POLITECNICO DI MILANO
Department of Electronics, Informatics and Bioengineering
M.Sc. programme in Computer Science and Engineering

Deployment of Container Orchestration and Function-as-a-Service functions with Node Selection Method on Hybrid Cluster Architecture

Advisor: Prof. Pierluigi Plebani

Master Thesis of:
Alberto Marini
862838

Academic Year 2019-2020

Deployment of Container Orchestration and Function-as-a-Service functions with Node Selection Method on Hybrid Cluster Architecture

Alberto Marini

Academic Year 2019-2020

στον πατέρα μου
καί τη μητέρα μου
καί αγαπημένους άνθρωπουσ μου

Abstract

The stateless containerized Function-as-a-Service (FaaS) cloud computing model is increasing of paramount importance as it allows developers to create, execute and manage application packages as functions, without dealing with their own infrastructure.

The scope of this thesis is to verify the possibility of creating a hybrid architecture with the open-source software currently available, that is possible to perform FaaS functions, by selecting the desired node.

This thesis shows the development of a heterogeneous cluster with VPS on the cloud and Raspberry Pi. The chosen Container Orchestration platform is k3s, and as regards the framework for the Function-as-a-Service functions, the choice is OpenFaaS. The way that to deploy FaaS applications on OpenFaaS by selecting the desired node and making the serverless function compatible with different architectures is also presented.

Sommario

Il modello di cloud computing Function-as-a-Service (FaaS) basato su eventi ed eseguito in container stateless é sempre più di fondamentale importanza poiché consente agli sviluppatori di creare, eseguire e gestire i pacchetti applicativi come funzioni, senza doversi occupare della propria infrastruttura.

L'obiettivo in questa tesi é quello di verificare la possibilità di creare un'architettura ibrida con il software open-source ad oggi disponibile al cui interno é possibile eseguire funzioni FaaS in modo flessibile, selezionando il nodo.

In questa tesi viene mostrato lo sviluppo di un cluster eterogeneo con VPS su cloud e Raspberry Pi. La piattaforma di orchestrazione dei container scelta é k3s, mentre per quanto concerne il framework per le funzioni Function-as-a-Service, la scelta effettuata é quella di OpenFaaS. Infine, viene mostrato come effettuare il deploy delle applicazioni FaaS su OpenFaaS selezionando il nodo desiderato e rendere la funzione serverless compatibile con diverse architetture.

Contents

Abstract **i**

Sommario **iii**

Introduction **1**

1 FaaS & Container Orchestration **3**
1.1 Introduction . . . 3
1.2 FaaS platforms . . . 4
1.2.1 Apache OpenWhisk . . . 4
1.2.2 OpenFaaS . . . 5
1.2.2.1 The API Gateway . . . 6
1.3 Container Orchestration . . . 7
1.3.1 Docker & Docker Swarm . . . 7
1.3.1.1 Docker Engine . . . 7
1.3.1.2 Docker orchestration . . . 8
1.3.2 Kubernetes . . . 9
1.3.3 k3s . . . 10

2 Hybrid cluster **13**
2.1 Introduction . . . 13
2.2 Containers on hybrid environment . . . 13
2.3 FaaS deployment . . . 14
2.4 Implementation . . . 14

3 Cluster configuration **17**
3.1 Introduction . . . 17
3.2 Architecture . . . 17
3.3 Environment setup . . . 19
3.3.1 k3s cluster deployment . . . 19
3.3.1.1 OpenFaaS deployment . . . 21

3.3.1.2 OpenFaaS Functions 23
3.3.1.3 Node selection 23

4 Concluding remarks **25**
4.1 Future directions 25

A Raspberry Pi & k3s code scripts **27**

B OpenFaaS code scripts **31**

List of Figures

1.1 Architectural View of Apache OpenWhisk [The19] 4
1.2 OpenFaaS structure [Ale19] 5
1.3 Conceputal OpenFaaS architecture when Kubernetes is used as the orchestration provider [Ope19] 6
1.4 Docker Engine components flow [Doc19a] 8
1.5 Docker Swarm components[Doc19b] 8
1.6 Kubernetes node overview [Kub19b] 9
1.7 Kubernetes Master and Worker node components [Kub19a] . 10
1.8 k3s Server & k3s Agent [Ran19] 12

2.1 Hybrid Cluster Diagram 14
2.2 Deployment OpenFaaS function on a selected Worker node . . 15

3.1 k3s Single Server Architecture [k3s20] 18
3.2 k3s High Availability Architecture [k3s20] 18
3.3 List of Nodes of the Hybrid Cluster 20
3.4 Hybrid Cluster with Labels 20
3.5 OpenFaaS Services 21
3.6 OpenFaaS API Portal 22

Introduction

Function-as-a-Service (FaaS) and Container Orchestration platforms are becoming increasingly important tools for deploying services over the network, thanks to the flexibility and reliability. One of the main benefits is the optimization and performance of an application. Container Orchestration platforms are very popular nowadays among applications in multiple fields as cloud computing and so on. The implementation of the microservices logic has been adopted widely for various reasons, starting from the isolation of each application running as a Container. As a further step, Function-as-a-Service (FaaS) allows developers to write applications without handling the system resources. In this context, it is possible to create a short-lived container which can be triggered when needed and also the computation overload is purely on demand.

The goal of this thesis is to show that it is possible to have a deployment of serverless functions, with the selectivity of which node to use, on a hybrid FaaS cluster, using a Container Orchestration platform to orchestrate the entire network. The environment consists of a cluster with machines based on cloud and a physical device such as Raspberry Pi. We will present the capabilities and limitations, the consequent choices made during the configuration of the cluster and we will explain how to implement everything.

Structure of the document

The thesis is structured in the following way:

- **Chapter 1** provides an introduction to the principal FaaS and Container Orchestration platforms that we analyzed and used for reaching the main goals.
- **Chapter 2** discusses the chosen platforms for the hybrid cluster and how can be implemented.

- **Chapter 3** covers the architecture of the hybrid cluster environment, starting from the available resources, moving to the configuration choices, and regarding the implementation, from k3s to OpenFaaS function deployment.
- **Chapter 4** summarizes the results of this thesis, giving final reasoning and introducing the next steps for further developments.

Chapter 1

FaaS & Container Orchestration

1.1 Introduction

Function-as-a-Service (FaaS) is a type of computing known also as a serverless and event-driven, method of using cloud technology to enable higher efficiency in computer workflows and processes.

The main benefit of FaaS is the capability of offering cloud services using the required resources. Moreover, FaaS helps to concentrate only on applicative function, without considering the server provisioning, operative system, and other aspects.

FaaS platforms do not require coding to a specific framework or library. In fact, FaaS functions are like regular applications for language and environment. The only limit is the significant architectural restriction, peculiarly for state and execution duration.

In this Chapter, we are going to consider the main FaaS and Container Orchestration services. All the platforms discussed in this thesis are open-source and available on GitHub.

