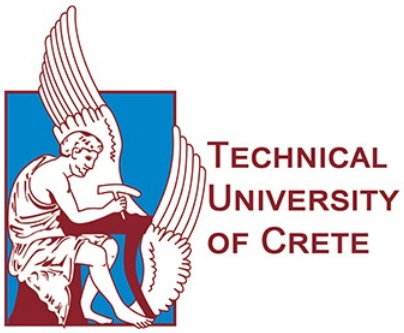
Technical University of Crete
School of Electrical and Computer Engineering



Support Auto-scaling in DockerSwarm

Author:
Michail S. Alexiou
michalis.alex@yahoo.com

Committee:
Professor Euripides G.M. Petrakis (Supervisor)
Associate Professor Michail G. Lagoudakis
Dr. Stelios Sotiriadis (Assistant Professor at Birkberk, UL, UK)

June 2019

Abstract

Docker Swarm is an open-source container orchestration platform natively managing a cluster of Docker engines. Docker Swarm utilizes the same command line from Docker to create a cluster of Docker engines (referred to as a swarm), deploy application services to a swarm, scale up or down containers running inside swarm nodes, and manage swarm behavior in general. Yet for all its advantages, Docker Swarm currently lacks the necessary tools for supporting automatic scaling of resources inside each swarm, resulting in a rather static environment incapable of adapting to the requirements of an online application. Unlike other competing tehcnologies, such as Kubernetes or Amazon EC2, which support auto-scaling in a containerized environment, to the best of our knowledge, no such solutions exist for Docker Swarm. This is exactly the problem our work is dealing with. Building upon Docker Swarm, we proposed Elixir, an autonomous agent that runs on top of Docker Swarm (i.e. the infrastructure provider side) and is capable of managing multiple and different online applications for each provider, monitoring the running worker nodes (Virtual Machines) required by each application, and automatically scaling up or down the used resources (CPU, Disk, etc.) on demand when necessary. The decisions for scaling are determined by the infrastructure provider and are based on resources metrics measured in real time such as CPU, or memory usages (or a combination of the above), during monitoring. Elixir contributes to achieving fault tolerance and high availability for a Docker Swarm system managing multiple applications rather than a single application. Elixir's node scaling approach is horizontal meaning that rather than reconfiguring the worker nodes of each application with larger or smaller characteristics (as would be in the case of vertical scaling), it will add/delete worker nodes with the same characteristics to the application's swarm. We run several experiments based on a simulated, but realistic use case scenario. The experimental results demonstrate that the implementation of Elixir in a system managing an application, has a significant impact on the availability and response time of an application charged constantly with an increasing workload.

Acknowledgements

I would like to express my sincere gratitude towards my supervisor, Prof. Euripides Petrakis, for his constant encouragement, guidance and support during my efforts to complete this thesis. I am also grateful to Prof. Michail Lagoudakis and Prof. Stelios Sotiriadis for serving on my thesis committee. Special thanks should go to my fellow members of the Kouretes Team and the Fi-Star lab. I would like to thank Spyros Argyropoulos, as well, for his helpful and valuable advises. Finally, I would like to thank my all friends for supporting me throughout my years at the Technical University and my entire family, especially my father Stelios, my mother Aleka, my sister Maria, and my cousin Lina, for teaching me the value of effort and to always aim for the stars.

Contents

1 Introduction

One of the many advantages that came along with the introduction of cloud computing is the idea of virtualization. Virtualization software made applications easier and faster to deploy by providing an abstraction layer that decouples the physical hardware from the environment that runs on top of it and by supporting features like isolation and resource customization. In the early days of cloud computing, application providers would have to allocate a great amount of either physical or virtual resources, in order to make sure that their infrastructures have the resources to support possible increased demands of the application usage at all times. However, service provision proved to be very costly for both clients and providers, since the client would have to pay for a fixed amount of resources and the provider would have to allocate a maximum amount of resources regardeless of application needs, instead of using them more beneficially with an adaptive business plan in mind. 'The market demands are never static, as they shift based on people's needs at that time, all while resources flow in and out of availability.' [1] In order for a business to stay competitive in these circumstances, it must be able to change compute resource provision decisions fast. That situation led the need for a more adaptive approach and to the idea of resource scalability.

Scalability is defined as the ability of a computer infrastructure to handle a growing amount of work for systems, networks, or processes in a graceful manner, with the least possible down time or (even better) no down time at all, in order to maximize the availability of running applications and the usage of the their computing resources at all times. Based on the steps that a scaling method follows, it can be further distinguished in two individual categories, referred to as vertical scaling and horizontal scaling. Vertical scaling is defined as the addition of more resources (Disk, Memory, CPU cores) to already existing worker nodes, by reconfiguring the size of these nodes with new (higher) resource limits. On the other hand, horizontal scaling refers to adding more resources to the system, by adding new worker nodes to the system instead. Vertical and horizontal down-scaling are the opposites of the two previously mentioned methods. A scalable Web application should be capable of serving an increasing or decreasing number of user requests (HTTP requests) with little or no down time depending the scaling method performed by the system. Scalablity often appears as an infrastructure problem, where the lack of compute resources can potentially lead to performance degradation of the system, in case of an increasing workload.

Auto-scaling, is a natural evolution of the above idea and refers to mechanisms capable of adapting to the resources depmands of running applications in real time. A problem inherent to auto-scaling relates to the monitoring for resources (per application) in real time and to taking independently (i.e. without human

[1] `https://www.touchsupport.com/what-is-scalability-and-why-does-it-matter-to-your-business/`

intervention) decision on when to scale resources up or down. There are already some tools in existence that support auto-scaling, for example Kubernetes POD Auto-scalers[2] automatically scales the number of PODS in a given cluster of worker nodes. The work in [2] shows how to support auto-scaling in a native Kubernetes environment. The work shows how a cluster of worker nodes can be adapted to the workload with the addition or deletion of new worker nodes (not just PODS). Similar approaches have been adopted by major cloud providers such as Amazon EC2 and Google GKE. Kubernetes is a mechanism for orchestrating containerized applications, Applications are deployed as PODS with each POD being a Docker environment. However, Kubernetes replaces some of the higher-level technologies that have emerged around Docker, such Docker Swarm, an orchestrator born out of Docker. 'It's still possible to use Docker Swarm instead of Kubernetes, but Docker Inc. has chosen to make Kubernetes part of the Docker Community and Docker Enterprise editions going forward.'[3]. The direct result of this situation, is that a promising technology such as Docker Swarm has been left devalued and unexploited.

