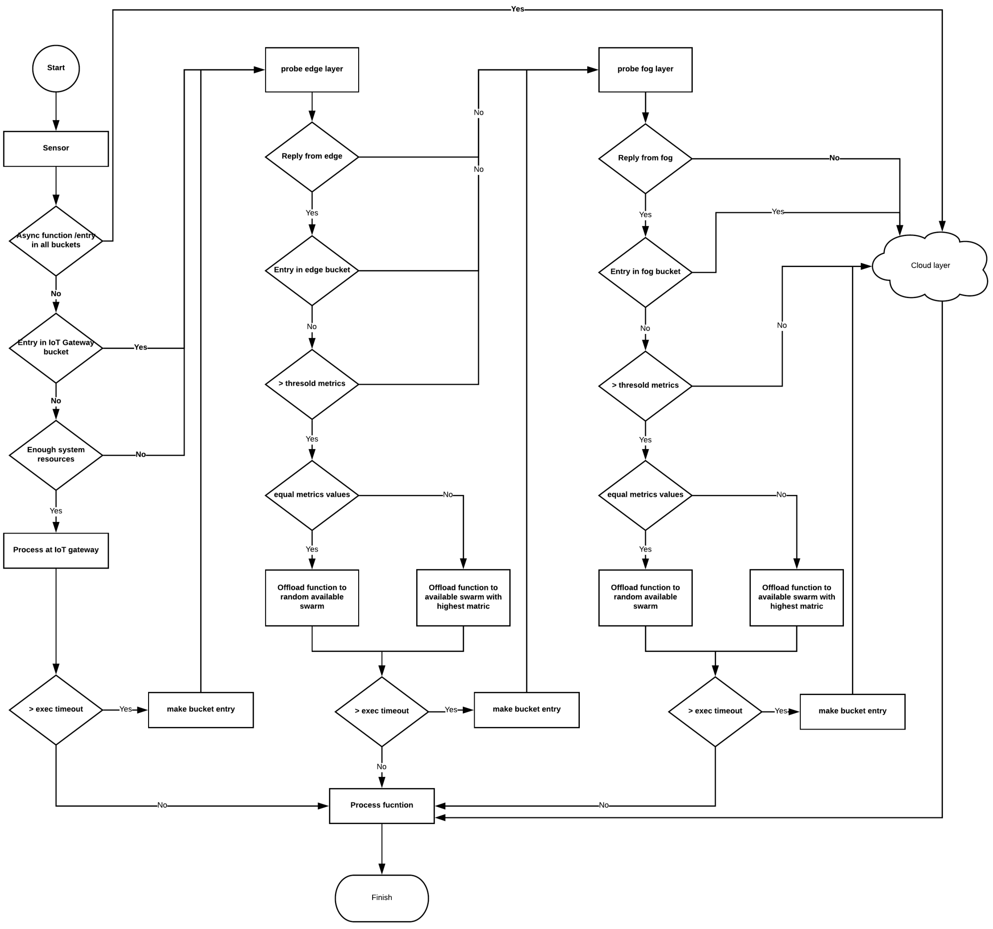

Figure 4.3: Function execution offloading flow diagram

We divide the function execution offloading flow at each layer for the ease of understanding.

Offloading flow at IoT-gateway

1. The IoT-gateway receives the data from the IoT device and triggers the function.

2. Asynchronous functions run longer than synchronous functions. IoT-gateway offload function to cloud layer when one of the following conditions exists:

   (a) If the function is asynchronous.

   (b) If function name exists in *Function bucket* at each layer.

3. IoT-gateway probes function to edge layer upon encountering following scenarios:

   (a) If function name exists in the *Function bucket* at IoT-gateway layer.

   (b) If IoT-gateway does not have enough *System resources.*

4. In the case of the synchronous function and IoT-gateway has adequate system resources; it starts the execution.

5. When an OpenFaaS function execution outlives the *exec timeout* without giving the output, the function name is stored in the function bucket with the current date-time stamp. Upon making the entry, IoT-gateway probes the next layer.

