UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DESIGN AND IMPLEMENTATION OF INTELLIGENT LOAD-BALANCING MECHANISMS FOR VIRTUAL NETWORK FUNCTIONS

Diploma Thesis

Georgios Charmanis

Supervisor: Athanasios Korakis

Volos 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DESIGN AND IMPLEMENTATION OF INTELLIGENT LOAD-BALANCING MECHANISMS FOR VIRTUAL NETWORK FUNCTIONS

Diploma Thesis

Georgios Charmanis

Supervisor: Athanasios Korakis

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΧΕΔΙΑΣΜΟΣ ΚΑΙ ΑΝΑΠΤΥΞΗ ΕΞΥΠΝΩΝ ΜΗΧΑΝΙΣΜΩΝ ΕΞΙΣΟΡΡΟΠΗΣΗΣ ΦΟΡΤΟΥ ΓΙΑ ΕΙΚΟΝΟΠΟΙΗΜΕΝΕΣ ΣΥΝΑΡΤΗΣΕΙΣ ΔΙΚΤΥΟΥ

Διπλωματική Εργασία

Γεώργιος Χαρμάνης

Επιβλέπων: Αθανάσιος Κοράκης

Βόλος 2021

Approved by the Examination Committee:

Supervisor **Athanasios Korakis**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Antonios Argyriou**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitrios Katsaros**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

First of all, I would like to warmly thank my supervisor, Professor Athanasios Korakis, who not only conveyed his love and enthusiasm for the network and telecommunications sector to me but also gave me the opportunity to evolve by offering me an internship in his laboratory, that later led to this thesis.

Next, I would like to express my deepest gratitude to the postdoctoral fellows Nikos Makris and Apostolos Apostolaras, who despite their heavy workload, provided me with consistent help and support by sharing their extensive knowledge with me.

To my family and friends, you should know that your constant love and support throughout this academic journey was worth more than what I can express on paper.

Last but not least, I would like to thank the commander of the military unit that I currently serve, Major Dimitrios Chatzoulis, who supported me by doing everything that he could to assist me complete this thesis while fulfiling my military obligations.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Georgios Charmanis

Abstract

We are going through an era -landmark- of technological development and innovation. Particularly in the field of wireless networks, the spread of the 5th generation network (5G) and the trend of the ”Network of Things” (Internet of Things) are expected to radically change the way the mechanisms are designed and implemented in the cloud. Every day, more and more ”things” are connected to the Internet, resulting in a rapid increase in the volume of information that travels through the network and is processed by it.

Thus, the problem of the optimal management of the available resources and, at the same time, of the ”fair” load-balancing on them, is becoming of major importance.

The fore-mentioned problem has forced network engineers to update the design of networks, moving away from the outdated server-client model, where the server is a central machine bombarded with user requests. The introduction of virtual machines (VMs, Containers), but also their control infrastructures, now offer greater flexibility and security in the management of applications.

This thesis is the design and implementation of an intelligent load balancing mechanism in the Kubernetes management environment, that can operate both independently and on top of other Kubernetes mechanisms. Our implementation uses state-of-the-art tools and can be fully integrated into an operating system, in order to efficiently allocate system resources to incoming customer requests. Our system can be adapted autonomously based on extensive measurements taken in the background, in order to change the resource allocation dynamically, in proportion to the system load.

This work presents in detail the tools used and the implementation that took place. For the evaluation of the proposed solution, the NITOS experimental infrastructure was used, with large-scale experiments in real environment.

Περίληψη

Διανύουμε μια εποχή -ορόσημο- της τεχνολογικής ανάπτυξης και καινοτομίας. Ιδιαίτερα στον τομέα των ασυρμάτων δικτύων, η διάδοση του δικτύου 5ης γενιάς (5G) και η τάση του ”Δικτύου των Πραγμάτων” (Internet of Things) προβλέπεται να αλλάξουν ριζικά τον τρόπο σχεδιασμού και υλοποίησης των μηχανισμών στο cloud. Καθημερινά, όλο και περισσότερα ”πράγματα” συνδέονται στο Internet, με αποτέλεσμα να αυξάνεται ραγδαία ο όγκος πληροφορίας που ταξιδεύει και διαχειρίζεται το δίκτυο.

Άρα, το πρόβλημα της βέλτιστης διαχείρισης των διαθέσιμων πόρων και, παράλληλα, της ”δίκαιας” εξισορρόπησης φόρτου σε αυτούς, καθίσταται πλέον μείζονος σημασίας.

Αυτό έχει αναγκάσει τους μηχανικούς δικτύων να εξελίξουν τον σχεδιασμό των δικτύων, ξεφεύγοντας από το απαρχαιωμένο μοντέλο του server-client, όπου ο server είναι ένα κεντρικό μηχάνημα που βομβαρδίζεται από requests των χρηστών. Η εισαγωγή των εικονικών μηχανών (VMs, Containers), αλλά και των υποδομών ελέγχου αυτών, προσφέρει πλέον μεγαλύτερη ευελιξία και ασφάλεια στη διαχείριση των εφαρμογών.

Αυτή η διπλωματική εργασία αποτελεί τον σχεδιασμό και την υλοποίηση ενός έξυπνου μηχανισμού εξισορρόπησης φόρτου στο περιβάλλον διαχείρισης Kubernetes, ο οποίος δύναται να λειτουργήσει και αυτόνομα αλλά και σε συνδυασμό με άλλους μηχανισμούς του Kubernetes. Η υλοποίηση μας χρησιμοποιεί εργαλεία τελευταίας γενιάς και μπορεί να ενσωματωθεί πλήρως σε ένα λειτουργικό σύστημα, με σκοπό την αποδοτική ανάθεση των πόρων του συστήματος σε εισερχόμενες αιτήσεις πελατών. Το σύστημα μας μπορεί να προσαρμόζεται αυτόνομα βάσει εκτενών μετρήσεων που παίρνει στο παρασκήνιο, ώστε να αλλάζει την ανάθεση πόρων δυναμικά, αναλογικά με τον φόρτο του συστήματος.

Στην εργασία παρουσιάζονται εκτενώς τα εργαλεία που χρησιμοποιήθηκαν και η υλοποίηση που πραγματοποιήθηκε. Για την αποτίμηση της προτεινόμενης λύσης, χρησιμοποιήθηκε η πειραματική υποδομή NITOS, με πειράματα μεγάλης κλίμακας σε πραγματικό περιβάλλον.

Table of contents

List of figures

Abbreviations

IoT	Internet of Things
K8s	Kubernetes
VM	Virtual Machine
HPA	Horizontal Pod Autoscaler
CPU	Central Processing Unit
DNS	Domain Name System
IP	Internet Protocol
UI	User Interface

Chapter 1

Introduction

Nowadays, a need has emerged for hyperscale and continuous delivery, especially on high demand. As a result, big enterprises have started to migrate their services from big monolithic machines to microservices running on cloud. Using containers to deploy microservices, it is now possible to control and update cloud applications without interrupting their runtime.

Microservices is an architectural design for building a distributed application. They break an application into independent, loosely-coupled, individually deploy-able services. This architecture allows for each service to scale or update without disrupting other services in the application and enables the rapid, frequent and reliable delivery of large, complex applications, so that applications can be continuously delivered to end users [1]. Deploying a containerized application in production, usually means hundreds or thousands of containers running. Controlling and configuring all of those containers can be challenging without a proper framework.

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery [2]. The smallest, most basic deployable objects in Kubernetes are pods. A Pod is meant to run a single instance of the application on the cluster. Pods are created usually in groups, called replicas, by a controller, to run the application.

Such a set of replicated Pods are created and managed by a controller, such as a Deployment, capable of scaling the deployment horizontally, changing the number of pods as necessary [3]. For example, if the incoming requests are too many to be handled by a single pod, the controller can be configured to increase the number of replicas. A set of related pods

that have the same set of functions is also called a ”service” and this is what is visible to the outside world. Every new pod is assigned a new IP address, IP addresses for pods are not stable; therefore, direct communication between pods is not generally possible. However, services have their own IP addresses, which are relatively stable; thus, a request from an external resource is made to a service rather than a pod, and the service then dispatches the request to an available pod [4].

However, which specific replica of the Deployment, will get to execute the incoming request?

To answer this question, Kubernetes uses load balancing mechanisms, which redirect the incoming requests ”fairly” among the replicas using a round-robin algorithm.

1.1 Subject of Thesis

This thesis is the design and implementation of an intelligent load-balancing mechanism, which can either operate in combination with the K8s inbuilt LoadBalancer or autonomously. The developed algorithm was formulated by adapting the ”water filling” method to the subject’s specific needs and compared with a round-robin load balancer, who was based on the K8s mechanisms.