1.1 Motivation

In Andre Bondi's work we learn about the term "load scalability"[1]. He suggests that a system has load scalability, if it is capable of functioning without any drastic changes in its performance, nor any delays, nor counterproductive resource usage at any size of workload, while optimizing its resource consumption. According to A.Bondi some of the factors that can have a negative effect over load scalability include "the scheduling of a shared resource", as well as "inadequate exploitation of parallelism". These statements suggest to us, that a well functioning scalable system depends greatly on its load balancer (or scheduler) in order to achieve balance in distributing the load among the worker nodes of the system.

In the following, we perform an exemplary test on a simple swarm implementation illustrated at Fig.1. This swarm has 1 manager online and no workers. Our goal is to showcase a scenario, where a static environment (in this case the swarm with 1 manager) becomes in need of scalability. A swarm manager always acts both as a manager and worker at the same time. Inside this manager node runs a containerized Web application. This application finds the shortest path for given edge to every other edge in a predefined graph with the use of Dijkstra's algorithm, a simple yet CPU stressing application. This single node has 1 CPU, 20GB Disk and 2GB RAM as available resources, is a small flavor, and has _ubuntu_18.04 as its base image. We perform this test for 500 concurrent users with the help of the Apache Bench load testing tool. Fig.2 illustrates the swarm's response time while the system serves an increasing number of requests. Fig.3 illustrates the manager's CPU usage also while the requests are being performed. During this experiment, we notice that the response times increase

[2]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[3]`https://www.touchsupport.com/what-is-scalability-and-why-does-it-matter-to-your-business/`

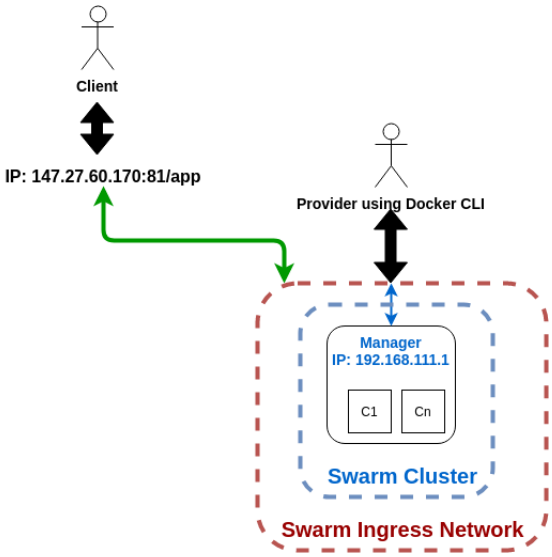


Figure 1: A simple swarm example, with 1 manager.

drastically for more than 50,0000 requests. We conclude that this significant rise in the swarm's response time occurs while the manager's CPU usage almost reaches 60%. So it is safe to assume, that in case we used a higher number of concurrent users, the manager's CPU usage would eventually overload, making the whole system non responsive at the end. This potential problem could be avoided by making the swarm scalable.

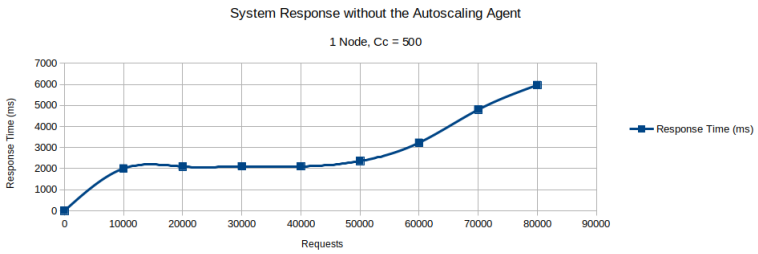


Figure 2: System's response with 1 node and Cc = 500

In this thesis, we design and implement a solution to handle gracefully all the increasing workload automatically and turn the idea of auto-scaling in Docker Swarm into a realistic option for users.

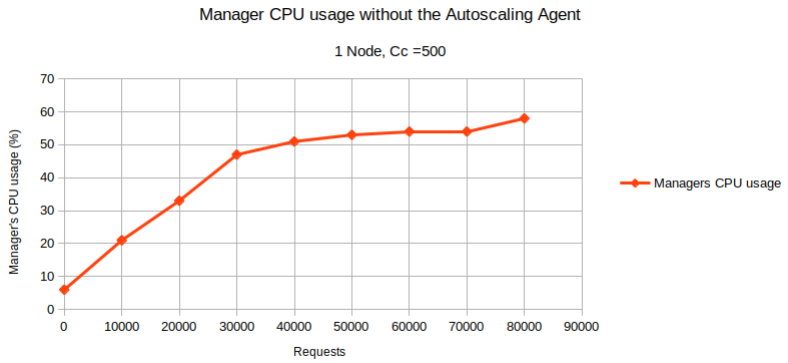


Figure 3: Manager's CPU usage with 1 node and Cc = 500

1.2 Problem Definition

Docker Swarm offers the advantage of scaling up or down services fast and easily, in the form of scaling containers running these services, with the use of a specific command in the Docker CLI by the provider of a Web application (Docker client). As of yet, however, there is no official automatic mechanism that can deal with this problem without any human interference. And even if there was one, scaling up or down the containers deployed for a service, would contribute absolutely nothing to the overall performance of the system. Containers are, by definition, processes that have access to the entirety of a server's resources, and each time they are deployed, they will use as much percentage of these resources (CPU or Disk) as they need to, in order to carry out their task, in this case keep a Web application up and running for the public. There are ways that can help the user limit the containers' access to specific CPU cores of the server, however these methods are rather complicated and counterproductive, if we hope to keep the system simple, and the virtual machines used as workers, lightweight.

The only reliable way to actually deal with the increasing or decreasing incoming workload in the system, would be to scale up or down the system's available resources, by implementing either some form of vertical or horizontal scaling in the worker nodes of the system. Even though, Docker Swarm supports adding more worker nodes to a swarm cluster dynamically, there is still no official mechanism that could complete this task automatically, and it is a very difficult task, consisting of many different steps, that requires a lot of effort,time and work to be carried out by the provider of infrastructure.