6. If function finishes processing within the defined value of *exec timeout*, gateway finished the function processing.

Offloading flow at edge layer

1. IoT gateway offloads the function execution to edge layer if the edge layer sends a multicast response in answer to the probe multicast request.

2. The function is offloaded to the next layer if:

   (a) current layer remains unresponsive for a defined time limit.

   (b) there is an entry of function name in *Function bucket* at edge layer.

   (c) If the metric value is lower than the threshold.

3. In case of more than one Docker swarm:
   (a) if obtained metrics are equal (equal number of active nodes), the function is randomly offloaded to one of the available nodes.
   (b) else, we offload the function to the swarm with highest metric.
4. If function exceeds layer's *exec timeout*, a *Function bucket* entry is made, and function execution is offloaded to the next layer
5. else, the function is processed at the edge layer successfully.

Offloading flow at fog layer

1. Function is offloaded to the fog layer if it sends a valid multicast reply.
2. The function is offloaded to the next layer if:
   (a) current layer remains unresponsive for a defined time limit.
   (b) there is an entry of function name in *Function bucket* at fog layer.
   (c) If the metric value is lower than the threshold.
3. In case of more than one Docker swarm:
   (a) if obtained metrics are equal (equal number of active nodes), the function is randomly offloaded to one of the available nodes.
   (b) else, we offload the function to the swarm with highest metric.
4. If function exceeds layer's *exec timeout*, a *Function bucket* entry is made, and function execution is offloaded to the next layer
5. else, the function is processed at the fog layer successfully.

Offloading flow at cloud layer

1. Upon receiving the function execution offloading, cloud layer processes the function and returns the result.

4.3 Implementation

We are using Raspberry Pi 3 Model B as the IoT gateway connected to *dth 11*[2] temperature and humidity sensor in our implementation. The fog layer is constructed using three docker swarms with three Raspberry Pi 3 Model B in each swarm. All Raspberry Pi devices are build using raspbian[3] operating system. We are using ubuntu 16.04 LTS as our edge layer device whereas IBM cloud functions is used as the cloud layer. For the database specific to our demonstration, we are using MongoDB 3.6 in high-availability. Our serverless function is collecting temperature and humidity values from the *dth 11* sensor and writing the data to the database. In a realistic scenario, the function is writing data in milliseconds to the database. However, we successfully tested function offloading mechanism by changing the defined threshold values. We use python 3.5 to implement the design explained in Section 4.2.3. A few of the supporting functions are explained in the rest of this section. These functions are essential for the implementation of the proposed architecture.

- ***threshold_ values()*** defines the threshold value for system resources. We use python *psutil*[4] package to determine the current CPU, memory and power utilisation at IoT, fog and edge devices. Our threshold value for memory and CPU consumption is 80% each and 30% for remaining battery.

- ***multicast_ sender()*** sends multicast requests over a defined multicast IP and port.

- ***whitelister()*** function takes a *Function bucket* in form python dictionary as with an input and removes a function name if condition satisfies.

- ***serverless_ executor()*** function is responsible for triggering the serverless function at each layer. It can be achieved by accessing an OpenFaaS function using the URL. This function also takes care of metrics evaluation, function offloading from one layer to another.

Our approach gives end-users flexibility of writing custom functions suitable to their requirements.

[2] https://learn.adafruit.com/dht/overview

[3] https://www.raspberrypi.org/downloads/raspbian/

[4] https://pypi.org/project/psutil/3.3.0/

Chapter 5

Evaluation

This chapter evaluates and analyses the proposed solution based on different evaluation criteria. In this chapter, We describe the evaluation criteria and their applicability to our solution.