1.2 FaaS platforms

Many Function-as-a-Service (FaaS) platforms have been created to satisfy the different needs for deployment. Up to now, the bigger part of FaaS platforms are open-source or to be paid. Especially in the open-source world it is quite important that behind a Faas platform there is an active community for supporting, solving problems and releasing new versions with updated functionalities.

In this section, we consider Apache OpenWhisk and OpenFaaS.

1.2.1 Apache OpenWhisk

Apache OpenWhisk is an open-source distributed Serverless platform. OpenWhisk manages the infrastructure, servers and scaling using Docker containers. The OpenWhisk platform is based on Scala and supports different programming languages like .Net, Go, Java, JavaScript, PHP, Python, Ruby, and Swift.

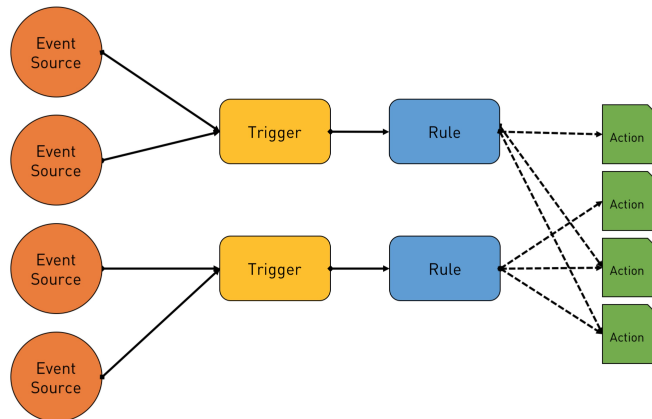

Figure 1.1: Architectural View of Apache OpenWhisk [The19]

The architectural scheme, shown in Figure 1.1 includes the following concepts

- Event Source: passes the set of parameters that are essential for the invocation.
- Trigger: endpoints that are explicitly called by event sources such as databases, stream processing engines, file systems, and line-of-business applications.

- Rule: acts as the connection between Triggers and Actions by creating a loosely coupled association between them.
- Action: represents the standalone functions, completely autonomous and independent of the event sources.

Apache OpenWhisk can be used with an account on IBM BlueMix, a Vagrant Machine or with Docker on Ubuntu. We no longer consider OpenWhisk due to issues faced in setup and its minimal dependence on Kubernetes for container orchestration.

1.2.2 OpenFaaS

OpenFaaS is an open-source Functions-as-a-Service framework created by Alex Ellis, available on GitHub [1] under the MIT license. OpenFaaS is built with a focus on ease of use, simplicity, portability, and openness. As shown in Figure 1.2, OpenFaaS has two key components: an API Gateway and a functioning watchdog. Prometheus, instead, is used for recording metrics. It also supports multiple container orchestrators like Kubernetes and Docker Swarm.

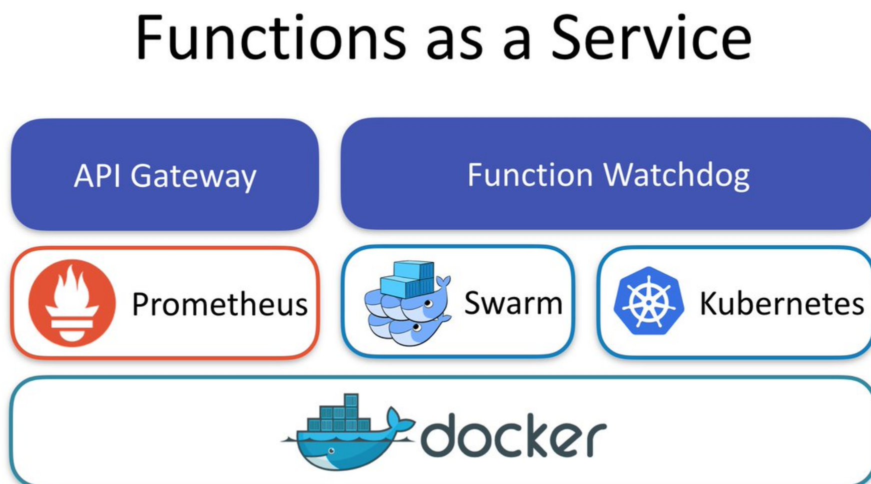

Figure 1.2: OpenFaaS structure [Ale19]

OpenFaaS is growing rapidly in the Open-Source Serverless Frameworks field, with continuous updates and new functionalities.

[1] https://github.com/openfaas/faas

1.2.2.1 The API Gateway

One of the main benefits of OpenFaaS is the capability of allowing all developers to write functions in any programming language like C#, Go, NodeJS, Python, and Ruby.

OpenFaaS supports two types of function triggers: HTTP and event-based. In addition, OpenFaaS can be integrated with Apache Kafka, an open-source distributed streaming platform for building real-time data pipelines and stream processing applications.

The API Gateway provides a route into the deployed functions and collects metrics through Prometheus. The Gateway has a User Interface for deploying functions and invoicing them. Due to the API Gateway, it is possible to scale up from zero using also auto-scaling functionalities via AlertManager and Prometheus.

Figure 1.3, it represents the conceptual architecture when Kubernetes is used as the orchestration provider.

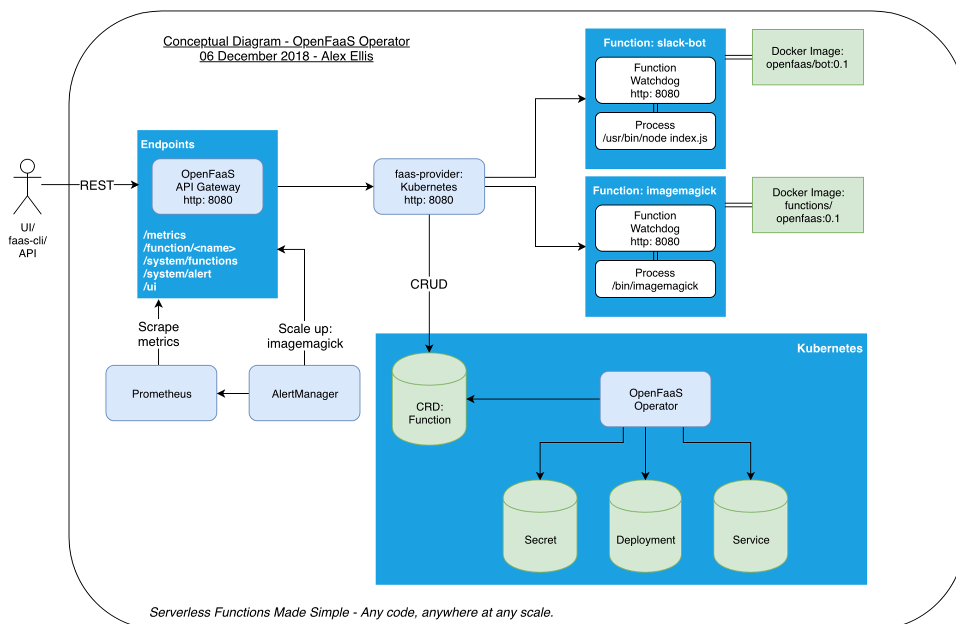

Figure 1.3: Conceptual OpenFaaS architecture when Kubernetes is used as the orchestration provider [Ope19]

1.3 Container Orchestration

Container Orchestration platforms has changed the way of software organizations build, ship, and maintain applications. A container is run by using Container Orchestration platforms and it has its own filesystem, CPU, memory, process space, and more. A container is decoupled by the underlying infrastructure, which means that it is portable across cloud and OS distribution. Due to Container Orchestration platforms, it is possible to handle the automation of all aspects of coordinating and managing containers. The main benefits of Container Orchestration are increased portability, simple and fast deployment, and improved security.