These are the problems that our work is dealing with, the problems of automatic service scalabily, automatic infrastructure scalability, and system monitoring on a virtualized cloud environment managed by Docker Swarm.

1.3 Solution

In this work we design and implement an autonomous agent, capable of monitoring multiple swarms for a single infrastructure provider, all running different Web applications, and also capable of automatically provisioning or deleting the extra worker nodes in these swarm clusters, based on the CPU or Disk usage metrics received during monitoring. In Fig.9 we see an example of how the Elixir agent is implemented into the system, as a middle layer between the OS and the Docker Swarm environment. In continuation of our previous example scenario with the simple swarm, let us see how would the situation proceed when Elixir is implemented into the system:

- Elixir gets notified that the CPU usage of the manager is too high on the particular swarm.
- Elixir initializes the procedures required for the creation of a new worker node.

- The new worker node is created by OpenStack, and has the Docker Engine, as well as the Zabbix agent installed in its OS, all done automatically by Elixir.
- Elixir scales up the containers running the application service and updates the load balancer.
- Elixir waits 1 minute and then continues its routine health checks until a new system threat arises on each of the swarms managed by it.

1.4 Contributions

Elixir is an agent operating a layer above the common Docker Swarm orchestrator installed on a Linux server. Therefore, all its contributions are complementing the already existing operations of Docker Swarm. The experimental results demonstrated that Elixir responds to the increasing (or decreasing) resource demands of each application leading to significantly faster response times compared to a non-auto scaled implementation. The contributions of Elixir can be listed as follows:

- Provide an official mechanism that can scale up and down services running on Linux servers (worker nodes) automatically, without the need of a human overseer to type the proper command each time on the Docker CLI.
- Provide an official mechanism that can scale up and down automatically worker nodes themselves (horizontal autoscaling resources), when it is required for the system to continue functioning properly, therefore providing high availability for the running Web applications.
- Add to the system the capability of monitoring the online Linux servers running these applications, and understand whether there is an internal error in the server or not, based on the received system diagnostic metrics, again without the need for human interference.
- Provide an official mechanism that can manage multiple applications at once for each provider.
- Increase safety and security for the system running these Web applications, with the use of a reverse proxy server node above all other worker nodes. This will ensure that the I.P. addresses of the worker nodes remain hidden from public view, and lay the ground for future versions of the system to be even safer from attacks, withe the use. This reverse proxy server node will also serve as a scheduler to properly manage the workload of the system, as well as a database center to gather all the necessary metrics concerning CPU usage, Disk usage, and availability of the worker nodes.

2 Background

In this chapter, we provide some valuable information about the knowledge background required for understanding our work, and the software tools that we deployed for the completion of this thesis. Section 2.1 describes briefly some basic concepts concerning virtualization, and how it can be distinguished into hardware virtualization and operating system virtualization, while sections 2.2 and 2.3 dive deeper into these concepts by describing OpenStack and Docker as tools for achieving hardware and operating system virtualization respectively. The next section (section 2.4) is about OpenFlow, a tool that we did not directly deploy into Elixir, but whose contributions as a SDN (Software Defined Network) tool, proved to be very beneficial for our work. Section 2.5 provides a few basic information about what an autonomous agent really is, and section 2.6 describes the monitoring tool Zabbix, and how it is implemented within our work. Finally, section 2.7 offers a few key information about the relational database MySQL, which we use to store all the metrics we receive, while monitoring our worker nodes, and section 2.8 describes the NGINX server, who serves both as a reverse proxy server, as well as a scheduler for our system.

2.1 Virtualization

"Virtualization provides a number of benefits. It enables a flexible allocation of physical resources to virtualized applications where the mapping of virtual to physical resources as well as the amount of resources to each application can be varied dynamically to adjust to changing application workloads. Furthermore, virtualization enables multi-tenancy, which allows multiple instances of virtualized applications ("tenants") to share a physical server. Multi-tenancy allows data centers to consolidate and pack applications into a smaller set of servers and reduce operating costs. Virtualization also simplifies replication and scaling of applications. There are two types of server virtualization technologies that are common in data center environments—hardware-level virtualization and operating system level virtualization."[3][4][5]

2.1.1 Hardware Virtualization

"Hardware virtualization involves virtualizing the hardware on a server and creating virtual machines that provide the abstraction of a physical machine. Hardware virtualization involves running a hypervisor, also referred to as a virtual machine monitor (VMM), either on the bare metal server (Type-I Hypervisor) or above the existing OS of the server (Type-II Hypervisor). The hypervisor emulates virtual hardware such as the CPU, memory, I/O, and network devices for each virtual machine. Each VM then runs an independent operating system and applications on top of that OS. The hypervisor is also responsible for multiplexing the underlying physical resources across the resident VMs."[3]

[4] https://blog.netapp.com/blogs/containers-vs-vms/
[5] https://www.vgyan.in/type-1-and-type-2-hypervisor/

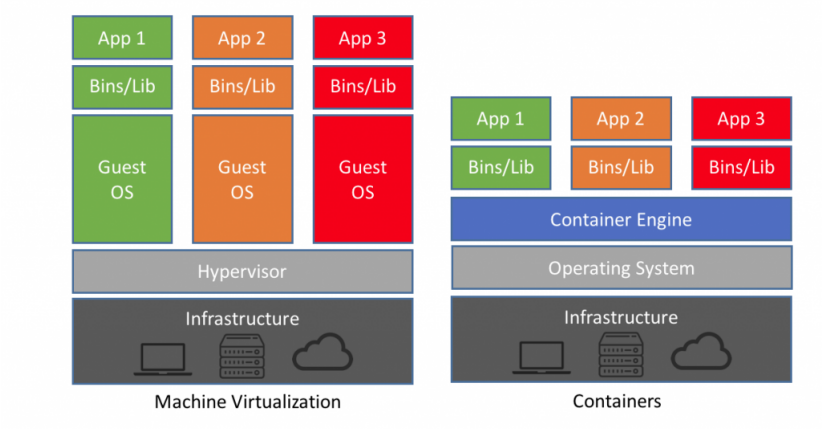


Figure 4: A virtualized system and a containerized system

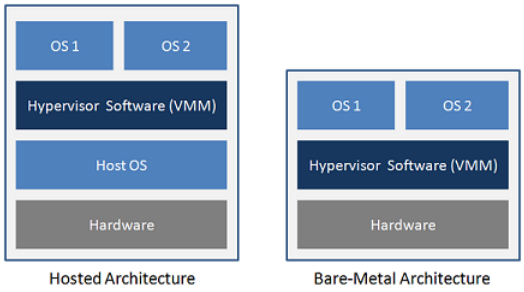


Figure 5: Structures of Type-I and Type-II hypervisors

"Modern hypervisors support multiple strategies for resource allocation and sharing of physical resources. Physical resources may be strictly partitioned (dedicated) to each VM, or shared in a besteffort manner. The hypervisor is also responsible for isolation. Isolation among VMs is provided by trapping privileged hardware access by guest operating systems and performing those operations in the hypervisor on behalf of the guest OS. Examples of hardware virtualization platforms include VMware ESXi, Linux KVM and VirtualBox." [3]