5.1 Methodology

The architecture introduced in Section 4.1 is evaluated based on seven desired characteristics. We defined these characteristics considering several aspects of an ideal solution such as secure communication, work with limited system resources and heterogeneous devices. Additionally, we also evaluate function execution offloading by comparing the execution timeout in different conditions. The characteristics considered in this work are as detailed next:

5.1.1 Evaluation criteria

- **Ease of deployment:** The deployment of the proposed architecture should be smooth and easy. An ideal deployment minimizes manual installation efforts by automating the related procedure.

- **High availability:** The solution is expected to deploy across the number of IoT devices and multiple cloud computing layers such as fog, edge and cloud. It is extremely critical for the desired solution to guarantee essential services with minimum downtime.

- **Fault tolerance:** Fault tolerance is one of the key properties considering the vulnerable nature of IoT devices. IoT devices are more susceptible to component failure. The solution should trigger immediate recovery from any failure.

- **Device Heterogeneity:** The solution should effectively abstract the heterogeneity of IoT devices. Developers often like to use multiple programming languages depends on the scenario. Our solution should provide function runtime heterogeneity by allowing the development in desired programing language to developers.
- **Scalability:** We can not predict the rate of function execution in advance which leads to the requirement of a highly scalable solution that can scale up and down instantly based on the requirement.
- **Security and Privacy:** Security and privacy are arguably one the most critical requirement of any solution. An ideal architecture should extend the secure function execution environment to developers along with secure internal communication among internal components.

5.2 Analysis

This section presents the detailed analysis of the proposed solution from criteria defined in Section 5.1.1:

Ease of deployment

OpenFaaS is an easy platform to deploy in IoT networks. The only prerequite for OpenFaaS is Docker. It is feasible to deploy Docker on many devices such as Raspberry Pi and Arduino. A framework such as resin.io[1] make it simple to deploy docker across the heterogeneous IoT networks [30]. Any device with Docker and Internet connection can be used to deploy the OpenFaaS framework [27]. The platform has specific installation instruction for ARM-based devices. The whole OpenFaaS installation at Raspberry Pi 3 takes approximately 55MB on the device's memory.

High availability

Devices at the fog and edge layers are divided into more than one Docker swarms. Docker swarm based deployment provides two levels of high availability. The first level of high-availability comes from the multiple nodes in the Docker swarms. More than one number of swarms are responsible for the second level of high- availability. For instance, if one node is not responding within a swarm, the manager delegates the task to another node. Similarly, if

[1] https://docs.resin.io/introduction/

one of the swarm is not responding, the offloading mechanism will forward the task to another available swarm. Figure 5.1 shows the high availability of the proposed architecture. In addition to the above mentioned high-availability

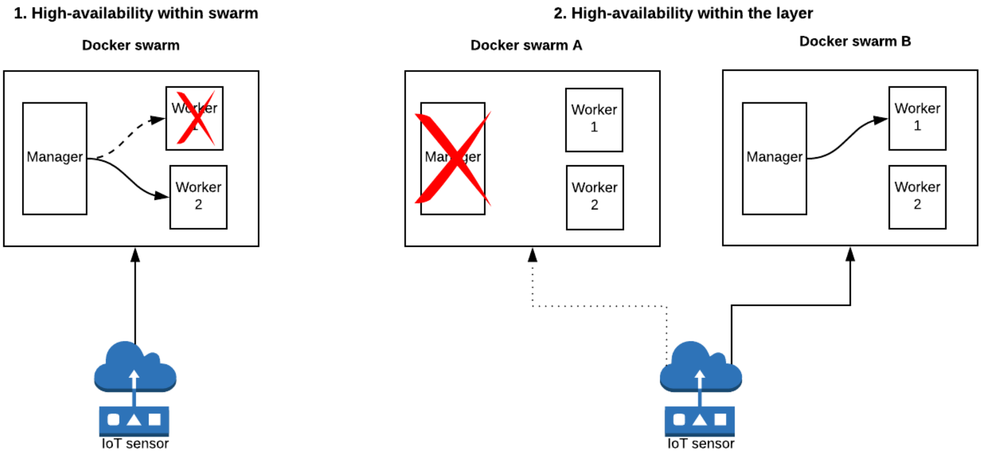

Figure 5.1: High availability within the layer

scenarios, our architecture has the capability of offloading the function execution to the next layer of the current layer is not available. Figure 5.2 shows the high available multi-layered architecture. High availability at the IoT-gateway entirely depends upon the presence of the redundant gateway. Cloud providers implement the cloud layer's high-availability.