In this section, we are going to consider Docker, Kubernetes, and k3s. The latter one is a platform based on Kubernetes, but developed with a different approach to solving problems during the deployment on heterogeneous devices.

1.3.1 Docker & Docker Swarm

Docker is an open-source project with the main focus of automatizing application deployment. By using Docker, the abstraction level allows us to package and run an application in the isolated environment called container. In particular, containers are lightweight because they do not require the extra load of a hypervisor, and they can run directly within the host machine's kernel.

1.3.1.1 Docker Engine

The Docker Engine is a client-server application with three major components:

- A server, a program called the daemon process, also known as `dockerd`.
- REST API useful for talking to the daemon and instruct it what to do.
- A command-line interface (CLI) client, also known as docker command.

The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting language or direct CLI commands. Many other Docker applications use the underlying API and CLI.

The daemon creates and manages Docker objects, such as images, containers, networks, and volumes as shown in Figure 1.4.

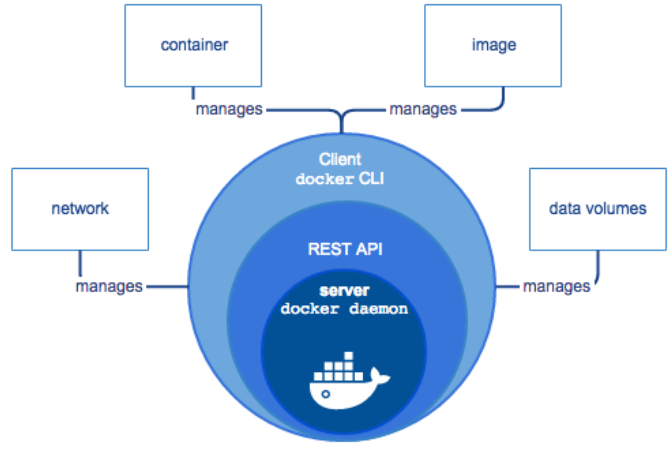

Figure 1.4: Docker Engine components flow [Doc19a]

1.3.1.2 Docker orchestration

Starting from Docker Engine 1.12, and newer version, has been introduced the Swarm mode, a clustering and scheduling tool for Docker containers.

Docker Swarm uses scheduling capabilities to ensure that there are sufficient resources for all the distributed containers. Swarm assigns containers to underlying nodes and optimizes resources by automatically scheduling container workloads to run on the most appropriate host. The main benefit provided by Swarm mode is about being sure that each container has been launched on a system satisfying the resources requested, while maintaining high-performance levels and optimal efficiency.

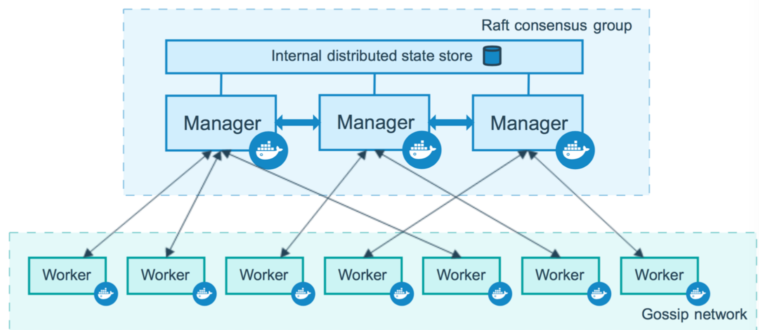

Figure 1.5: Docker Swarm components[Doc19b]

1.3.2 Kubernetes

Kubernetes is a container orchestration platform. Kubernetes automates the process of scaling, managing, updating and removing containers, and makes use of various concepts and abstractions.

The main benefit of Kubernetes is the possibility of allowing the automation of container provisioning, networking, load-balancing, security and scaling across all the nodes.

It allows users to run containers across multiple compute nodes. A node can be a virtual machine or a bare-metal server). Once Kubernetes has taken control over a cluster of nodes, containers can be turned on or off at any given time.

There are two basic concepts about a Kubernetes cluster shown in Figure 1.6. The first is the node, a common term for VMs and/or bare-metal servers that Kubernetes manages. The second term is pod, which is a basic unit of deployment in Kubernetes. A pod is a collection of related Docker containers that need to coexist. Each pod is assigned with a unique IP address within the cluster, allowing the application to use ports without conflict. Pods are created and destroyed on nodes as needed to conform to the state specified in the pod definition.

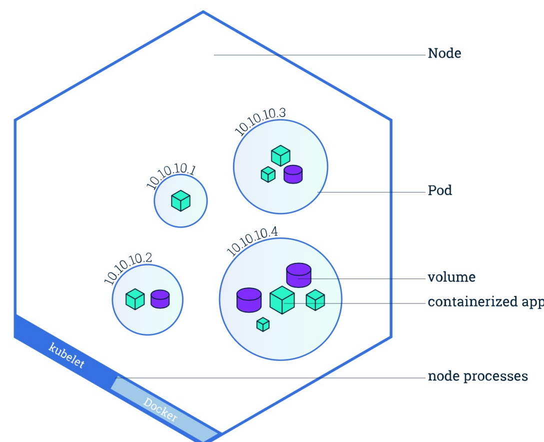

Figure 1.6: Kubernetes node overview [Kub19b]

Coming back to the nodes, there are two types of nodes as shown in Figure 1.7. One is the Master Node, the main access point from which administrators and other users interact with the cluster to manage the scheduling and deployment of containers. It controls the scheduling of pods across various Worker nodes, where the application runs. The master node's job is to

make sure that the desired state of the cluster is maintained. The Kubernetes Worker node runs an agent process named `kubelet` that is responsible for managing the state of the node and it collects performance and health information from nodes, pods, and containers.