2.1.2 Operating System Virtualization

"Operating system virtualization involves virtualizing the OS kernel rather than the physical hardware. OS-level virtual machines are referred to as containers. Each container encapsulates a group of processes that are isolated from other containers or processes in the system. The OS kernel is responsible for implementing the container abstraction. It allocates CPU shares, memory and

network I/O to each container and can also provide file system isolation."[3]

"Similar to hardware virtualization, different allocation strategies may be supported such as dedicated, shared and best effort. Containers provide lightweight virtualization since they do not run their own OS kernels, but instead rely on the underlying kernel for OS services. In some cases, the underlying OS kernel may emulate a different OS kernel version to processes within a container, a feature often used to support backward OS compatibility or emulating different OS APIs."[3]

2.2 OpenStack

OpenStack is a cloud platform which began as a joint project between NASA and Rackspace. It was originally intended to be an open source alternative that has compatibility with the Amazon Elastic Compute Cloud (EC2) cloud offering. Today, OpenStack consists an important player in the cloud platform industry, as it continues to grow and gain adoption both in its open source community and the enterprise market. OpenStack has a very modular design, and each one of its modules controls a different resource that can be virtualized for the end user. Depending on the project, however, maybe not all modules are necessary each time, in order to use OpenStack's services successfully. OpenStack's key components consist of:

- **Dashboard:** A Web interface component provided with OpenStack.
- **Keystone:** Keystone functions as common authentication procedure across the cloud OS and can integrate easily with other authentication services such as LDAP. Keystone has to authority to grant the users access to the vast services and applications of OpenStack. Its authentication methods include standard credentials (username and password) or token-based logins.
- **Glance:** The component that manages the images. Once we're authenticated,there are a few resources that need to be available for an instance to launch. In order for the created server to be fully operational and useful, it requires an operating system, which it chooses from a registry of preinstalled disk images, ready to boot from. Glance serves as this registry within an OpenStack deployment.
- **Neutron:** Neutron is the components that manages the network in an OpenStack implementation. After the authentication of Keystone, and the provision of a disk image by Glance, the next step is to gain access to a virtual network and receive a functional IP address. Openstack uses Open vSwitch as an orchestration tool for the underlying virtualized network.
- **Nova:** Once there's an image, network, key pair, and security group available, an instance can be launched by Nova. Nova checks for the

availability of resources, and organises the spawning of the instance on a Virtual Machine.

- **Cinder:** It is the block storage management component. These storages are called volumes and can be created and attached to instances.
- **Swift:** It is the object storage management component. Object storage is a simple content-only storage system. Files are stored without the metadata that a block filesystem has. These are simply containers and files. The files are simply content. Swift has two layers as part of its deployment: the proxy and the storage engine.
- **Ceilometer:** It collects resource measurements and is capable of monitoring the cluster, originally used as a metering system for billing users.
- **Heat:** It is the orchestration component of OpenStack, which enables launching multiple instances that are intended to work together. Heat is, however, also compatible with AWS Cloud Formation templates and implements extra features in addition to the AWS CloudFormation template language.

2.3 Docker

Docker on a very basic level resolves the issue of an application working on some platform and not working on some different platform. In cases where the client tries to run multiple and different applications in the same server, most of the modules required for these applications don't work properly together. Each application has different dependencies, which most of the time prove to be conflicted. The same problem can arise in situations where the developer wishes to update one of these applications, causing many maintenance problem for the rest of the applications running on the server. Docker opts to resolve this issue of deployment, for it is a tool designed to make it easier to deploy and running applications with the use of containers.

2.3.1 Docker Advantages

- **Portability:** An application inside a docker container can run on any system and server, as long as it has installed Docker.
- **Composability:** Most modern Web applications are the combination of several different software components such as a server, a database, etc. Docker makes it easy to compose some of these elements either into a single container or multiple containers that can be maintained independetly.
- **Isolation:** With Docker every application works in isolation in its own container both from other applications running on the same system and from the underlying system as well.

- **Orchestration and Scaling:** Docker containers are lightweight solutions, and as such they are very easy to create multiple of them on a single server or scale them up easily across a cluster of servers.

2.3.2 Docker Shortcomings

- **Not Virtual Machines:** Containers are not the alternative of Virtual Machines. Even though containers provide a degree of isolation, they can not achieve the isolation levels of Virtual Machines, since each Virtual Machine runs in ts own instance of an OS. Containers are very similar to processes however.
- **Stateless:** If the provider kills the container running a Web application, and the starts it again, this appliction will not have stored any its previous information.
- **Not Microservices:** Containers can be used in the creation of microservices, but that does not make them microservices themselves. Each microservice application should follow the official microservice design.

2.3.3 Containers

Containers allow a developer to package an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as if it was one package. In order to create a container capable of running an application, first of all, we need a to create the proper Dockerfile, that will describe the necessary steps to create a docker image out of the code for an application (the code can be Java, Python, PHP, etc.). Docker images contain all the applications dependencies and requirement and docker containers are basically runtime instances of these docker images. In case the client wants to install his custom image on his online server and create a public Web application with this image's container, he first has to upload this image into DockerHub, an online repository where public images are stored in order to be used later.