Fault tolerance

The proposed architecture is susceptible to failure like any other system. However, Docker swarm-based deployment makes sure to initiate new container upon failure to maintain the desired state. Docker recommends using an odd number of swarm managers to handle the failure of one or more manager nodes. An odd number of swarm manager nodes allows a Docker swarm to remain functional in case of the failure. Moreover, fault tolerance can be increased by distributing swarm manager nodes to different IoT devices. In such scenario, Failure of an IoT device does not fail multiple manager nodes, i.e. swarm failure.

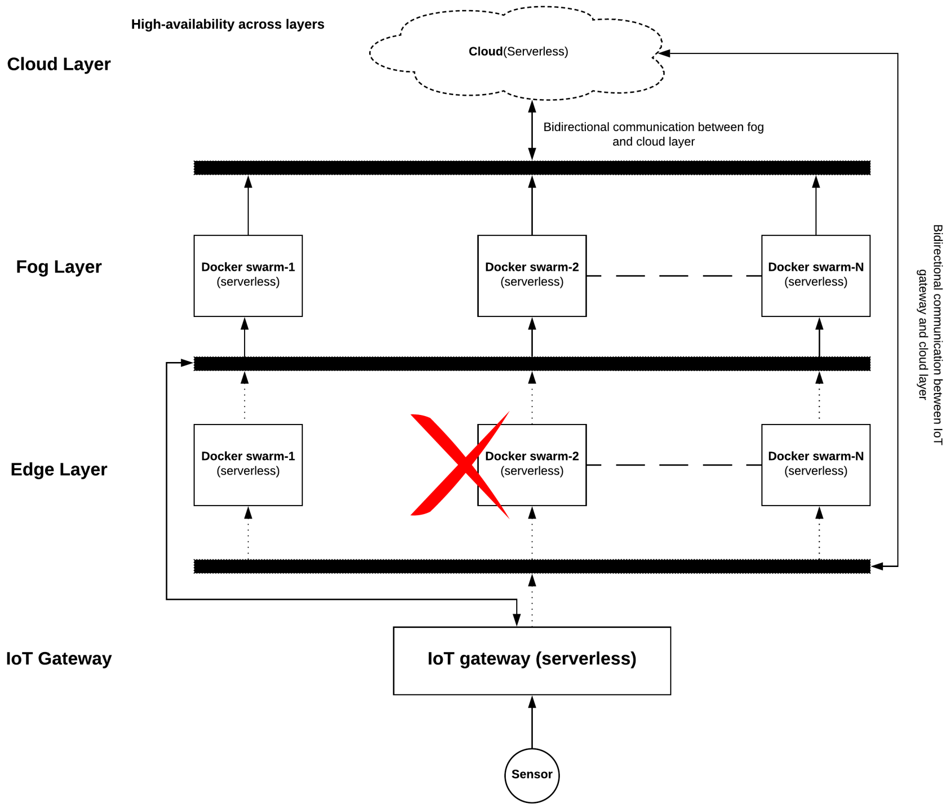
High-availability across layers
Cloud Layer
Cloud(Serverless)
Bidirectional communication between fog and cloud layer
Fog Layer
Docker swarm-1 (serverless)
Docker swarm-2 (serverless)
Docker swarm-N (serverless)
Bidirectional communication between IoT gateway and cloud layer
Edge Layer
Docker swarm-1 (serverless)
Docker swarm-2 (serverless)
Docker swarm-N (serverless)
IoT Gateway
IoT gateway (serverless)
Sensor

Figure 5.2: High availability across layers

Device Heterogeneity

The architecture supports heterogenous IoT devices that have different characteristics such as processor architecture, resources, memory and physical location. However, Distributed computing framework can be deployed for heterogeneous devices using Docker containers [31]. It is also feasible to create a Docker swarm over multiple IoT devices to deploy OpenFaaS. The proposed solution uses OpenFaaS as the underlying serverless platform that supports many runtime environments such as Python, Javascript, Nodejs and Docker. Furthermore, the solution supports both synchronous and asynchronous functions.