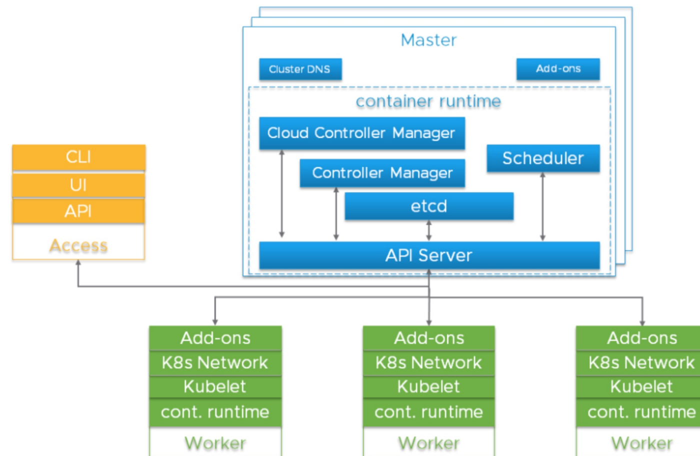

Figure 1.7: Kubernetes Master and Worker node components [Kub19a]

Moreover, Kubernetes provides also other abstraction functionalities like:

- Namespaces: it allows to create virtual clusters on top of a physical cluster.
- Labels: key/value pairs that you can assign to pods and other objects in Kubernetes.

Finally, one of the Kubernetes components that helps to keep an eye on the container deployment is Dashboard, a web-based UI from which is possible to deploy and troubleshoot apps and manage cluster resources

1.3.3 k3s

k3s is a lightweight Kubernetes version developed by Rancher Labs. The reason why k3s have been developed is the necessity of creating an optimized version for ARM64 and ARMv7 processor, capable to provide all the most important tools that are developed inside Kubernetes.

The k3s binary, available on GitHub [2], is less than 40MB. It requires only 512MB of RAM to run on a device like Raspberry Pi. Inside k3s binary

[2]https://github.com/rancher/k3s

are available all the low-level components required like containers, runc and kubectl.

The minimum system requirements for k3s are the following:

- Linux 3.10+
- 512 MB of RAM per server
- 75 MB of RAM per node
- 200 MB of disk space
- x86_64, ARMv7, ARM64

Looking closer to k3s, Rancher Labs removed legacy, alpha, non-default features and most in-tree plugins (cloud providers and storage plugins) which can be replaced without tree addons. Moreover, as default storage mechanism has been added sqlite3. etcd3 is available, but not the default. Everything is wrapped in simple launcher bash script A.2 in Appendix A that handles TLS and options in a short string.

The main k3s package required dependencies are the followings:

- containerd: a container runtime that can manage a complete container lifecycle.
- Flannel: one of the simplest networking interface that can create another flat network which runs above the host network, called *overlay* network.
- CoreDNS: it is a flexible, extensible DNS server that can server as the k3s cluster DNS.
- CNI: k3s supports container network interface, a standard created to configure container networking when containers are created or destroyed.
- Host utilities (iptables, socat, etc)

Figure 1.8 shows the relation between k3s Server and k3s Agent. In Kubernetes terminology, Server and Agent refer respectively to Master and Node. k3s bundles the Kubernetes components into combined processes that are presented as a simple server and agent model. Running k3s server will

start the Kubernetes server and automatically register the localhost as an agent. By default, k3s installs on the same machine server and agent, but it can be disabled an installed separately with a special flag as shown in bash script A.3 in Appendix A.

In particular, we can see that the network connection is controlled by the Tunnel Proxy component, instead, the k3s Server integrates the Scheduler and Controller Manager processes.

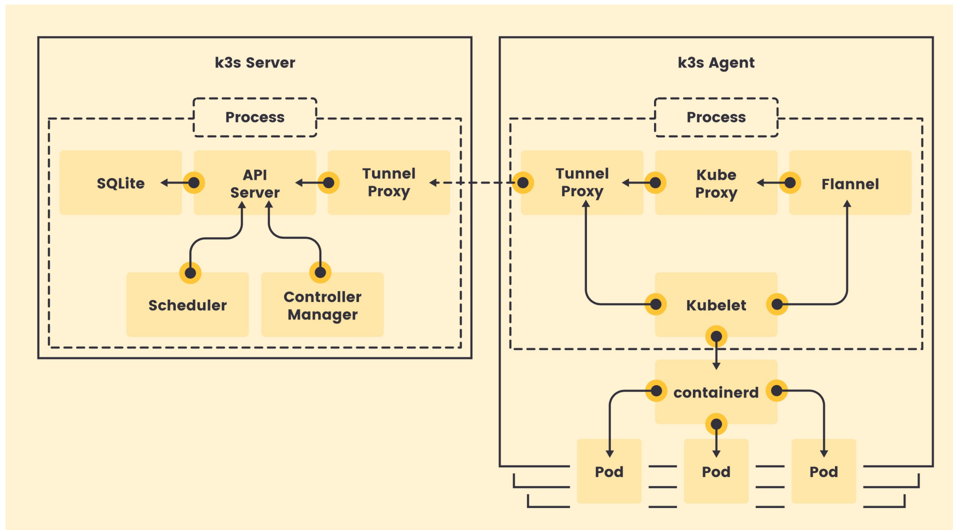

Figure 1.8: k3s Server & k3s Agent [Ran19]

By using k3s is possible to create a production cluster with heterogeneous devices. The support to x86_64, ARMv7, ARM64 is useful for solving the compatibility with different devices, creating a cluster network with a master node and deploying tasks over the network, targeting selected devices.

Chapter 2

Hybrid cluster

2.1 Introduction

This Chapter extends the Chapter 1 analyzing the characteristics of the chosen Container and FaaS platforms, and why they are the right choice for our cluster.

We verify how it is possible having a hybrid architecture with today's available open-source frameworks and the possibility to deploy FaaS functions, flexibly, selecting the desired node.

2.2 Containers on hybrid environment

Container Orchestration on the hybrid environment requires particular attention about performances and resources. The container virtualization requires a level of abstraction that is quite complex for a heterogeneous network of devices with different hardware.

For these reasons, a lightweight Kubernetes distribution as Rancher Labs' k3s has been chosen. It provides most of Kubernetes components for network management and container deployments. k3s is compatible with Raspberry Pi and runs using a low amount of resources thanks to the reduced dependencies.

In fact, in our hybrid cluster shown in Figure 2.1 we have three VM's based on cloud and two Raspberry Pi. There are multiple reasons why choosing a cloud platform, starting from the availability to the reduced costs in running a VM in an always-on environment, with backup and recovery options. One of the cloud virtual machines is the main Server, instead, the others are the nodes, called Workers in k3s.