2.3.4 Docker Swarm

A swarm is a collection of nodes, with the Docker engine installed, joined into a a single cluster. After the join, the provider can use the same Docker CLI that he used for controlling the services running in containers, but this time the commands will run at the from the manager of the swarm. The nodes in a swarm can be either physical or virtual, and they are distinguished into managers and workers. A physical node is a an electronic device operating as a server with finite resources, unless somebody adds a more advanced physical module, such as a CPU. A virtual node is a Virtual Machine running on top of a regular operating system and created by a hypervisor through virtualization of the infrastructure. Swarm managers are responsible for deploying a service in a swarm. During service deployment, containers are created and distributed

among the nodes (manager and workers) of the swam. The are 2 distrbution strategies and these are 'global' and 'replicated'. We can define distribution stategies with the –mode flag on the Docker CLI. During the 'global' mode, the manager places one task on each node that meets some predefined service placement constraints and resource requirements.[6] If the mode is not specified, then the system gets the 'replicated' mode by default. During the 'replicated' mode, the manager will automatically try to deploy the containers equally among all of the nodes. Swarm managers are the only machines in a swarm that can execute your commands, scale up or down services, and authorize other machines to join the swarm as workers. Workers are just there to provide computing resources to the system and do not have any authority. However managers can and will act as worker nodes as well every time, thus sharing the total workload.

2.3.5 Docker-Machine

Docker Machine is a tool that allows a provider to install Docker Engine (the Docker software) on virtual hosts, and manage these easily hosts with docker-machine commands. These Docker hosts can be either local Mac or Windows box, or on the providers company network, or data center, or on cloud providers like Azure, AWS, Digital Ocean and OpenStack. The Docker hosts themselves can be sometimes referred to as, managed "machines".

2.3.6 Docker SDK

Docker provides an API for interacting with the Docker daemon (called the Docker Engine API), as well as SDKs for Go and Python. The SDKs allow the provider to build and scale Docker applications and solutions quickly, without the use of the Docker CLI. Although the Docker SDK is not a common choice for developers to work with, mostly because of its lack , if studied in depth and used properly it can prove itself a valuable tool for controlling both Docker containerized applications and Docker Swarm node applications automatically.

2.4 OpenFlow

"Traditionally, networking hardwares from different vendors often have special configuration and management systems, which limits the interacting between routers and switches from different manufacturers. OpenFlow is a feasible solution this problem, by being an open programmable network protocol for configuring and managing network switches from various vendors. It enables us to offload the control plane of all the switches to a central controller and lets a central software define the behavior of the network. Thus network administrators can use OpenFlow software to manage and control traffic flow among different branded switching equipments."[7]. Initially a derivative technology of

[6] https://docs.docker.com/engine/swarm/

[7] http://www.fiber-optic-transceiver-module.com/openvswitch-vs-openflow-what-are-they-whats-their-relationship
html

OpenFlow, OpenVSwitch (open source virtual switch) started as a networking tool, with some of OpenFlows capabilities, most commonly used in SDN applications and especially in OpenStack. However, it evolved with the latest versions of OpenStack, and is now capable of providing the exact same services as OpenFlow, such as interconnection of virtual devices in the same host or between different hosts, which has been proven very helpful for our work.

2.5 Autonomous Agent

An autonomous agent is defined as a program or programmed device which can act without any human supervision or interference, most commonly in a remote location (e.g. on a remote server). "An autonomous agent can be seen as a system capable of interacting independently and effectively with its environment via its own sensors and actuators in order to accomplish some given or self-generated task(s)."[4]

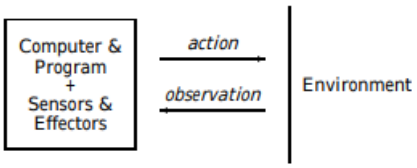


Figure 6: A simple autonomous agent structure as described in "Concepts and Autonomous Agents"[4]

"The term 'autonomous agent' is, as most terms in AI, ambiguously used. What one researcher would consider an autonomous agent, another refers to as a simulation program."[4]

2.6 Zabbix

Zabbix is an enterprise-class open source distributed monitoring solution, initially created by Alexei Vladishev, and currently maintained by Zabbix SIA. As a monitoring software, it keeps track of multiple parameters on clusters of servers, perfoming health checks and integrity checks. Zabbix supports a flexible notification mechanism that enables e-mail based alerts for users in case of disaster events, as well as excellent reporting and data visualization features, from the gathered and stored metrics. All of its reports and graphs are easily accessible via a web-based frontend interface, from any location.[8]

- **Data Gathering:** Zabbix items can perform calculations through arithmetic expressions, and create virtual data, whose values are periodically recalculated. The results are stored on the database of the Zabbix server,

[8] `https://www.zabbix.com/server_monitoring`

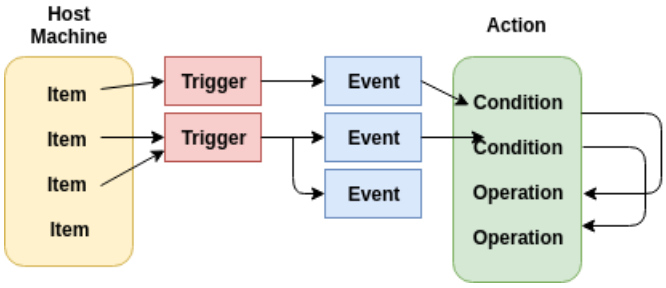


Figure 7: zabbix's workflow

and from there they are used to generate graphs, raise alarms, or send notifications to users. It is also possible to create custom agent checks. Any user can expand Zabbix agent's functionality by creating customs scripts in Perl or Python.

- **Problem Prediction:** There are predictive functions available as tool in Zabbix, that will process the incoming data in order to find trends and construct predictions of possible future system behavior. History data can also be analysed in order to gain some useful insight about future situations. We do not utilise these features in our current work, since they were unnecessary for our goal, however, they can prove very useful for future versions of Elixir, should we want to extend its capabilities. Zabbix also provides "hysteris" a very important function for Zabbix triggers, since it helps avoid data corruption (because of metrics fluctuating values), by putting the triggers on problem state only when the upper limit is reached and to normal state when the metrics are below the threshold value.

- **Configurable Alerting:** Zabbix offers the ability to send custom messages to notify the users depends on the information that each user is interested in. The notification may have such fields as date and time,host name, item value, trigger value, escalation history, etc. Each item gathers information about a specific resource from the server it is installed and a trigger will check whether the metric received from the item is within the permissible limits. The Fig.7 presents a simple workflow of Zabbix, which illustrates clearly the connection between items, triggers and actions.