Scalability

The proposed architecture uses the auto-scaling feature implemented in OpenFaaS. The openfaas architecture includes function auto-scaling by scaling up or down the function deployment on-demand. OpenFaaS accomplishes the auto-scaling using Prometheus matrices. OpenFaaS has a defined value of min and max replicas. By default, the minimum possible number of replicas is 1 whereas, one can launch maximum 20 replicas while scaling up. It is also possible to disable auto-scaling by setting same value for min and max replicas or setting *com.openfaas.scale.factor=0*. The scaling factor defines the number of replicas initiated upon the alarm is generated and is set to 20% by default. We have explained auto-scaling in detail in Section 3.3.2.

Security and Privacy

OpenFaaS allows us to implement basic authentication (username/password authentication) while deploying functions. Furthermore, a certificate-based HTTP authentication can also be enabled for the functions [32]. However, the security mechanism also depends upon the capability of the IoT device. The proposed solution also incorporates the built-in security features of the Docker implementation.

5.2.1 Offloading Analysis

```
from pymongo import MongoClient
from datetime import datetime
import json
import time

def handle(req):
    client = MongoClient("mongodb://10.10.10.10:27017")
    db = client.test
    json_req = json.loads(req)
    db_entry = {"Humidity": json_req["Humidity"],
                  "Temperature_farenheit":json_req["
                      Temperature_farenheit"],
              "Temperature_celsius": json_req["
                 Temperature_celsius"],
              "Latitude": json_req["Latitude"],
              "Longitude": json_req["Longitude"],
              "Time": datetime.now()}
    db.sensordata.insert(db_entry)
    return {"statusCode": "200"}
```

Listing 5.1: Mongowriter OpenFaaS function

We evaluated our proposed solution by analysing the function execution offloading at the different layers. The *exec_ timeout* for each layer are presented in the Table 4.1. We used the setup described in section 4.3 that consisted of Raspberry Pi 3 as part of Docker swarm at IoT-Gateway and edge layer. We used *dth 11*, Ubuntu 16.04 LTS machine and IBM cloud functions for sensor, fog and cloud layers respectively. Listing 5.1 consists of a synchronous OpenFaaS function that takes values from the sensor and writes them to the MongoDB database. This function was deployed across all layers. We performed 10 iterations for each of the cases mentioned in Table 5.2 and 5.3 to calculate the average value and the standard deviation.

Case	**Executed at**	**1st execution(time taken in seconds)**	**2nd execution(time taken in seconds)**
IoT-Gateway timeout	Edge Layer	22.1764	2.1617
IoT-Gateway and Edge timeout	Fog Layer	58.3487	1.8188
IoT-Gateway, Edge and Fog Timeout	Cloud Layer	99.8052	1.8126

Table 5.1: Function execution time in case of threshold timeout

Table 5.2 shows the function execution time in case of threshold timeout at different layers. The function takes higher time for the first execution as it runs for whole *exec_ timeout* at each layer. Our solution also creates a bucket entry at each layer that allows the function to skip *exec_ timeout* and results in low function execution time values during second execution. we have a low value of standard deviation in each case as he time taken during each iteration does not differ significantly. Figure 5.3 demonstrates the difference in the time taken for the first and second execution in each layer.