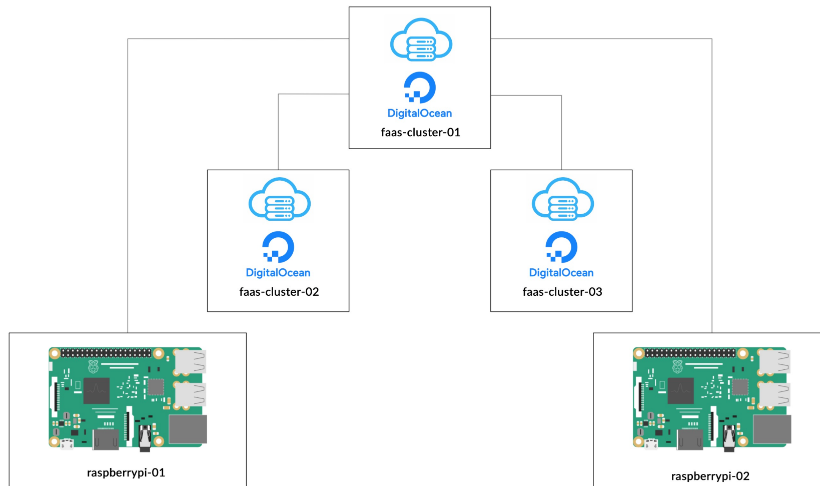

Figure 2.1: Hybrid Cluster Diagram

2.3 FaaS deployment

Function-as-a-Service is a framework for building serverless functions on top of containers. A serverless function is short-lived, not a daemon, not stateful and it executes in a few seconds.

We decided to deploy on the hybrid cluster the OpenFaaS framework that is compatible with Kubernetes and Docker. Moreover, OpenFaaS can be expanded adding functionalities like Grafana dashboards[1] and many other tools helpful for handling the deployment of OpenFaaS functions.

2.4 Implementation

We identified OpenFaaS and k3s to enable our goal of deploying FaaS function on a hybrid cluster.

Provided a cluster of machines, which can comprise cloud VM's and other heterogeneous devices such as Raspberry Pi, we bootstrap the k3s server on the master node. Then, we connect each worker node to the master by deploying k3s and adding the relative master node token as shown in code A.6.

For the Raspberry Pi's Worker nodes it is necessary to install the Operative System Raspberry OS on their microSD card and proceed with k3s

[1] https://grafana.com/grafana/dashboards/3526

installation. Then, we use an abstraction functionality of Kubernetes such as labels, to rename each node. The Label is a key/value set that can be assigned to every object in Kubernetes. After the k3s cluster has been set, each node has been labeled uniquely, on the Master node is possible to deploy OpenFaaS and the relative CLI with the code B.1.

By deploying OpenFaaS, all the services such as AlertManager, FaaS-idles, Gateway, Nats, Prometheus, and Queue-Worker are distributed over the hybrid cluster. This solution's quite straightforward to guarantee that the serverless platform is always available. If a Worker node goes offline, the lost OpenFaaS services will be recreated on another Worker node or moved to the server, depending on a load of each machine in the cluster.

Once deployed the OpenFaaS function with a Label set in the **.yml** file, as shown in Figure 2.2, if the OpenFaaS function includes the tag relative to a k3s worker node label, then the deployment will be set to that node. If the OpenFaaS function requires higher performances, OpenFaaS timeout will stop the deployment.

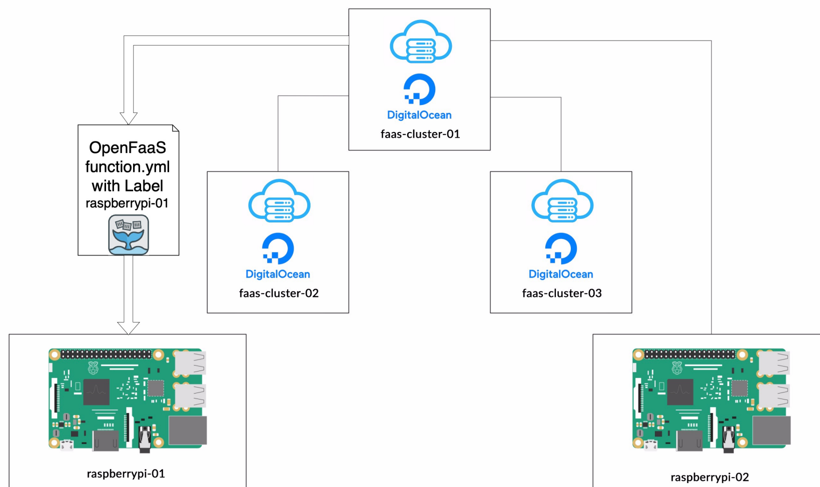

Figure 2.2: Deployment OpenFaaS function on a selected Worker node

We verify how it is possible having a hybrid architecture with today's available open-source frameworks and the possibility to deploy FaaS functions in a flexible way selecting a node.

Chapter 3

Cluster configuration

3.1 Introduction

After the reasoning of how to implement a hybrid cluster, in this Chapter, we enter the project details focusing on how it has been built using the available devices and resources and how it has been configured. The main goal is to validate the solution proposed in section 2.4 that is the verification of how it is possible having a hybrid architecture and the possibility to deploy FaaS functions selecting exactly each node.

3.2 Architecture

The architecture that we are going to cover is based on five machines. The chosen solution is a k3s single server architecture as shown in Figure 3.1. This solution provides the following configuration:

- The main server
- Embedded SQLite Database on the server node
- Agent nodes that are connected to the main server
- Networking handled with Flannel as Cluster Networking Interface

Besides Flannel, as described in Chapter 1.3.3, k3s integrates also CoreDNS as a cluster DNS provider, the Traefik Ingress controller, and Klipper service load balancer.

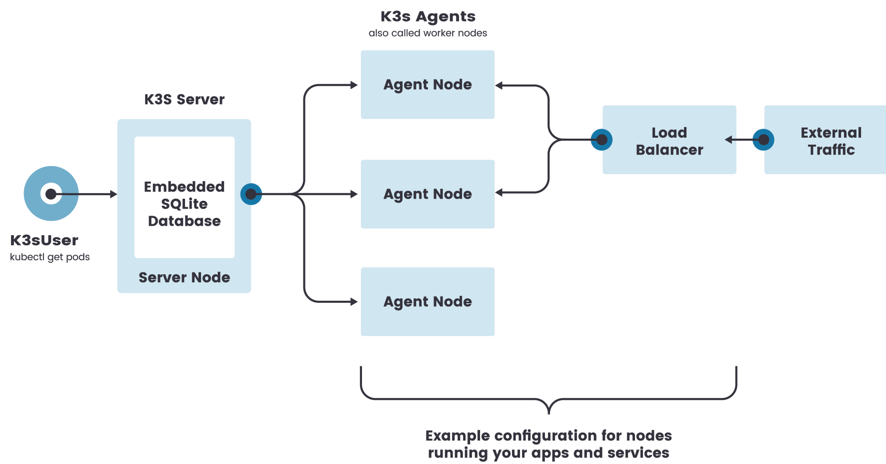

Figure 3.1: k3s Single Server Architecture [k3s20]

Moreover, k3s can be deployed also in case of High Availability requirements[1]. In this case, it is necessary to create a cluster with two or more server nodes, one or more agent nodes, an external database, and a fixed registration address. One of the biggest constraints of a High Availability cluster is the requirement at level hardware with a minimum 2 vCPU and 4 GB of RAM.