- **Zabbix Web Frontend:** It contains real time graphing features, that enable custom or automatic graph generation from the data stored in the database of the Zabbix server. It also has the Global Dashboard, which provides many details about the monitored hosts.

- **Distributed Monitoring:** Zabbix supports a method of monitoring remote servers through Zabbix proxies, that will gather the necessary data and later forward their metrics to the Zabbix server. This can prove an

efficient way to simplify the monitoring of a worker node and improve the performance of the central Zabbix server.

- **Security and Authentication:** It supports multiple authentication methods such as HTTP basic authentication and LDAP authentication, as well as encryption between Zabbix separate components, like Zabbix servers and agents, in order to secure communications. The encryption can be either Cerficiate-based or pre-shared key-based encryption.

2.6.1 Zabbix Server

The Zabbix server constitutes the central component in the entirety of Zabbix software, that calculates triggers, generates graphs, sends notifications to users, and to which Zabbix agents or proxies report all their gathered information. The server contains a relational database in which all the statistical and operational information is stored, and can perform check on the network of servers, in order to know whether a monitored host is responsive or not.

"The functioning of a basic Zabbix server is broken into three distinct components, the Zabbix server, the Web frontend and the database storage. All of the configuration information for Zabbix is stored in the database, which both the server and the Web frontend interact with. For example, when you create a new item using the Web frontend (or API) it is added to the items table in the database. Then, about once a minute Zabbix server will query the items table for a list of the items which are active that is then stored in a cache within the Zabbix server. This is why it can take up to two minutes for any changes made in Zabbix frontend to show up in the latest data section."[9]

2.6.2 Zabbix Agent

A Zabbix agent should be installed on the monitoring target, in order to gather information about local resources (CPU, Disk, etc.) and applications successfully, all while offering some basic processing of these data. The agent later reports these metrics to the Zabbix server for additional processing. Zabbix agents are written in C language and are very efficient tools, mainly because of their use of native system calls. After installing the Zabbix agent on a server, the agent's configuration file must be updated, in order to include such information as the type of checks it performs (Active or Passive), as well as the IP of the Zabbix server.

Zabbix Agent Characteristics:

- **Small footprint and low resource:** The Zabbix agent can be run on multiple nodes with limited resources. With the installation of the Zabbin agent, also come a configuration file, which declares who is the

[9] `https://www.zabbix.com/documentation/4.2/manual/concepts/server`

central Zabbix server of this agent, as well as what kind of monitoring is the agent supposed to be doing.

- **Polling - Passive checks:** Zabbix servers requests periodically a value from the Zabbix agent, and the Zabbix agent will return the value after processing the request.
- **Trapping - Active checks:** The Zabbix agent requests from the Zabbix server a list of active checks that need to be performed and sends the results periodically.

2.6.3 Zabbix API

Zabbix has its own API, that provides access to almost all of Zabbix's available functions. The existence of such API, however, lays the ground for great customisation and further expansion of Zabbix's available services and functions. Some of the advantages of the Zabbix API can be listed as follows:

- **Easy Integration with Third Party Software:** The Zabbix API can be integrated easily with any third party software that is capable of making and receiving external calls, as opposed to other monitoring software tools such as JIRA, or Bugzilla. This third party software can also prove an important step towards creating custom based IT solutons.
- **Data Retrieval:** Zabbix stores in its database a plethora of information about the servers or systems it monitors, and not using this data efficiently would consist a great waste. With the help of the Zabbix API, however, users can use these information for deducting complex conclusions about the system's behavior, that otherwise would not be possible to obtain.
- **Mobile Applications:** Monitoring a modern IT environment requires constant attention, if the provider wants to remain competitive in his business. This is where mobile devices and implementations can be useful, and Zabbix API makes such implementations possible.

2.7 MySQL

MySQL is a relational database management system (RDBMS), used for storing and managing huge volume of data. This is called relational database because all the data is stored into different tables and relations are established using primary keys or other keys known as foreign Keys. A RDBMS is a software that:

- Enables the client to implement a database with tables, columns and indexes.
- Guarantees the referential integrity between rows of various tables.
- Updates the indexes automatically.
- Interprets an SQL query and combines information from various tables.

2.8 NGINX

2.8.1 Reverse Proxy Server

A forward proxy server works as an intermediate between client and server, hiding every time the identity of the client and offering him "anonymity". The clients traffic will be routed by the proxy and the proxy will make the requests to the server, so responds return to the proxy server and the proxy server forwards them back to the client.

A reverse proxy offers the opposite service, by hiding the IP of the server. A Web application can have multiple servers behind his reverse proxy. This reverse proxy server accepts requests from clients and forwards them to the backend servers, meanwhile the client remains completely unaware of which server he logged into or even which port he used (could be an unsafe port,different from 8080).

2.8.2 Load Balancing

A load balancer distributes incoming client requests among a group of servers, in each case returning the response from the selected server to the appropriate client. Load balancers are constitute a common deployment choice, every time a Web application requires multiple servers in order to handle efficiently a huge volume of incoming traffic. The same application should run in each one of these servers, and the job of the load balancer is to distribute the requests in such way,that will make the best of each server's available capacity. Another common word for "load balancer" is "scheduler", even though in reality these two modules work for different purposes, especially since in most implementations they are based on the same scheduling algorithm, that of round robin.

2.8.3 NGINX Reverse Proxy Server and Load Balancer

The open source NGINX software [10] can work both as reverse proxy as well as a load balancer offering a plethora of useful features, including:

- Many load-balancing algorithms to choose from.
- Communication protocols such as HTTP/2 and WebSocket.
- Security strategies such as whitelists and blacklists.
- Scripting for advanced use cases, using Lua, Perl, and JavaScript (with the nginScript dynamic module [Editor – Now called the NGINX JavaScript dynamic module]).

[10] `https://www.nginx.com/blog/docker-swarm-load-balancing-nginx-plus/`

An updated version of NGINX, titled NGINX Plus, features many more options, such as more advanced authentication protocols, as opposed to free-to-use NGINX, although, it comes at great cost for the user. Possibly, one of its most key advantages is the ability to run diagnostic checks on the nodes for which it acts as a workload loadbalancer.