Case	**Executed at**	**First execution(standard deviation in seconds)**	**Second execution(standard deviation in seconds)**
IoT-Gateway timeout	Edge Layer	0.0449	0.0469
IoT-Gateway and Edge timeout	Fog Layer	0.2801	0.0755
IoT-Gateway, Edge and Fog Timeout	Cloud Layer	0.1179	0.1341

Table 5.2: standard deviation in case of threshold timeout

Case	**Executed at**	**Execution time (in seconds)**	**Standard deviation (in seconds)**
All layers are functional	IoT-Gateway	4.5269	0.0575
IoT gateway is unavailable	Edge layer	2.1860	0.0417
IoT-Gateway and Edge are unavailable	Fog layer	12.3600	0.2232
IoT-Gateway, Edge and Fog are unavailable	Cloud layer	7.3726	0.2833

Table 5.3: Function execution time in case of unavailability of the layers

Table 5.3 contains the time taken for the function execution and standard deviation when lower layers are unavailable. Our solution checks the availability of the devices at each layer before offloading function execution to the

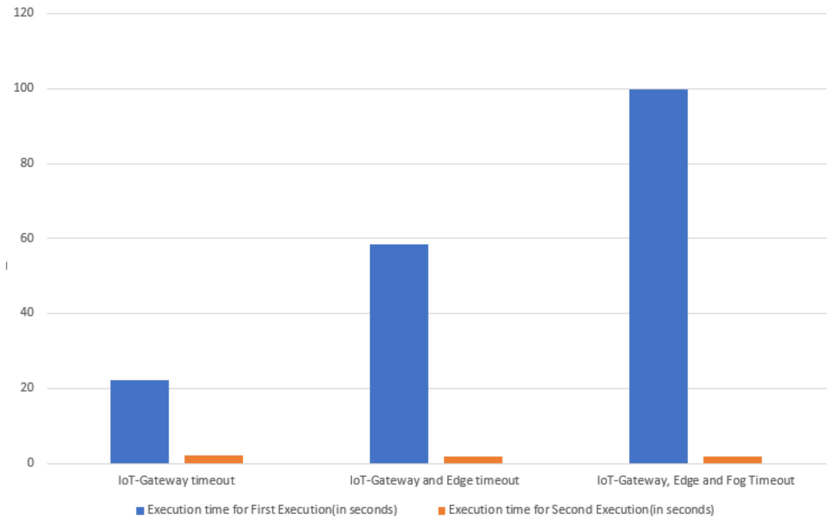

Figure 5.3: Comparison of time taken during first and second execution on timeout

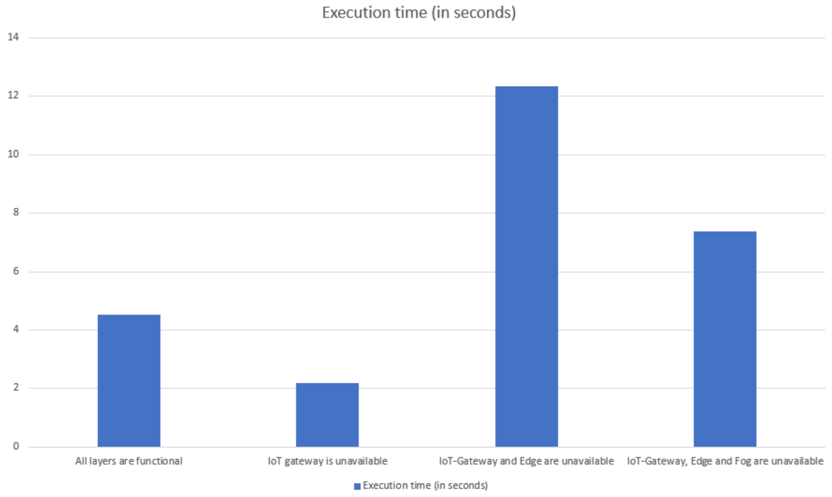

Figure 5.4: Function execution time in case of unavailability of the layers

layer above. As previously discussed, the primary reason for comparatively higher values of execution time is due to the time taken in the second execution in case of timeout. Figure 5.4 shows the time taken at different layers upon function execution offloading. Our proposed solution demonstrate the successful deployment of a serverless platform for the Internet of Things.