1.1.1 Contribution

The thesis contribution can be summarized as follows:

1. A baseline cluster was created and managed through Kubernetes, deploying all the relevant monitoring and visualization tools.

2. The Kubernetes Horizontal Pod Autoscaler (HPA) was thoroughly examined and was the basis for the deployment of the first algorithm.

3. Two ancillary services were deployed, a custom DNS resolver and a metrics fetcher, communicating with the Prometheus monitoring solution.

4. A novel load balancing solution was developed and deployed in the cluster, written in Python language, enabling the dynamic resource allocation of the requests to the available service replicas.

5. All the functionality and logic was embedded in a single container pod, enabling its portability across different clusters.

6. A thorough comparison of the two algorithms was made and the conclusions were explained.

1.2 Thesis Structure

Chapter 2 presents the NITOS testbed, used for all the experiments conducted in this thesis. NITOS nodes also host the entire Kubernetes cluster.

Chapter 3 contains the analysis of concepts like containers, microservices and container orchestration, as well as a complete breakdown of the Kubernetes environment and its components.

Chapter 4 describes the system setup, that is the essential services that the cluster needs to function and the tools that we used to monitor the cluster and collect the results.

Chapter 5 presents the theoretical approach of our goal, the development of the two algorithms and of two background services and demonstrates the results of the two algorithms running.

In Chapter 6, all the conclusions and the possible future extensions are discussed.

Chapter 2

Testing Platform

2.1 About NITLAB

NITLAB [5] stands for Network Implementation Testbed Laboratory of the Department of Electrical and Computer Engineering at University of Thessaly. NITLAB is also affiliated with the Centre for Research & Technology Hellas (CERTH). The research of the lab focuses on the design, study and implementation of wireless and wired schemes and their performance in the real environment. In this context, NITlab has developed a facility named NITOS, which stands for Network Implementation Testbed using Open Source platforms. NITOS is one of the facilities of the OneLab Federation and it can also be accessed through the OneLab portal.

NITOS [6] facility currently consists of over 100 operational wireless nodes and is designed to achieve reproducibility of experimentation, while also supporting evaluation of protocols and applications in real world settings. NITOS facility is geographically separeted in 3 deployments. The Outdoor one at the exterior of the University of Thessaly (UTH) campus building, the Indoor one at the basement of the UTH's building and the Office testbed deployed at CERTH's office building in Volos.

The control and management of the facility is done using the cOntrol and Management Framework (OMF) open-source software. Users can perform their experiments by reserving slices (nodes, access points, base stations or frequency spectrum) of the testbed through the NITOS scheduler that, together with OMF support, ease of use for experimentation and code development.

The NITOS platform is open to any researchers who would like to test their protocols in real-world settings. They are given the opportunity to implement their novel protocols and

Figure 2.1: NITOS Outdoor Testbed

study their behavior in a custom tailor-made environment. NITlab is constantly in the process of extending its testbed capabilities.

The main experimental components of NITOS are:

- A wireless experimentation testbed, which consists of 100 powerful nodes (some of them mobile), that feature multiple wireless interfaces and allow for experimentation with heterogeneous (Wi-Fi, WiMAX, LTE, Bluetooth) wireless technologies.

- A Cloud infrastructure, which consists of 7 HP blade servers and 2 rack-mounted ones providing 272 CPU cores, 800 Gb of Ram and 22TB of storage capacity, in total. The network connectivity is established via the usage of an HP 5400 series modular Openflow switch, which provides 10Gb Ethernet connectivity amongst the cluster's modules and 1Gb amongst the cluster and GEANT.

- A wireless sensor network testbed, consisting of a controllable testbed deployed in UTH's offices, a city-scale sensor network deployed in Volos city and a city-scale mobile sensing infrastructure that relies on bicycles of volunteer users. All sensor platforms are custom, developed by UTH, supporting Arduino firmware and exploit several wireless technologies for communication (ZigBee, Wi-Fi, LTE, Bluetooth, IR).

- A Software Defined Radio (SDR) testbed that consists of Universal Software Radio Peripheral (USRP) devices attached to the NITOS wireless nodes. USRPs allow the

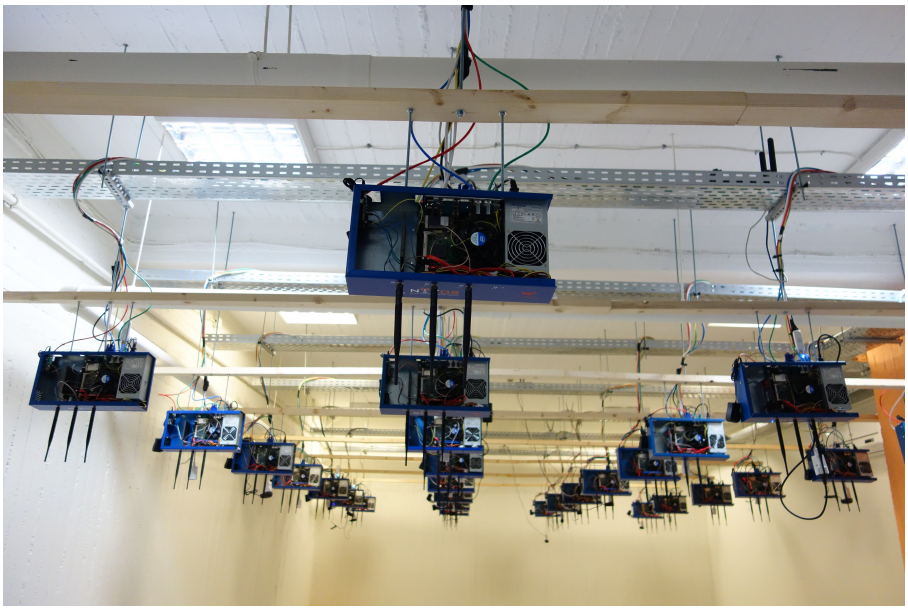


Figure 2.2: NITOS RF Isolated Testbed

researcher to program a number of physical layer features (e.g. modulation), thereby enabling dedicated PHY layer or cross-layer research.

- A Software Defined Networking (SDN) testbed that consists of multiple OpenFlow technology enabled switches, connected to the NITOS nodes, thus enabling experimentation with switching and routing networking protocols. Experimentation using the OpenFlow technology can be combined with the wireless networking one, hence enabling the construction of more heterogeneous experimental scenarios.

  The testbed is based on open-source software that allows the design and implementation of new algorithms, enabling new functionalities on the existing hardware. The control and management of the testbed is done using the cOntrol and Management Framework (OMF) open-source software. NITOS supports evaluation of protocols and applications under real world settings and is also designed to achieve reproducibility of experimentation.

Chapter 3

Containers and Microservices

3.1 Introduction

It is crucial, for the deeper understanding of the subject of this thesis, that we take a closer look at what are containers and microservices, as well as the reasons why we need them.

First of all, what are microservices? According to [7], a microservices architecture splits the application into multiple services that perform fine-grained functions and are part of your application as a whole. Each of the microservices will implement a different logical function for the application. Traditional applications have monolithic architectures where all the application's components and functions are in a single instance; microservices break apart monolithic applications into smaller parts.

Microservices are implemented by using containers. Containers are packages of your software that include everything that it needs to run, including code, dependencies, libraries, binaries, and more. Docker and Kubernetes are the most popular frameworks to orchestrate multiple containers in enterprise environments. Compared to virtual machines (VMs), containers share the operating system kernel instead of having a full copy of it, such as making multiple VMs in a single host. Although it's possible to put microservices into multiple VMs, containers would be typically used in this case since they take up less space and are faster to boot up.