2.9 Related Applications

2.9.1 Kubernetes Horizontal Pod Autoscaler

"Kubernetes container orchestration and management system automates deployment, scaling and management of containerized applications. Each application can run in many copies in clusters of nodes (each node is typically deployed as a VM). Each node can run more than one instance of the application. The maximum number of nodes is specified in advance. Inside a node, an application is deployed in clusters of containers referred to as "pods"; each pod is deployed as a separate Docker environment running one or more containers. Kubernetes resumes responsibility for scheduling and managing each application and its replicas inside each cluster. Kubernetes can also be configured to balance the traffic across clusters."[2]

"The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization (or, with custom metrics support, on some other application-provided metrics)."[11] The Horizontal Pod Autoscaler is implemented as a control loop (default value of period is 15 seconds), a layer above the previously mentioned controllers. During each period, the metrics from the resource metrics API for each pod targeted by the Horizontal Pod Autoscaler. The controller divides the receive metrics value by the desired metrics value, and then multiplies the outcome of this division with the current number of pod replicas. The outcome of this multiplication is the desired number of pod replicas.

2.9.2 Amazon Elastic Compute Cloud

"Amazon Elastic Container Service (ECS)5 by Amazon Web Services (AWS) allows management of Docker containers on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances (VMs). It can be used to launch and stop containerized applications by making API calls, allows monitoring the state of the cluster from a centralized service and integrates with many familiar AWS features like Elastic Load Balancers, CloudTrail, CloudWatch etc. The cluster setup is a single step process in which the number and flavor of EC2 instances needed for the cluster is specified. The rest of the setup process, as well as the management of those instances is handled by the ECS service. In terms of autoscaling, ECS provides a mechanism which lets a user configure policies on how scaling operations take place. A policy consists of a set of rules and a set of

[11] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

actions. Rules typically refer to thresholds defined upon utilization metrics and actions refer to scaling operations. An example policy would be the following: If CPU utilization in all cluster nodes is above x%, create a new node with y amount of computing resources and add it to the cluster. ECS, autoscaling operations are driven my measurements of the actual computing resources consumed. In addition to scaling, ECS can move applications (tasks) across the cluster of nodes in order to achieve better utilization, meaning that ECS's scheduler operates based on the actual run-time utilization of each application"[2]

2.9.3 Azure Kubernetes Service

"AKS (Azure Kubernetes Service) nodes run on Azure virtual machines. It allows connection of storage to nodes and pods, upgrade cluster components, and use GPUs. AKS supports Kubernetes clusters that run multiple node pools to support mixed operating systems and Windows Server containers. Linux nodes run a customized Ubuntu OS image, and Windows Server nodes run a customized Windows Server 2019 OS image. As demand for resources change, the number of cluster nodes or pods that run the clients services can automatically scale up or down. The client can use both the horizontal pod autoscaler or the cluster autoscaler. This approach to scaling lets the AKS cluster automatically adjust to demands and only run the resources needed."[12]

[12] `https://docs.microsoft.com/bs-latn-ba/azure/aks/intro-kubernetes`

3 Architecture

Elixir is a an autonomous agent written in Python 2.7. It supports monitoring, up-scaling, down-scaling and managing multiple swarms of nodes running different Web applications.

Fig.8, illustrates a typical Docker Swarm environment with multiple swarms, each one running a different Web application, and all controlled by one provider. Each node (manager or worker) can have from 1 to N containers inside, running the same Web application. In this exemplary system there is one manager for each swarm, there is no load balancer and no way of keeping track of the managers and the workers resource usage.

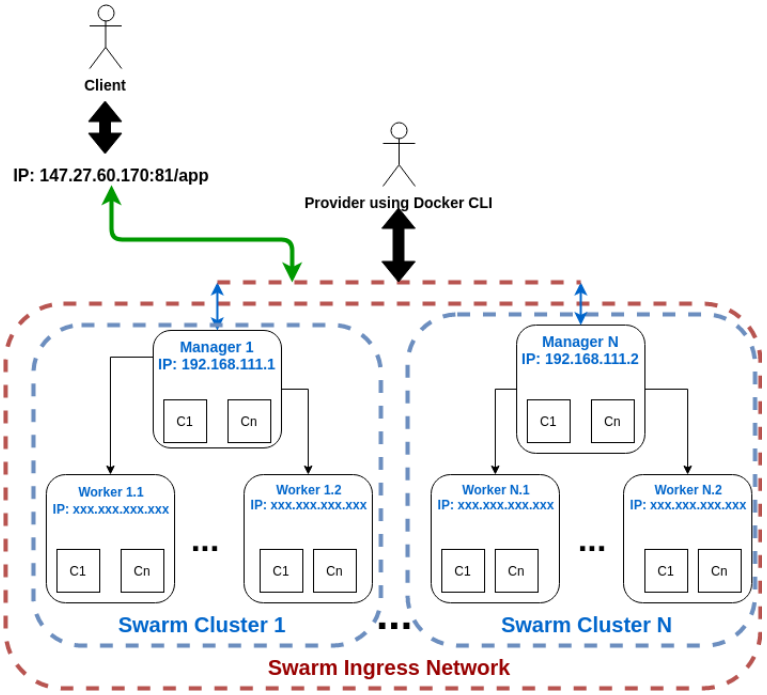


Figure 8: Before Elixir's implementation in the system

Docker Swarm does not provide real time monitoring of each swarm's nodes. So in cases of system overload, there is no way for the swarm to react and adapt to the situation, and will probably end up failing. In order to deduce whether the system is functioning properly or not, the infrastructure provider is expected to use an external tool for monitoring. These external tools, however, tend to be pay-to-use and difficult to operate alongside Docker Swarm's software. But even if the provider was able to add such a monitoring tool to his system's imple-

mentation, the only way for him to add extra worker nodes in case of resource demand, would still be to do it manually through Docker CLI. These problems can be resolved with the implementation of Elixir into the system.

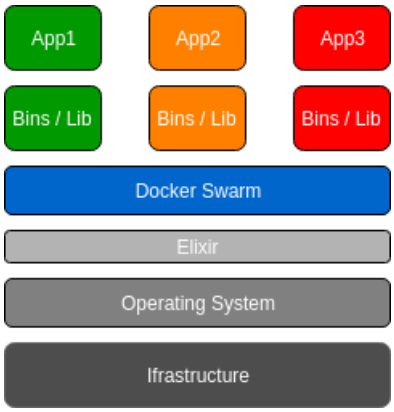


Figure 9: The Elixir agent deployed in the system.