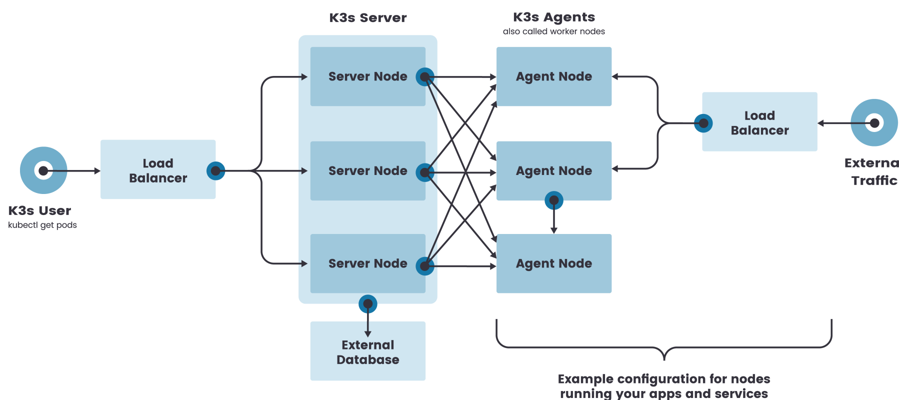

Figure 3.2: k3s High Availability Architecture [k3s20]

[1] https://rancher.com/docs/k3s/latest/en/architecture/

In our case, the server and two nodes are running in Ubuntu, distributed across different countries. In particular, the VPS server is based in Frankfurt, and the two other VPSs are based in Amsterdam. On the network side, each VPS has been assigned to a Floating IP address. This solution provides a stable IP address.

The main server node has the following specification:

- 2 vCPUs
- 4GB
- 80GB disk storage

Instead, the cloud based nodes have the following specifications:

- 1 vCPUs
- 2GB
- 50GB disk storage

Besides, there are two Raspberry Pis with Raspberry Pi OS connected from Italy. For installing the Raspberry Pi OS on the microSD card has been used the script A.1 in Appendix A.

3.3 Environment setup

Previously we have introduced the available machines, explainer the different configurations compatible with k3s focusing on the specifications of server, nodes, and networking.

The following section is going to explain how the environment has been set up from the k3s cluster creation to the OpenFaaS function deployment.

3.3.1 k3s cluster deployment

The k3s installation is quite straightforward. Once installed it, with the code script A.2 in Appendix A, the server can be activated by using the string A.4 in Appendix A.

The following necessary step for deploying the cluster is to add every single worker to the server. To be able to communicate with the worker nodes, the server generates a token code stored at `/var/lib/rancher/k3s/server/node-token`.

Then, k3s should be installed on each worker node and added to the cluster by running the installation script A.5 in Appendix A. `K3S_URL` will be the server URL with add the port *:6443*, instead the `K3S_TOKEN` is the string found before inside the *node-token* file.

Then, k3s has completed the deployment of the hybrid cluster as shown in Figure 3.3.

```
root@faas-cluster-01:~# sudo k3s kubectl get node
NAME              STATUS   ROLES    AGE     VERSION
raspberrypi-02    Ready    <none>   6m29s   v1.18.8+k3s1
faas-cluster-01   Ready    master   15h     v1.18.8+k3s1
faas-cluster-02   Ready    <none>   15h     v1.18.8+k3s1
faas-cluster-03   Ready    <none>   15h     v1.18.8+k3s1
raspberrypi-01    Ready    <none>   6m26s   v1.18.8+k3s1
```

Figure 3.3: List of Nodes of the Hybrid Cluster

At this point, each node has been labeled using the code at A.8. For instance, the Raspberry Pi nodes has been labeled respectively `type=rspbpi-01` and `type=rspbpi-02`. For each node on the cloud side, instead, the labels are for FaaS-Cluster-02 `type=faas-02` and for FaaS-Cluster-03 `type=faas-03` as shown in Figure 3.4.

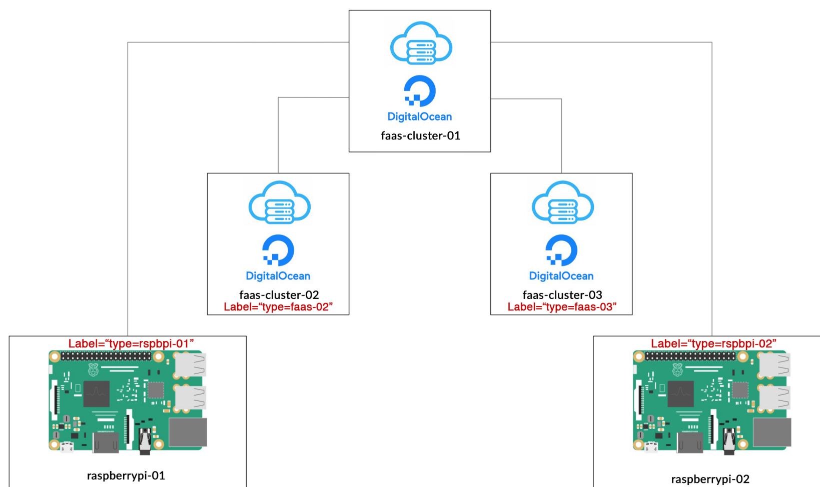

Figure 3.4: Hybrid Cluster with Labels

3.3.1.1 OpenFaaS deployment

As previously explained, the k3s cluster has been provisioned. OpenFaaS provides a `faas-cli` that has been installed using the script below:
`curl -sSL https://cli.openfaas.com | sudo -E sh`

The OpenFaaS deploying has been done using `kubectl` and the plain YALM file. This solution is suggested for development-only projects. In a production environment, OpenFaaS suggests to use Helm chart provided in the `faas-netes` repository[2].

To deploy OpenFaaS is necessary to clone the repository in script B.2 in Appendix B. The whole stack che be deployed with the script B.3 in Appendix B. For accessing OpenFaas is necessary to create a password for the gateway using the script B.4 in Appendix B. Last step is to deploy OpenFaaS with the following code:
`cd faas-netes && kubectl apply -f ./yaml`

With the command `kubectl get deploy -n openfaas`, it can be also monitored that the main OpenFaas services are running.

```
root@faas-cluster-01:~# kubectl get deploy -n openfaas
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
nats                1/1     1            1           33m
prometheus          1/1     1            1           33m
gateway             1/1     1            1           33m
faas-idler          1/1     1            1           33m
basic-auth-plugin   1/1     1            1           33m
queue-worker        1/1     1            1           33m
alertmanager        1/1     1            1           33m
```

Figure 3.5: OpenFaaS Services

Last step is the login in the OpenFaaS UI using the script B.5. The port used by OpenFaaS is the `31112`.