Chapter 6

Conclusion

This work demonstrated an implementation of serverless computing in an IoT network. We considered the resource constrained nature and heterogeneity of IoT devices in our solution. Devices with different attributes such as processing capability, memory, battery, sensors were set up in a cluster using a Docker-based orchestration mechanism called Docker swarm. Our work also inherits some beneficial properties of Docker swarm such as fault-tolerance and high availability.

In particular, we constructed a multi-layered architecture for our solution with layers such as IoT-Gateway, fog, edge and cloud layer. Available open-source serverless platforms such as Apache OpenWhisk, OpenFaaS, Kubeless and Fission were surveyed. Upon careful evaluation, we selected OpenFaaS at IoT-Gateway, fog, edge layers due to its ease of deployment on arm architecture-based devices and flexibility. OpenFaaS was also preferred over Kubeless as latter can only be deployed using kubernetes. The architecture used IBM cloud functions as cloud layer due to its cost-effectiveness. IBM cloud functions is an IBM proprietary version of Apache OpenWhisk.

An algorithm was designed for the function execution offloading among different layers of our architecture. The maximum function execution time was defined at each layer depending upon the resources availability. The offloading decision was made based on the availability of active nodes in the Docker swarm. Functions were written in python and deployed at each layer by using the OpenFaaS command line interface.

The solution later evaluated on various factors such as ease of deployment, high availability, fault-tolerance, device and function heterogeneity, security and privacy. We also compare the function execution time at various layers of proposed solution. We showed that it is possible to successfully deploy a serverless platform on IoT devices and use it to perform various tasks with the help of serverless functions. The concepts presented in this work can be expanded, and further research can be conducted. In this context, some interesting directions for future work are the following:

1. Defining the trigger alarms for function execution offloading specific to our architecture. At the moment, the architecture uses features built into to offload a function from one layer to another.

2. An implementation using kubernetes as this solution heavily relies on the Docker swarm.

3. Implementing a logging server along with monitoring functionality for the proposed architecture. This solution relies on the OpenFaaS functionalities that do not take offloading mechanism into account.

4. An authentication mechanism for function deployment and execution separated from OpenFaaS authentication.

5. Implementation of secure offloading of function execution from one layer to another.

Bibliography

[1] Janet Ellen Abbate. From arpanet to internet: A history of arpa -sponsored computer networks, 1966–1988, 1994. https://repository.upenn.edu/dissertations/AAI9503730/. Accessed 31 July 2018.

[2] A. L. Russell. The internet that wasn't. *IEEE Spectrum*, 50(8):39–43, August 2013.

[3] Cisco. Cisco visual networking index, 2017. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html. Accessed 31 July 2018.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[5] Yury Izrailevsky. Completing the netflix cloud migration, 2016. Completing the Netflix Cloud Migration : https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration. Accessed 31 July 2018.

[6] Cisco. Internet of things, 2016. Internet of Things - CISCO : https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf. Accessed 24 June 2018.

[7] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013.

[8] S. Deshmukh and R. Shah. Computation offloading frameworks in mobile cloud computing : a survey. In *2016 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC)*, pages 1–5, March 2016.

[9] S. Singh and N. Singh. Internet of things (iot): Security challenges, business opportunities amp; reference architecture for e-commerce. In *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 1577–1581, Oct 2015.

[10] M. H. Miraz, M. Ali, P. S. Excell, and R. Picking. A review on internet of things (iot), internet of everything (ioe) and internet of nano things (iont). In *2015 Internet Technologies and Applications (ITA)*, pages 219–224, Sept 2015.

[11] University of Maryland website. Don't we all need arms, 2016. WWW page of the cs.umd.edu: `https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/arm/`. Accessed 04 Aug 2018.