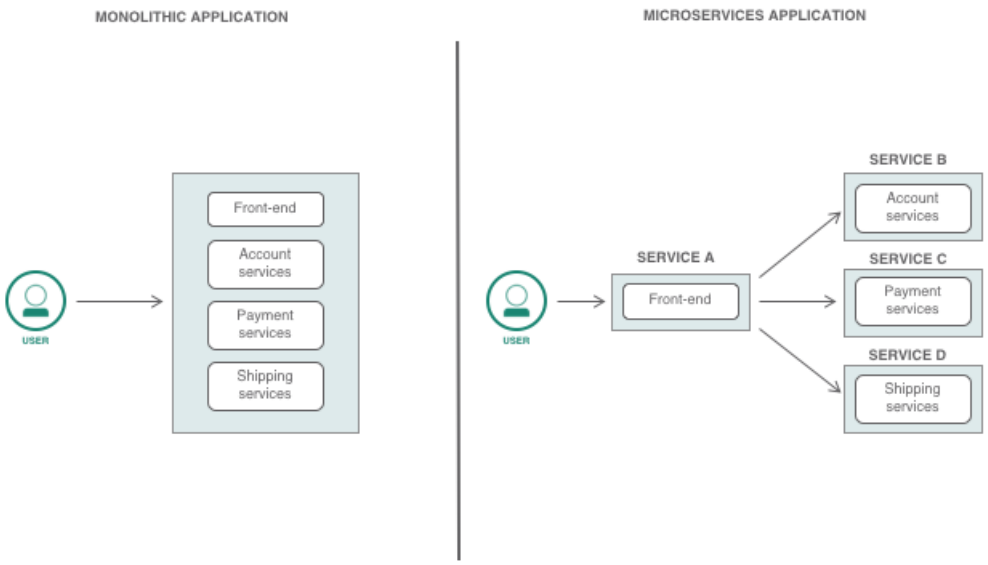
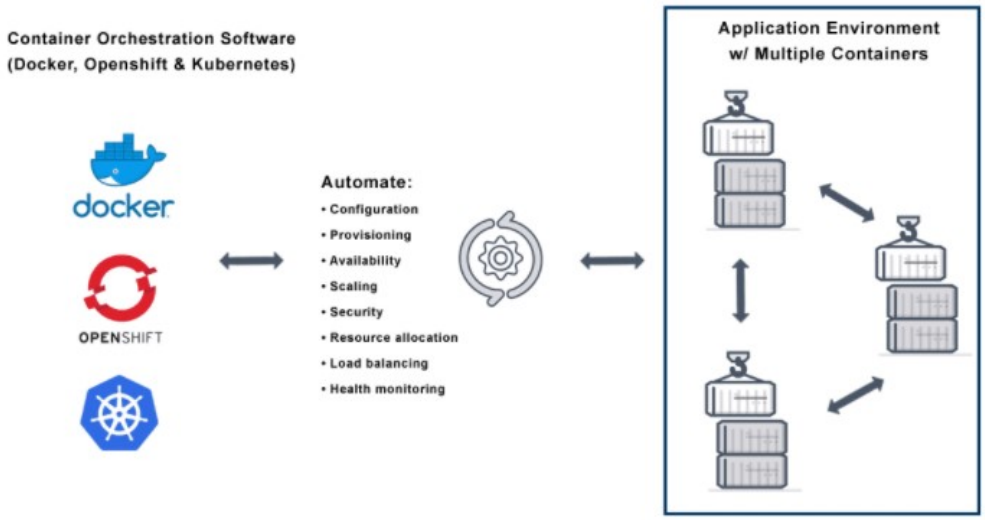


Figure 3.1: Comparison between monolithic and microservices architectures.

3.2 Container Orchestration

Container orchestration is the automatic process of managing or scheduling the work of individual containers for applications based on microservices within multiple clusters. The widely deployed container orchestration platforms are based on open-source versions like Kubernetes, Docker Swarm or the commercial version from Red Hat OpenShift. The following diagram demonstrates the container orchestration process.



Container orchestration works with tools like Kubernetes and Docker Swarm, with Ku-

bernetes being our subject of interest. Configurations files tell the container orchestration tool how to network between containers and where to store logs. The orchestration tool also schedules deployment of containers into clusters and determines the best host for the container. After a host is decided, the orchestration tool manages the lifecycle of the container based on predetermined specifications. Container orchestration tools work in any environment that runs containers. [8]

3.3 Diving into Kubernetes

Kubernetes, also known as K8s, is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start operating. This behaviour needs to be handled by a system. Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more.

Kubernetes and its components are presented according to the official Kubernetes Documentation [9].

3.3.1 Kubernetes Components

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

3.3.2 Control Plane Components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied). Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine.

kube-apiserver

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

kube-scheduler

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on. Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager

A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager makes possible the linking of the cluster into the cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with the cluster. The cloud-controller-manager only runs controllers that are specific to your cloud provider.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

- Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- controller: For setting up routes in the underlying cloud infrastructure
- Service controller: For creating, updating and deleting cloud provider load balancers

3.3.3 Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

kube-proxy

Kube-proxy is a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. Kube-proxy maintains network rules on nodes. These network rules allow network communication to the Pods from network sessions inside or outside of the cluster. Kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

Container runtime

The container runtime is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

3.3.4 Addons

Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the kube-system namespace.

DNS

While the other addons are not strictly required, all Kubernetes clusters should have cluster DNS, as many examples rely on it. Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services. Containers started by Kubernetes automatically include this DNS server in their DNS searches.

Web UI (Dashboard)

Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

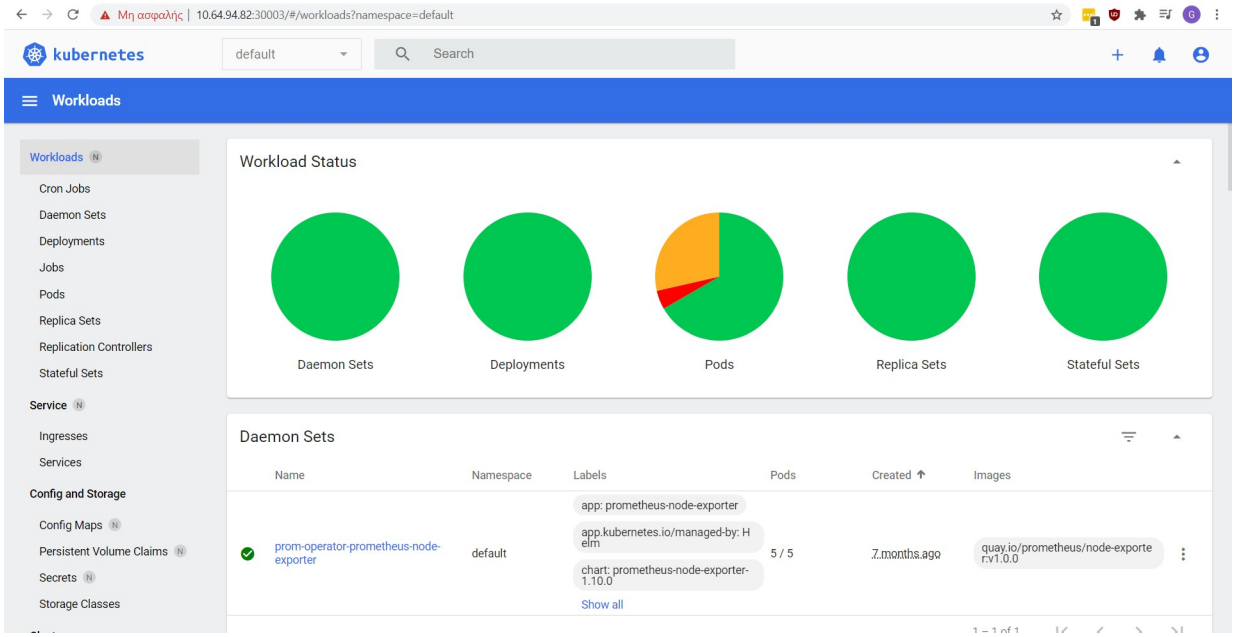


Figure 3.2: Web UI of a functioning cluster.

Container Resource Monitoring

Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

Cluster-level Logging

A cluster-level logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

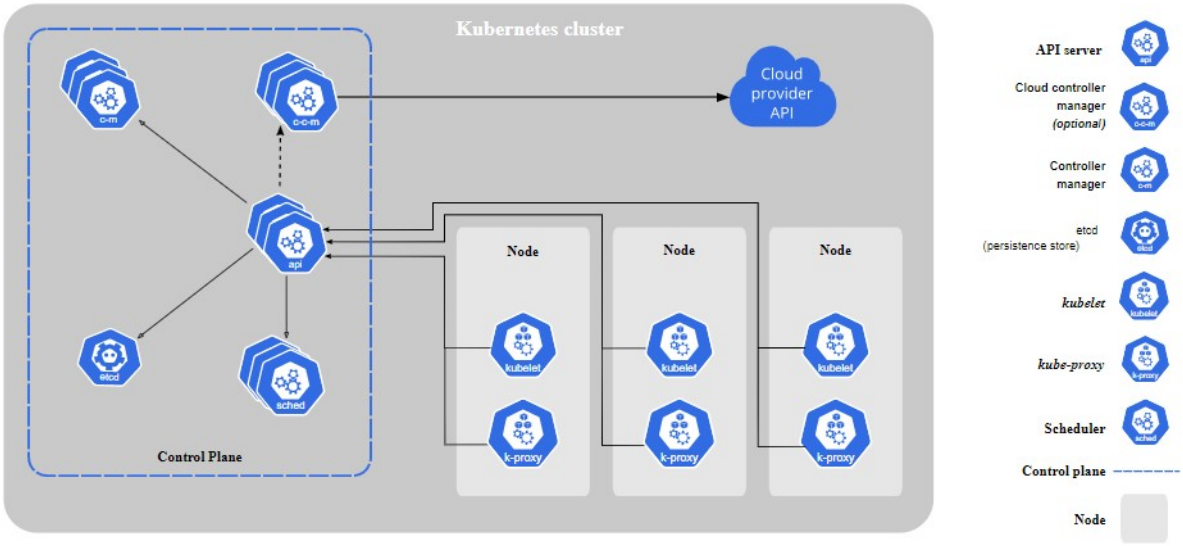


Figure 3.3: A K8s cluster with all the components tied together.