[2] https://github.com/openfaas/faas-netes/blob/master/HELM.md

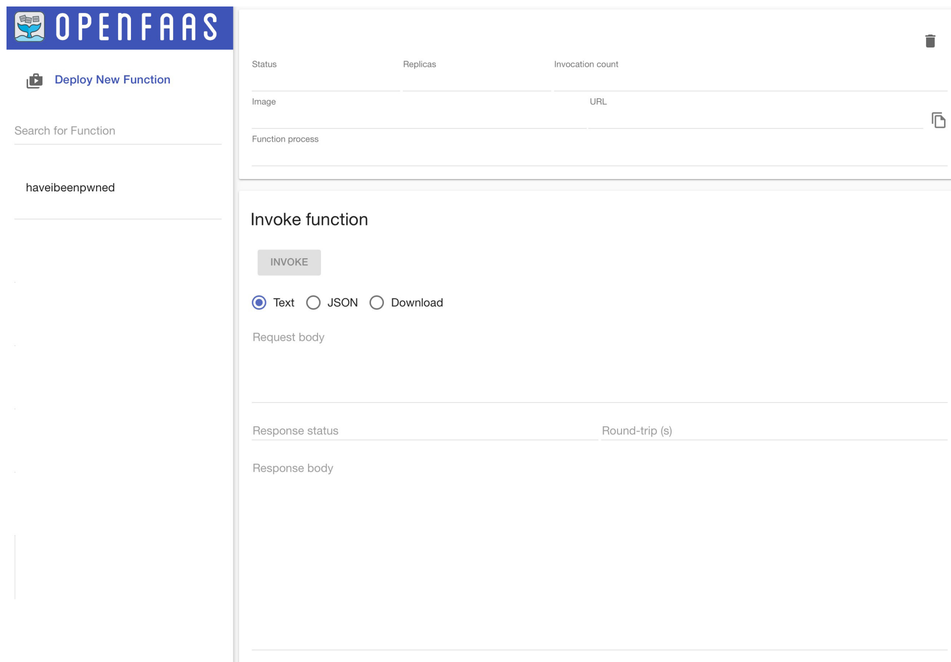

Figure 3.6: OpenFaaS API Portal

3.3.1.2 OpenFaaS Functions

The OpenFaaS function can be deployed using the `faas-cli` and built-in templates. It is also possible to use any binary for Windows or Linux in a Docker container. Another functionality that can be implemented in an OpenFaaS function is the chaining of one function from another. By being all the nodes on the same network, each function can make use of each other directly, keeping a low level of latency by being on the same network.

In the heterogeneous cluster configuration described in this thesis, it is necessary that each OpenFaaS function should provide multi-architecture support. In our case, the cloud nodes and server are based on `amd64` architecture, instead the RaspberryPi's 3 are based on `armhf` architecture.

A possible solution found it is the Docker Buildx CLI Plugin. In the detail, Docker Buildx is an experimental feature that extends the `docker build` command with the full support of the features provided by Moby BuildKit[3] builder toolkit. Docker Buildx can be installed on the Docker version from Docker 18.09+ and Docker 19.03.

With Docker `buuildx` can be built a function for multi-architecture platforms. As shown in the code script 3.1 below, the function is built for amd64, arm7, and arm64 and also pushed it on the Docker registry.

Code 3.1: Docker buildx build script

```
1 docker buildx build function_name.yml \
2 --platform linux/amd64,linux/arm/v7,linux/arm64 \
3 --output "type=image,push=true" \
4 --tag username/function_name:latest ...
  build/function_name/
```

3.3.1.3 Node selection

Once the function has been built, it can be deployed on the nodes by selecting them. For example, in the following code scropt 3.2, it is shown the deployment of a function on the raspberry-01 node. The `-constraint` flag is used for selecting the node based on the label added in section 3.3.1.

Code 3.2: OpenFaaS function deployment on a node

```
1 faas deploy -f function_name.yml --constraint ...
  "type=rspbpi-01"
```

[3]https://github.com/moby/buildkit

Chapter 4

Concluding remarks

The research reported in this thesis has introduced the principal FaaS Platforms & Container Orchestration, focusing on the ability of the possibility to create a hybrid cluster.

OpenFaaS proved to be suitable tools for our purposes, is easy and fast to set up, and capable to offer the required scalability and customizability. Underneath, k3s has shown to be the right Kubernetes platform being lightweight and compatible with a heterogeneous level of devices.

To investigate the possibility to deploy serverless functions with the selectivity of which node to use using, we deployed a cluster based on a server and two nodes cloud-based. Then we added two Raspberry Pis. After setting up the k3s hybrid cluster architecture, OpenFaaS has been deployed, and also the FaaS function selecting the desired node.

4.1 Future directions

All the information in this thesis will be useful in the future works. It is clear which are the right tools to utilize and how to configure a heterogeneous cluster.
To implement a hybrid cluster architecture we relied on k3s, which in turn is a great solution for container orchestration. From the results, it comes out that the computational power provided by Raspberry Pi's 3 is not enough for complex OpenFaaS function deployment. With the growing computational power provided by new devices, like the latest Raspberry Pi 4, for some tasks, this kind of limitation could be solved. In the case of HA cluster with Container Orchestration is still suggested to use server and worker nodes with higher hardware specification as illustrated by k3s[1].

[1] https://rancher.com/docs/k3s/latest/en/installation/ha/

Although k3s is a very supported Open-Source project, other solutions can be tested. At the same time, the community around OpenFaaS is growing up rapidly in the last years and has many different projects are in working progress.

OpenFaaS founder, Alex Ellis, has released a lightweight version called faasd[2]. It offers many of the benefits of containers and OpenFaaS, but without the complexity and operational costs of Kubernetes. The container automation in faasd is handled by `containerd`.

Even if faasd is a promising solution, it is still under active development and not ready for a production environment. Moreover, functionalities provided by k3s or Kubernetes are still not available. In the future, faasd could be another solution for deploying a hybrid cluster with FaaS functions, especially when the devices involved are generally provided of low computational power.

The next step, then, is to transform the cluster in a real production environment. To realize a system using k3s and OpenFaaS it will be necessary to understand how to implement a more robust network, optimized for the multi-architecture deployment.

On the network side, k3s includes Flannel. One of the main Flannel constraints is that every node IP needs to be able to see every other node IP. For a production environment where availability, privacy, and performances are required, should be considered building up a VPS network using solutions like WireGuard with Kilo[3].

At the same time, Docker's `Buildx` used in Chapter 3.3.1.2 is still an experimental features. It means that cannot be used in production environments. With the next Docker releases, `dockerx` could be supported officially.

In this thesis, the capability to select exactly each node has been reached through a careful naming and a modification of each OpenFaaS function, based on the different hardware architectures inside the cluster. So, there is the need to find automation for labeling and deployment based on the actual architecture and the available resources on each worker node.