[12] A. B. S., H. M.J., J. P. Martin, S. Cherian, and Y. Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *2014 Fourth International Conference on Advances in Computing and Communications*, pages 247–250, Aug 2014.

[13] Docker.com. What is docker. WWW page of the www.docker.com: `https://www.docker.com/what-docker`.

[14] A. Tosatto, P. Ruiu, and A. Attanasio. Container-based orchestration in cloud: State of the art and challenges. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 70–75, July 2015.

[15] Docker.com. Manage swarm security with public key infrastructure (pki). WWW page of the www.Docker.com: `https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/`. Accessed 11 May 2018.

[16] Docker.com. Administer and maintain a swarm of docker engines. WWW page of the www.Docker.com: `https://docs.docker.com/engine/swarm/admin_guide/`. Accessed 11 May 2018.

[17] Docker.com. Raft consensus in swarm mode. WWW page of the www.Docker.com: `https://docs.docker.com/engine/swarm/raft/`. Accessed 11 May 2018.

[18] Docker.com. Manage swarm service networks. WWW page of the www.Docker.com: `https://docs.docker.com/v17.09/engine/swarm/networking/`. Accessed 11 May 2018.

[19] Mike Roberts. Serverless architectures, 2016. WWW page of the martinfowler.com: `https://martinfowler.com/articles/serverless.html`. Accessed 14 May 2018.

[20] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 162–169, Dec 2017.

[21] Openwhisk. Git repository of Apache OpenWhisk project: `https://github.com/apache/incubator-openwhisk`. Accessed 11 May 2018.

[22] Openwhisk cli. Git repository of Apache OpenWhisk CLI: `https://github.com/apache/incubator-openwhisk-cli`. Accessed 21 May 2018.

[23] Michael Mendenhall and Buell Duncan. Bluemix is now ibm cloud, 2017. WWW page of the IBM cloud blogs: `https://www.ibm.com/blogs/bluemix/2017/10/bluemix-is-now-ibm-cloud/`. Accessed 11 May 2018.

[24] Openfaas. Git repository of OpenFaaS: `https://github.com/openfaas/faas`. Accessed 11 May 2018.

[25] Deployment guide for docker swarm. Git repository of OpenFaaS: `https://docs.openfaas.com/deployment/docker-swarm/`. Accessed 11 May 2018.

[26] Alex Ellis. Alert rules, 2017. Git repository of OpenFaaS: `https://github.com/openfaas/faas/blob/master/prometheus/alert.rules.yml`. Accessed 22 June 2018.

[27] Docker comes to raspberry pi. WWW Raspberrypi blog: `https://www.raspberrypi.org/blog/docker-comes-to-raspberry-pi/`. Accessed 14 May 2018.

[28] Kubernetes on (vanilla) raspbian lite. Git repository of alexellis: `https://gist.github.com/alexellis/fdbc90de7691a1b9edb545c17da2d975`. Accessed 14 May 2018.

[29] Alex Ellis. Your serverless raspberry pi cluster with docker, 2017. WWW Alex Ellis' Blog: `https://blog.alexellis.io/your-serverless-raspberry-pi-cluster/`. Accessed 14 May 2018.

[30] Y. Gao, H. Wang, and X. Huang. Applying docker swarm cluster into software defined internet of things. In *2016 8th International Conference on Information Technology in Medicine and Education (ITME)*, pages 445–449, Dec 2016.

[31] Daniel JosÃ© Bruzual Balzan. Distributed computing framework based on software containers for heterogeneous embedded devices. Master's thesis, Department of Computer Science and Engineering, Aalto University School of Science and Technology, Espoo, Finland, 2017. `https://aaltodoc.aalto.fi/handle/123456789/28568?show=full`.

[32] Alex Ellis. Lock-down openfaas for the public internet, 2017. WWW Alex Ellis' Blog: `https://blog.alexellis.io/lock-down-openfaas/`. Accessed 29 May 2018.