Chapter 4

System Setup

4.1 Cluster Setup

For the Kubernetes cluster setup, we used a permanent VM of the lab as the control plane. We then configured a ”worker” OMF image to load every time on NITOS nodes with all the proper settings.

Our first task, was to become familiar with the K8s inbuilt load balancer, the HPA.

The HPA [10] is a Kubernetes mechanism that automatically scales the number of Pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization (or, with custom metrics support, on some other application-provided metrics). Note that Horizontal Pod Autoscaling does not apply to objects that can’t be scaled, for example, DaemonSets. The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed metrics such as average CPU utilisation, average memory utilisation or any other custom metric to the target specified by the user.

We installed the proper Helm Charts and created a simple pod running an apache server for testing. The HPA by default takes under consideration 2 fundamental metrics: 1) The CPU utilization and 2) the Memory Consumption. This thesis’ load balancer will work on top of the HPA, achieving a more intelligent load balancing procedure. We experimented by setting a threshold of 50% regarding the CPU utilization and sending continuous traffic to the apache server. As soon as the percentage exceeded the threshold, new replicas were automatically created. If the traffic suddenly stopped, some of the replicas were deleted. We will use the

HPA later in combination with our own load balancer.

4.2 Monitoring Tools

4.2.1 Prometheus

Prometheus [11] is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

Features

Prometheus's main features are:

- a multi-dimensional data model with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language to leverage this dimensionality
- no reliance on distributed storage; single server nodes are autonomous
- time series collection happens via a pull model over HTTP
- pushing time series is supported via an intermediary gateway
- targets are discovered via service discovery or static configuration
- multiple modes of graphing and dashboarding support

Components

The Prometheus ecosystem consists of multiple components, many of which are optional:

- the main Prometheus server which scrapes and stores time series data
- client libraries for instrumenting application code

- a push gateway for supporting short-lived jobs
- special-purpose exporters for services like HAProxy, StatsD, Graphite, etc.
- an alertmanager to handle alerts
- various support tools

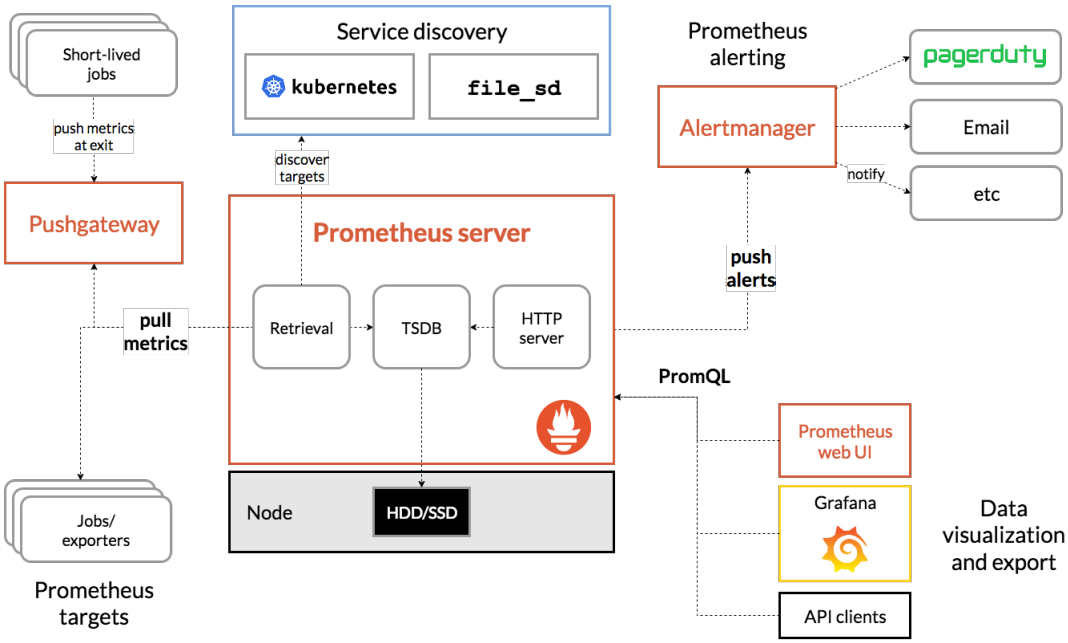


Figure 4.1: Prometheus Architecture and some of its ecosystem components.

4.2.2 Grafana

Grafana [12] is an open-source platform for data visualization, monitoring and analysis. It allows users to create dashboards with panels, each representing specific metrics over a set time-frame. Every dashboard is versatile, so it could be custom-tailored for a specific project or any development and/or business needs.

There is a variety of supported data sources for Grafana (Prometheus, MySQL, Postgres to name just a few) , for each of those, Grafana has a customized query editor and specific syntax.

Grafana Notions

- A Panel is the basic visualization building block presented per the metrics selected. Grafana supports graph, singlestat, table, heatmap, and freetext panels, as well as inte-

gration with official and community-built plugins (like world map or clock) and apps that could be visualized, too. Each panel can be customized in terms of style and format; all panels could be dragged, dropped, resized, and rearranged.

- A Dashboard is a set of individual panels arranged on a grid with a set of variables (like server, application and sensor name). By changing variables, you can switch the data being displayed in a dashboard (for instance, data from two separate servers). All dashboards could be customized and sliced and diced depending on the user needs. Grafana has a large community of contributors and users, so there is a large ecosystem of ready-made dashboards for different data types and sources.

- Dashboards can utilize annotations to display certain events across panels. When hovering over an annotation, you can get event description and tags, for instance, to track when server responds with 5xx error code or when the system restarts. This way, it is easy to correlate with a time, specific event and its consequences in an application and investigate system behaviour.

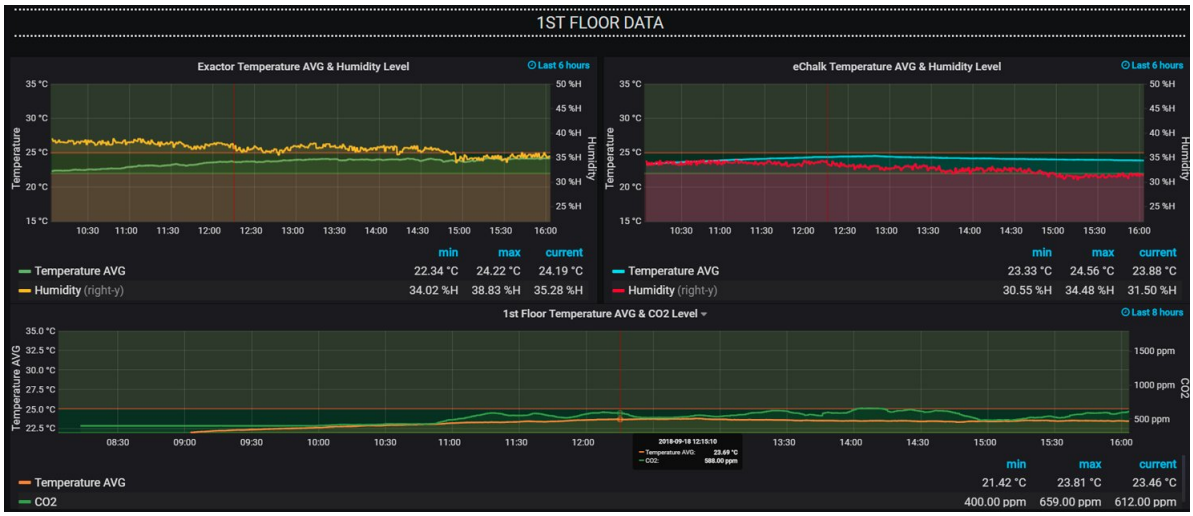


Figure 4.2: Grafana Dashboard with Office Weather Parameters.

4.2.3 Prometheus Adapter

The Prometheus adapter [13] is a Kubernetes Aggregation Layer extension and operates as an extension API server. It knows how to communicate with both Kubernetes and Prometheus, acting as a translator between the two. The adapter processes the metrics coming from Prometheus as follows:

- Discovery: it discovers available metrics.
- Association: it determines which kubernetes resource each metric is associated with.
- Naming: it determines how it should expose the metrics in the custom metric API.
- Querying: Finally, it figures out how it should query Prometheus to get the actual numbers.