[2]https://github.com/openfaas/faasd
[3]https://github.com/squat/kilo

Appendix A

Raspberry Pi & k3s code scripts

Code A.1: Automatic bash script for Raspbian installation on microSD Card

```
1  #!/bin/bash
2
3  set -e
4  set -o pipefail
5
6  RASPBERRYHOSTNAME=$1
7  RASPBERRYMOUNTPATH=$2
8  PUBLICSSHKEY=$3
9  RASPBERRYTIMEZONE=$4
10
11 IMG=$(ls raspbian.img)
12 if [[ -z "$IMG" ]]; then
13     wget https://downloads.raspberrypi.org/ ...
         raspbian_lite_latest -O raspbian.zip
14     unzip raspbian.zip
15     mv *.img raspbian.img
16     rm -f raspbian.zip
17 fi
18
19 sudo dd bs=1M if=raspbian.img of=/dev/sda ...
     status=progress
20
21 sudo mkdir $RASPBERRYMOUNTPATH
```

```
22 sudo mount /dev/sda2 $RASPBERRYMOUNTPATH
23 cat wpa_supplicant.conf | sudo tee -a ...
    $RASPBERRYMOUNTPATH/etc/wpa_supplicant/ ...
    wpa_supplicant.conf > /dev/null
24 echo $RASPBERRYHOSTNAME | sudo tee ...
    $RASPBERRYMOUNTPATH/etc/hostname > /dev/null
25 echo "127.0.1.1        $RASPBERRYHOSTNAME" | sudo tee ...
    -a $RASPBERRYMOUNTPATH/etc/hosts
26
27 sudo rm $RASPBERRYMOUNTPATH/etc/localtime
28 sudo cp $RASPBERRYMOUNTPATH/usr/share/zoneinfo/ ...
    $RASPBERRYTIMEZONE $RASPBERRYMOUNTPATH/etc/localtime
29
30 sudo sed -i 's/^#PasswordAuthentication ...
    yes/PasswordAuthentication no/g' ...
    $RASPBERRYMOUNTPATH/etc/ssh/sshd_config
31 sudo sed -i 's/^UsePAM yes/UsePAM no/g' ...
    $RASPBERRYMOUNTPATH/etc/ssh/sshd_config
32 sudo mkdir $RASPBERRYMOUNTPATH/home/pi/.ssh
33 echo -n $PUBLICSSHKEY >> authorized_keys
34 sudo mv authorized_keys ...
    $RASPBERRYMOUNTPATH/home/pi/.ssh/
35 chmod 644 ...
    $RASPBERRYMOUNTPATH/home/pi/.ssh/authorized_keys
36
37 sudo umount $RASPBERRYMOUNTPATH
38 sudo mount /dev/sda1 $RASPBERRYMOUNTPATH
39 sudo touch $RASPBERRYMOUNTPATH/ssh
40
41 echo -n ' cgroup_enable=cpuset cgroup_enable=memory' ...
    | sudo tee -a $RASPBERRYMOUNTPATH/cmdline.txt
42 sudo sh -c "tr -d '\n' < ...
    $RASPBERRYMOUNTPATH/cmdline.txt > ...
    $RASPBERRYMOUNTPATH/cmdline2.txt"
43 sudo mv $RASPBERRYMOUNTPATH/cmdline2.txt ...
    $RASPBERRYMOUNTPATH/cmdline.txt
44
45 sudo umount $RASPBERRYMOUNTPATH
```

Code A.2: k3s installation script with Agent

```
1 curl -sfL https://get.k3s.io | sh -
```

Code A.3: k3s installation script without Agent

```
1 curl -sfL https://get.k3s.io | ...
  INSTALL_K3S_EXEC="--disable-agent" sh -
```

Code A.4: k3s server activation

```
1 sudo k3s server
```

Code A.5: k3s node installation & connection to server using $TOKEN

```
1 curl -sfL https://get.k3s.io | ...
  K3S_URL=https://server_address:6443 ...
  K3S_TOKEN=token_code sh -
```

Code A.6: k3s node installation and token connection

```
1 curl -sfL https://get.k3s.io | ...
  K3S_URL=https://server_address:6443 ...
  K3S_TOKEN=token_code sh -
```

Code A.7: k3s get node command

```
1 sudo k3s kubectl get node
```

Code A.8: Node labeling on k3s

```
1 kubectl label nodes raspberrypi-01 type=rspbpi-01
2 kubectl label nodes raspberrypi-02 type=rspbpi-02
3 kubectl label nodes faas-cluster-02 type=faas-02
4 kubectl label nodes faas-cluster-03 type=faas-03
```

Appendix B

OpenFaaS code scripts

Code B.1: OpenFaaS CLI installation

```
1 curl -sSL https://cli.openfaas.com | sudo -E sh
```

Code B.2: Clone OpenFaaS repository

```
1 git clone https://github.com/openfaas/faas-netes
```

Code B.3: Deploy openfaas and openfaasfn namespaces

```
1 kubectl apply -f ...
    https://raw.githubusercontent.com/openfaas/faas-netes/ ...
    master/namespaces.yml
```

Code B.4: OpenFaaS password creation

```
1 # generate a random password
2 PASSWORD=$(head -c 12 /dev/urandom | shasum| cut -d' ...
    ' -f1)
3
4 kubectl -n openfaas create secret generic basic-auth \
5 --from-literal=basic-auth-user=admin \
6 --from-literal=basic-auth-password="$PASSWORD"
```

Code B.5: OpenFaaS password creation

```
1 export OPENFAAS_URL=http://127.0.0.1:31112
2
3 echo -n $PASSWORD | faas-cli login --password-stdin
```

Bibliography

[Ale19] Alex Ellis. *OpenFaaS Function as a Service*. [Online; accessed October 11, 2019]. 2019. URL: https://blog.alexellis.io/introducing-functions-as-a-service/.

[Doc19a] Docker. *Docker Engines components flow*. [Online; accessed October 13, 2019]. 2019. URL: https://docs.docker.com/engine/docker-overview/.

[Doc19b] Docker. *Docker Engines components flow*. [Online; accessed September 11, 2020]. 2019. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/.

[Kub19a] Kubernetes. *Kubernetes Master and Worker components*. [Online; accessed October 15, 2019]. 2019. URL: https://kubernetes.io/blog/2018/08/03/out-of-the-clouds-onto-the-ground-how-to-make-kubernetes-production-grade-anywhere/.

[Kub19b] Kubernetes. *Kubernetes node overview*. [Online; accessed October 13, 2019]. 2019. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/.

[Ope19] OpenFaaS Project. *Conceptual OpenFaas diagram*. [Online; accessed October 13, 2019]. 2019. URL: https://docs.openfaas.com/architecture/gateway/.

[Ran19] Rancher Labs. *k3s Server & k3s Agent*. [Online; accessed October 13, 2019]. 2019. URL: https://k3s.io.

[The19] TheNewStack. *Architectural View of Apache OpenWhisk*. [Online; accessed October 14, 2019]. 2019. URL: https://thenewstack.io/behind-scenes-apache-openwhisk-serverless-platform/.

[k3s20] k3s. *k3s High Availability infrastructure*. [Online; accessed September 1, 2020]. 2020. URL: https://rancher.com/docs/k3s/latest/en/installation/ha/.