The adapter performs each of the steps for each metric. These steps are formally described for each metric with a rule:

```
rules:
  - seriesQuery: 'http_requests_total{kubernetes_namespace!="",kubernetes_pod_name!=""}'
```

The discovery step is defined by a ”seriesQuery”, which is a query that returns a metric series definition (not numbers).

```
rules:
  - seriesQuery: 'http_requests_total{kubernetes_namespace!="",kubernetes_pod_name!=""}'
    resources:
      overrides:
        kubernetes_namespace: {resource: "namespace"}
        kubernetes_pod_name: {resource: "pod"}
```

The association step maps labels to known resources. It is introduced by the ”resources” keyword, followed by an ”overrides” map, where labels are mapped to known resources.

```
resources: {template: "kubernetes_<<.Resource>>"}
```

4.3 Thesis Custom Setup

Initially, we had to deploy Prometheus and Grafana on our cluster. For that, we used the ”Prometheus Operator” chart from Helm to avoid two different installations and configurations. However, to link our metrics with the HPA, we had also to deploy the Prometheus adapter, an extra package for leveraging the metrics collected by Prometheus and using them to make scaling decisions [14]. After the installation process, the following pods were deployed:

- Alertmanager - The Alertmanager handles alerts sent by client applications such as the Prometheus server. It takes care of deduplicating, grouping, and routing them to the correct receiver integration such as email, PagerDuty, or OpsGenie. It also takes care of silencing and inhibition of alerts.
- Prometheus-Adapter
- Kube-State-Metrics - The kube-state-metrics is focused on generating completely new metrics from Kubernetes' object state.
- Grafana
- Node Exporter (on every worker node) - The Node Exporter is an 'official' exporter that collects technical information from Linux nodes, such as CPU, Disk, Memory statistics.
- Prometheus-Operator - The Prometheus Operator provides Kubernetes native deployment and management of Prometheus and related monitoring components.
- Metrics-Server - The Metrics Server collects resource metrics from Kubelets and exposes them in Kubernetes apiserver through Metrics API for use by Horizontal Pod Autoscaler and Vertical Pod Autoscaler.

Prometheus also exposed some of the basic Kubernetes mechanisms as services, to gain access to cluster metrics.

At this point, even if the cluster monitoring pods were up and running, there was no access to Prometheus or Grafana from outside the lab. To overcome this issue, the next step was to expose Prometheus and Grafana as a NodePort service.

Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service [15]. Services allow your applications to receive traffic. Services can be exposed in different ways by specifying a type in the ServiceSpec:

- ClusterIP (default) - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.
- NodePort - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>. Superset of ClusterIP.

- LoadBalancer - Creates an external load balancer in the current cloud (if supported) and assigns a fixed, external IP to the Service. Superset of NodePort.
- ExternalName - Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com) by returning a CNAME record with its value.

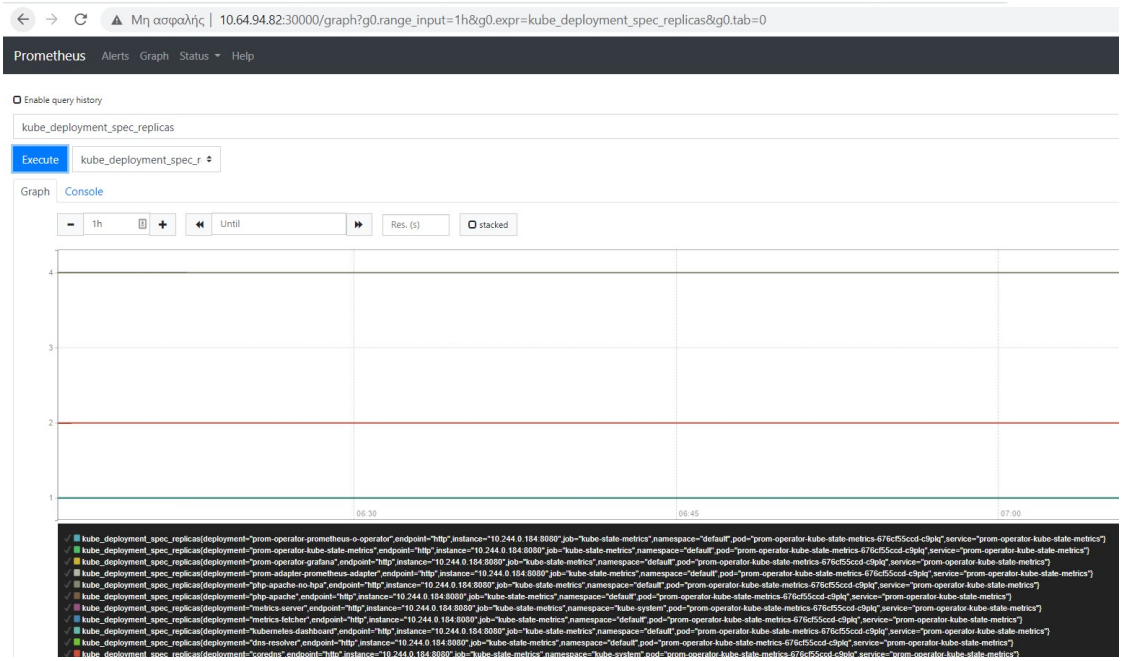


Figure 4.3: Accessing Prometheus using the NodePort service.

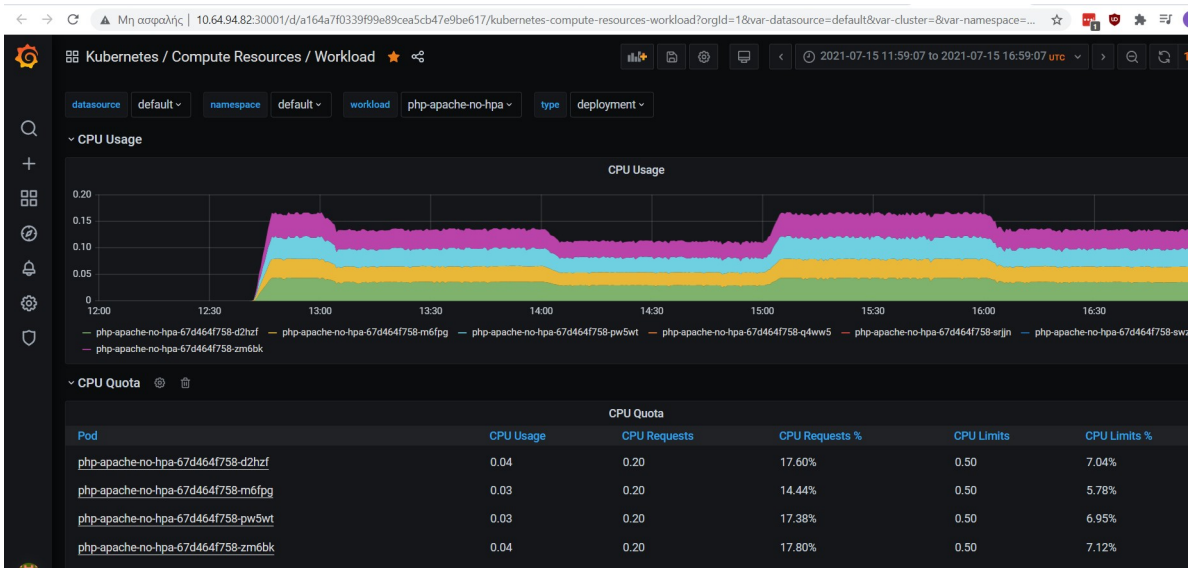


Figure 4.4: Accessing Grafana Dashboards using the NodePort service.

Chapter 5

Intelligent Load-Balancing

5.1 Introduction

Our approach of the problem is consisted of two major parts. The first part is the implementation of a round-robin load-balancing algorithm, which, knowing the IP addresses of the copies of the replicas of the application, assigns the incoming requests to each replica. The number of replicas can be passed as an external parameter, which makes the algorithm able to function with a variable number of replicas, thus in combination with the automatic scaling of HPA. The second part is the adaptation of the ”Water Filling” [16] method to a load-balancing algorithm. As in the previous case, this algorithm can also function with a non-fixed replica set.

What follows is a detailed explanation of both of the approaches, an analysis of the ”Water Filling” method, the codes of the algorithms and an explanation of some additional Services that were necessary to be implemented and run in the background to provide the algorithms with the appropriate data.

5.2 Round-Robin Approach

The round-robin policy is implemented by the following python code:

```
import os
import sys
import collections
import requests
```

```
import time
subnets_file = ”IPs.txt”
with open(subnets_file, 'r') as f:
    subnets = collections.deque(f.read().splitlines())
res = requests.get(”http://” + subnets[0])
print(subnets[0], res)
subnets.rotate(-1)
with open(subnets_file, 'w') as f:
    for s in subnets:
        f.write(”%s\n” % s)
```

Code Explanation:

Let's assume that the file ”IPs.txt” contains the IP addresses of the replicas of the application (how this happens will be explained in the final section). At first, a file manipulation is done to extract each IP in a python-friendly data type, in our case, a list. Then, a GET request is performed on the first IP, the result is saved and printed and then the list is rotated by sending the first entry to the end. Finally, the rotated list is saved.

5.3 The ”Water Filling” Method

The water-filling algorithm is a technique used for allocating optimal power among different channels in multicarrier schemes. It provides the optimality for the channels with Additive White Gaussian Noise and intersymbol interference (ISI). The algorithm is known as water filling as we think of the communication medium as if it was some kind of water container with a weird kind of shape and having irregular or asymmetrical bottom. Each available channel is then a portion of the container having its own depth. To allocate power we imagine pouring water into this container. The amount of water depends on the desired maximum average transmit power. Once the container is full up to the top with water, the maximum quantity of water is present in the farthest portion of the container. This implies there is allocation of more power to the channels with the most favorable SNR. The ratio allocation to each channel varies nonlinearly with the maximum average transmit power. So if we have a fixed transmit power we can allocate it optimally to the different transmit channels using water filling algorithm. [17]

The water-filling problem can be abstracted and generalized into the following problem:

given $P > 0$, as the total power or volume of the water; the allocated power and the propagation path gain for the ith channel are given as s_i and a_i respectively, $i = 1...K$; and K is the total number of channels. Let $\{a_i\}_{i=1}^{K}$ be a sorted sequence, which is positive and monotonically decreasing, find that

$$\begin{array}{ll} \max_{\{s_i\}_{i=1}^{K}} & \sum_{i=1}^{K} \log\left(1 + a_i s_i\right) \\ \text{subject to:} & 0 \leq s_i, \forall i \\ & \sum_{i=1}^{K} s_i = P \end{array} \quad (1)$$

Since the constraints are that (i) the allocated power to be nonnegative; (ii) the sum of the power equals P, the problem (1) is called the water-filling (problem) with sum power constraint. To find the solution to problem (1), we usually start from the Karush-Kuhn-Tucker (KKT) conditions of the problem, as a group of the optimality conditions, and derive the system (2) below from the KKT conditions,

$$\begin{cases} s_i = \left(\mu - \frac{1}{a_i}\right)^+, \text{for } i = 1, \ldots, K \\ \sum_{i=1}^{K} s_i = P \\ \mu \geq 0 \end{cases} \quad (2)$$

where $(x)^+ = \max\{0, x\}$. μ is the water level chosen to satisfy the power sum constraints with equality $\left(\sum_{i=1}^{K} s_i = P\right)$. The solution to (2) is referred as a solution of the Conventional Water Filling problem (1). It can be seen that the implied system (2) has been used to find the optimal solution. The existence of its Lagrange multipliers and the implication mentioned above determine that enumeration can be utilized to find the water level μ. [16]

5.3.1 A ”Water Filling” Load-Balancer

Below is presented the mathematical formulation and solution of the problem, based on the water filling method:

$N : Containers$

$P = (P_1, P_2, \ldots, P_N) =$ Number of requests per container (P_i)

$f_i = \log(1 + \gamma_i * P_i)$

We want to maximize

$$\sum_{i=1}^{N} f_i P_i$$

such that

$$\sum_{i=1}^{N} P_i < P_{max} \qquad P_i \leq P_{\max}$$

Alternatively, we can minimize

$$-\sum_{i=1}^{N} f_i P_i$$

such that

$$\sum_{i=1}^{N} P_i < P_{max} \qquad P_i \leq P_{\max}$$

Thus, we have:

$$\begin{aligned}
L(P,\lambda,\mu) &= -\sum_{i=1}^{N}\log(1+\gamma_i\cdot P_i) + \lambda\sum_{i=1}^{N}(P_i - P_{\max}) + \sum_{i=1}^{N}\mu_i(-P_i) \\
&= -\sum_{i=1}^{N}\log(1+\gamma_i P_i) + \lambda\sum_{i=1}^{N}(P_i - P_{\max}) - \sum_{i=1}^{N}\mu_i P_i \\
\frac{\partial L(P,\lambda,\mu)}{\partial P_i} = 0 &\Rightarrow -\frac{1}{1+\gamma_i P_i}\cdot\gamma_i + \lambda - \mu_i = 0 \Rightarrow \mu_i = -\frac{\gamma_i}{1+\gamma_i P_i} + \lambda \qquad (1) \\
\mu_i P_i = 0 \quad \forall i \in \{1,2,\ldots,N\} &\overset{(1)}{\Longrightarrow} (-\frac{\gamma_i}{1+\gamma_i P_i} + \lambda)\cdot P_i = 0 \overset{P_i>0}{\Longrightarrow} 1+\gamma_i P_i = \frac{\gamma_i}{\lambda} \Rightarrow \\
&\boxed{P_i^* = \frac{1}{\lambda} - \frac{1}{\gamma_i}} \square
\end{aligned}$$

5.4 The "Water Filling" Approach

The "water filling" - based policy is implemented by the following python code:

```
import sys
import collections
import requests
import numpy as np
import random
import subprocess


subprocess.call(['bash', 'config.sh', sys.argv[1]])
subnets_file = "IPs.txt"
```

```
with open(subnets_file, 'r') as f:
    subnets = collections.deque(f.read().splitlines())

N = len(subnets)  # number of containers
if (N < 1):
    exit(-1)

helper = open("helper.txt", "r")
names = []
ips = []

for line in helper:
    if line.strip():
        cols = line.split()
        names.append(cols[0])
        ips.append(cols[5])
helper.close()

# creating a dictionary looking like IP:REPLICA
dict = {ips[i]: names[i] for i in range(len(ips))}

t = 0
Time = 100000
p = np.zeros((N, Time))
pm = np.zeros((N, Time))
pma = np.zeros((1, Time))
lamda = np.zeros((1, Time))
epsilon = 0.01
lamda[0, 0] = 1
reqs = [0, 0, 0, 0]
gamma = 0
Pav = 5
```

```
for t in range(Time-1):
    for i, j in zip(range(N), subnets):
        val = dict[j]
        gamma_res = requests.get(
        'http://10.64.94.82:30000/api/v1/query?query=(rate(
        container_memory_usage_bytes{
        pod=~"php-apache-no-hpa.*",container="",
        service="prom-operator-prometheus-o-kubelet"}[1m]))')

        gamma_res = gamma_res.json()

        for k, c in zip(gamma_res["data"]["result"], range(N)):
            if(k["metric"]["pod"] == val):
                gamma = float(gamma_res["data"]["result"][c]["value"][1])
                break

        if(not gamma):
            gamma = random.random()
        if ((1/lamda[0, t]) > (1/gamma)):
            p[i, t] = max((1/lamda[0, t])-(1/gamma), 0)
        else:
            p[i, t] = 0
        pm[i, t+1] = ((t/(t+1))*pm[i, t])+((1/(t+1))*p[i, t])

    for j in range(N):
        reqs[j] = p[j, t]

    min_val = min(reqs)
    idx = reqs.index(min_val)
    res = requests.get("http://" + subnets[idx])
    print(subnets[idx], res)

    pma[0, t+1] = ((t/(t+1))*pma[0, t])+((1/(t+1))*(np.sum(p[:, t])))
    lamda[0, t+1] = lamda[0, t]+epsilon*(np.sum(p[:, t])-Pav)
```

```
subprocess.call(['bash', 'traffic.sh'])  #different bursts of traffic
```

Code Explanation:

At first, a subprocess is created to execute the config.sh script (the config.sh and the traffic.sh script will be explained in the following section). Once again, the file ”IPs.txt” contains the IP addresses of the replicas of the application. After that, we need to create a correspondence between each replica’s name and IP. To do that, we need to create a dictionary in the form of IP:REPLICA. This is done by creating a helping file (helper.txt). For now, we consider this functionality as a black box to stay on the point. So, the ”dict” dictionary contains every pair of IP and Replica. Then, some necessary variables need to be initialized according to the ”water filling” method. The ”p” matrix holds the Processor Allocation values per Container, The ”pm” matrix holds the Mean Processor Allocation values per Container, while the ”pma” matrix holds the Mean Processor Allocation values over all Containers.

After the initialization, the main algorithm follows. We execute a query to Prometheus API, which runs at 10.64.94.82:30000, to obtain the metric ”container_memory_usage_bytes” for all the replicas and we convert the result to a JSON file. The Python language interprets this file as a complicated dictionary. At this point, we need to obtain the metric value for the current replica based on the ”dict” dictionary and assign its value to the gamma variable. At the beginning, this metric may be zero so we assign a random number in (0,1) to gamma. We then set the p[i,t] value according to the water filling load-balancer, as formulated in the previous section. The replica that will handle the incoming request, is the min(p[i, t]) value, meaning that it is the least utilized replica. Finally, the pma and lamda variables are set and the traffic.sh script is called to create different bursts of traffic for the next loop.

5.5 Background Services

Now, let’s take a look at all the scripts and services that need to run to provide essential data to our algorithms.

5.5.1 DNS-Resolver Service

It was considered necessary to have a custom DNS Service, which would return all the IP addresses of the replicas of the application. This DNS was implemented using Flask, as a

Kubernetes deployment.

The Python Code:

```
import os
import sys
import socket

from flask import Flask,redirect, request
app = Flask(__name__)

@app.route('/')
def helper():
    domain = request.args["domain_name"]
    ips = socket.gethostbyname_ex(domain)
    return HELLO_HTML.format(ips)

HELLO_HTML = """
    <html><body>
        Domain Name resolved!<br>
        {0}
    </body></html>"""

if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

The YAML file:

```
apiVersion: v1
kind: Service
metadata:
  name: dns-resolver-service
spec:
  selector:
    app: dns-resolver
  ports:
  - protocol: "TCP"
```

```
    port: 6000
    targetPort: 5000
  type: NodePort

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dns-resolver
spec:
  selector:
    matchLabels:
      app: dns-resolver
  replicas: 1
  template:
    metadata:
      labels:
        app: dns-resolver
    spec:
      containers:
      - name: dns-resolver
        image: dns-resolver:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 5000
```

5.5.2 Metrics-Fetcher Service

We implemented a Service that expose all Prometheus metrics to an HTML page, again using Flask.

The Python Code:

```
import os
import sys
```

```
import socket
import requests

from flask import Flask,redirect, request
app = Flask(__name__)

@app.route('/')
def helper():

    PROMETHEUS = 'http://0.0.0.0:30000/metrics'
    metrics = request.args["metrics"]
    response = requests.get(PROMETHEUS + '/api/v1/query',
    params={'query': metrics})

    results = response.json()['data']['result']

    return HELLO_HTML.format(response, results)

HELLO_HTML = """
    <html><body>
        Metrics fetched!<br>
        {0}<br>
        {1}
    </body></html>"""

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5001)
```

The YAML file:

```
apiVersion: v1
kind: Service
metadata:
  name: metrics-fetcher-service
spec:
  selector:
```

```
    app: metrics-fetcher
  ports:
  - protocol: "TCP"
    port: 6001
    targetPort: 5001
  type: NodePort

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: metrics-fetcher
spec:
  selector:
    matchLabels:
      app: metrics-fetcher
  replicas: 1
  template:
    metadata:
      labels:
        app: metrics-fetcher
    spec:
      containers:
      - name: metrics-fetcher
        image: metrics-fetcher:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 5001
```

5.5.3 Scripts

traffic.sh

This script sends a request to our custom DNS server, retreives the IP addresses of the replicas and stores them to a file. Then, manipulates the file to extract them in a friendly form

and stores the desired output to a final file. After we get the IP addresses, we needed to create different bursts of traffic, so, we did a trick using the date. We extracted the time digit from the date (e.g. from Sun Sep 19 08:19:05 UTC 2021 would be 08), performed the modulo operation and set different seconds of sleep based on the result. Finally, the script executes the python file implementing Round-Robin algorithm.

The Code:

```
#!/bin/bash

curl 10.64.94.82:30859/?domain_name=$1 > helper.txt
grep -o
'[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}'
helper.txt > IPs.txt

while true;
do
        dat=$(date +"%H")
        res=`echo "$((dat % 3))" | bc`

        if [[ $res -eq 0 ]]
        then
            secs=0.2
        elif [[ $res -eq 1 ]]
        then
            secs=0.35
        else
             secs=0.5
        fi
        sleep $secs
        python3 RR_script.py
done
```

config.sh

The config.sh and traffic.sh mentioned in the water filling load balancer are basically the forementioned code divided in two parts.

config.sh

```
#!/bin/bash

kubectl get pods -o wide | grep "php-apache-no-hpa" > helper.txt
curl 10.64.94.82:30859/?domain_name=$1 > file.txt
grep -o
'[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}'
file.txt > IPs.txt
```

traffic.sh

```
#!/bin/bash

dat=$(date +"%H")
        res=`echo "$((10#$dat % 3))" | bc`
        if [[ $res -eq 0 ]]
        then
            secs=0.2
        elif [[ $res -eq 1 ]]
        then
            secs=0.35
        else
             secs=0.5
        fi
        sleep $secs
```

5.6 Results

For the sake of the experiment, we considered that the number of replicas is static (4), with each replica running on a different cluster node. However, the code can be slightly altered to

function with variable number of replicas. The application running is an Apache Server that responds with an ”OK!” (Code [200]) message if it receives a request. The duration of the experiment is about 2 hours and 50 minutes.

The Round-Robin algorithm:

Figure 5.1: RR algorithm performance

The water filling algorithm:

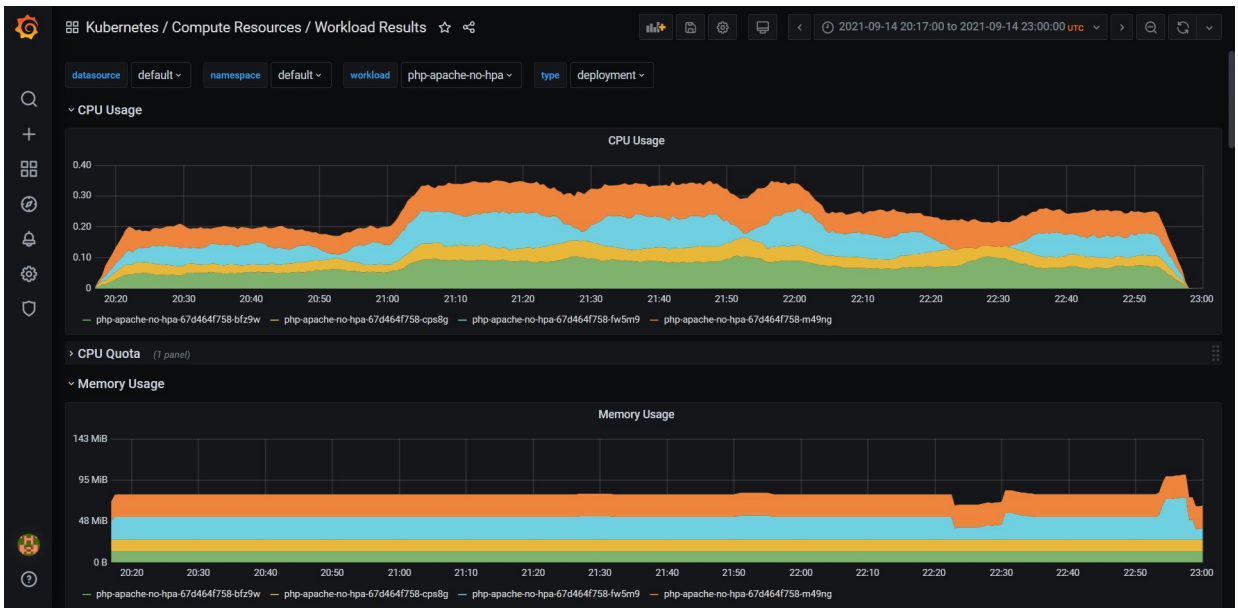


Figure 5.2: Water Filling algorithm performance

The Round-Robin algorithm shows a predictable behaviour. The 4 memory shares and the CPU usage for every replica are perfectly equal. Although it may seem that the Round Robin is the perfect load-balancer, it may not be the perfect choice. Let's say that our 4 replicas need to handle a total of 100 requests. If the load balancer is a RR algorithm, then each replica (each node) will handle 25 requests. But what will happen if suddenly a 5th node joins the cluster? Naturally, the RR will continue to assign requests equally between the nodes but, eventually, that 5th node will have handled fewer requests in total than the other nodes.

Now let's see why the water filling algorithm is a significantly superior load balancer, as far as the CPU Usage is concerned. If we zoom in a little bit in the previous screenshots, it is clear that a greater node utilization is achieved using the water filling load balancer.

Note: If you pay close attention, you will see that a replica crushed and was down for some time (right after 22:20). This did not messed up the algorithm, which continued to operate using 3 replicas until the Kubernetes re-started the 4th replica.

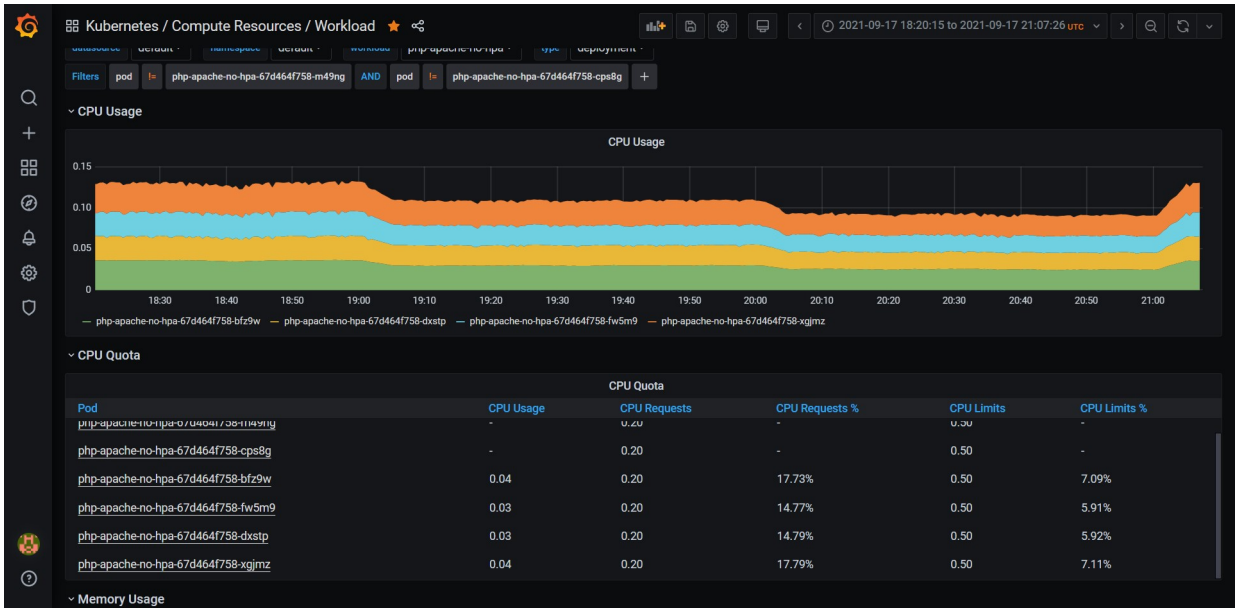


Figure 5.3: RR CPU Usage

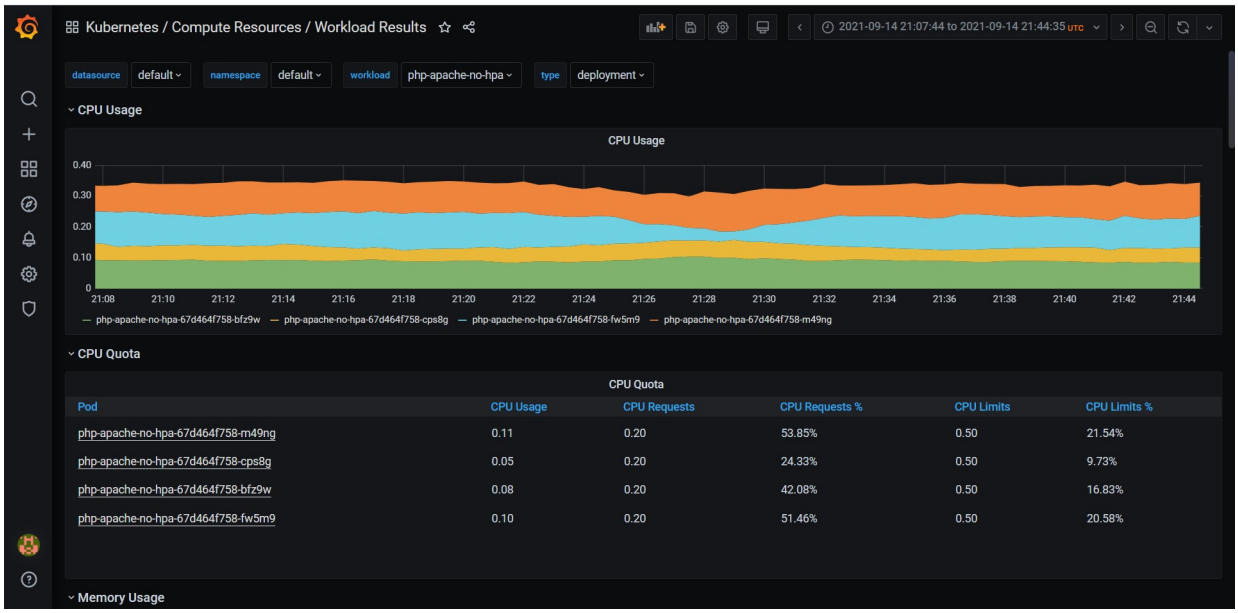


Figure 5.4: Water Filling CPU Usage

Chapter 6

Conclusions

6.1 Synopsis and Conclusions

In this thesis, we developed an intelligent load balancing mechanism in the Kubernetes framework, based on the water filling method. Initially, the problem was formulated and solved from a mathematical view, which helped us with the precise development of our algorithm. We compared our algorithm to the classic Round Robin, which we also implemented. By taking advantage of Kubernetes framework capabilities, we considered 4 replicas, running on different nodes, as a specification capable to prove our hypothesis.

The load-balancing algorithm that we implemented managed to significantly increase the CPU Utilization of our worker nodes, meaning that more user requests get executed, thus increasing the overall cluster performance. We also proved that our algorithm was able to properly handle common Kubernetes errors, like a pod crashing. To monitor our cluster and collect our results we used the Prometheus and Grafana monitoring tools, which provide interactive UI and real-time measurements.

After completing the experiment, we were curious to examine whether the algorithm that we implemented could be used as an energy-efficient solution. By examining the request allocation during runtime, we saw that the algorithm's behaviour was mostly to ”flood” a node with requests before moving on the next. That made us realize that if we could, somehow, deactivate or turn-off the nodes that are not going to be used in the near future, our algorithm could be considered as an energy-efficient solution. We would expect even better results if some kind of prediction method was integrated.

6.2 Future Extensions

There are some ideas that are considered as future extensions of our work. A simple, but interesting, step further would be to alter the code so that the algorithms could operate with a non-permanent number of replicas. All that needs to be done is periodically send requests to the DNS server to update the IPs.txt file to inform the algorithm about new replicas. In this way, our algorithm can be combined with the Kubernetes HPA, by altering the metrics limits and changing the number of pods while simultaneously performing intelligent load balancing.

We are also intrigued to find out how our algorithm performs compared to other known scheduling algorithms like Weighted Round Robin or Priority Scheduling, configured to run on a Kubernetes cluster.

Bibliography

[1] "What are microservices?." https://avinetworks.com/what-are-microservices-and-containers.

[2] "Kubernetes docs." https://kubernetes.io/.

[3] "Pod." https://cloud.google.com/kubernetes-engine/docs/concepts/pod.

[4] "Load balancer in kubernetes." https://www.sumologic.com/kubernetes/load-balancer/how-to-use-an-external-load-balancer-in-kubernetes/how-to-use-an-external-load-balancer-in-kubernetes.

[5] "Nitlab." https://nitlab.inf.uth.gr/NITlab/.

[6] "Nitos." https://nitlab.inf.uth.gr/NITlab/nitos.

[7] "Why should you use microservices and containers?." https://developer.ibm.com/articles/why-should-we-use-microservices-and-containers/.

[8] "Container orchestration." https://avinetworks.com/glossary/container-orchestration/.

[9] "What is kubernetes." https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[10] "Horizontal pod autoscaler." https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[11] "Prometheus introduction." https://prometheus.io/docs/introduction/overview/.

[12] "Grafana." https://dzone.com/articles/how-to-use-grafana-for-technical-monitoring-in-sof.

[13] "Prometheus adapter for kubernetes metrics apis." https://kb.novaordis.com/index.php/Prometheus_Adapter_for_Kubernetes_Metrics_APIs.

[14] "Prometheus metrics based autoscaling in kubernetes." https://www.metricfire.com/blog/prometheus-metrics-based-autoscaling-in-kubernetes/.

[15] "Using a service to expose your app." https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/.

[16] P. He, L. Zhao, S. Zhou, and Z. Niu, "Water-filling: A geometric approach and its application to solve generalized radio resource allocation problems," *IEEE Transactions on Wireless Communications*, vol. 12, no. 7, 2013.

[17] H. Kour, R. K. Jhaa, and S. Jain, "A comprehensive survey on spectrum sharing: Architecture, energy efficiency and security issues," *Network and Computer Applications*, vol. 103, pp. 29–57, 2018.