In this section, we will explore a little more in depth, all the separate modules that together form the system architecture that Elixir's auto-scaling is emulating, more specifically the one illustrated in Fig.10. We can imagine Elixir as some sort of middle layer between the OS of a computer and the Docker Swarm environment, as illustrated in Fig.9. Elixir will share the system's infrastructure and resources in order to function properly and achieve its goals, but it will actually work as an orchestrator a level above the common Docker Swarm, complementing in that way all the Docker Swarm's shortcomings and extending its capabilities. Elixir will also have complete access over all of the resources and information managed by the Docker Swarm orchestrator (i.e. containers, services, worker node status), and it will act as an automated mechanism managing the entirety of the system, "a manager to rule over all the other managers".

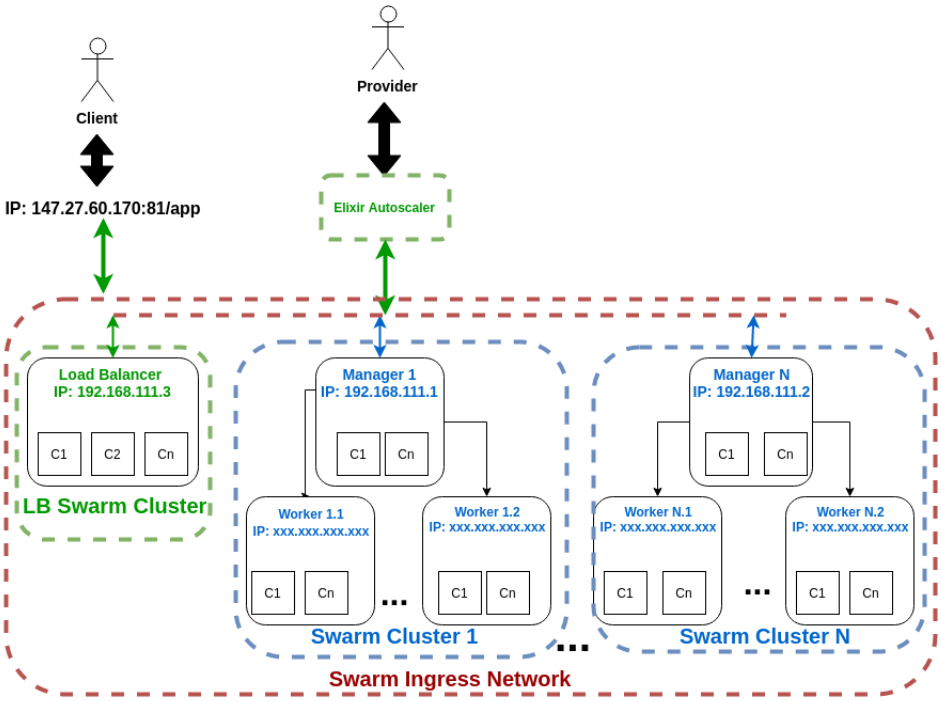


Figure 10: After Elixir's implementation in the system

3.1 Load Balancer

Perhaps the most significant action of Elixir, is the inclusion of a load balancer node on the system it manages. This node will form a swarm of its own, where it will deploy a containerized NGINX proxy server service by acting as its manager (and one and only worker), meanwhile gaining access on the swarm ingress network formed in the provider's computer. The swarm ingress network is an overlay network, which handles control and data traffic related to swarm services. It is the default network assigned to swarms on creation, when the user does not choose a custom user-defined network instead. The swarm ingress network also supports communication between nodes of different swarms, as long as these swarms are orchestrated on by the same Docker client. As we already mentioned in chapter 2.8, an NGINX proxy server is actually a reverse proxy server and is capable of hiding the IP addresses of the backend servers, guarding them in that way from possible malicious attacks. An NGINX proxy can also act as a load balancer for the system, distributing the incoming requests based on a selected load balancing algorithm. For our work, we choose to implement the round robin scheduling algorithm. Elixir processes the configuration file intended for the NGINX proxy server service, every time a node (manager or worker) is to be added to the system, and then restarts the service, while mounting the configuration file intended for that swarm. In listing 1 we present an example of that NGINX configuration file. Elixir handles different NGINX configuration files inside the load balancer node, one for each swarm it looks after.

```
http {
    server {
        listen 80;

        location / {
            proxy_pass http://backend;
        }
    }

    upstream backend {
        server 192.168.111.201:80;
    }
}
```

Listing 1: NGINX's configuration file for load balancing

Besides working as a reverse proxy server, the load balancer node will also act as the Zabbix server, keeping track of all the monitored hosts. This service is also implemented as a containerized application, however this time it do not require multiple services, one for each swarm. There will only be a single Zabbix server, gathering the information of all nodes (managers or workers) created by the provider, no matter which swarm they belong to or which application they serve. All the gathered data will be stored on a containerized MySQL database, to be processed later for future use.

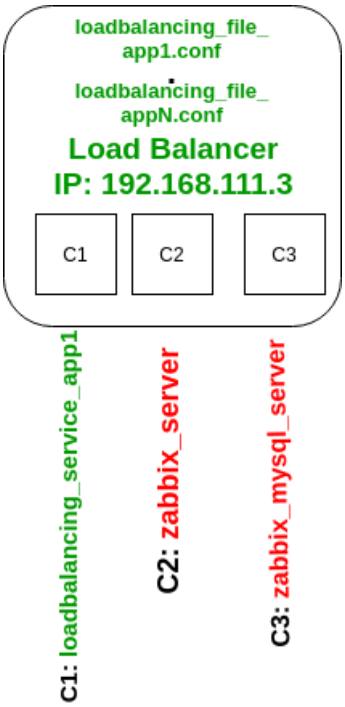


Figure 11: Load Balancer's structure

3.2 Manager

Manager nodes are responsible for handling swarm management tasks such as maintaining cluster state, scheduling services (assigning service containers to workers), scaling services and etc. The manager basically aims to preserve a consistent internal state of the swarm cluster and of all the services running on it. "To take advantage of swarm mode's fault-tolerance features, Docker recommends you implement an odd number of nodes according to your organization's high-availability requirements. When you have multiple managers you can recover from the failure of a manager node without downtime." [13] For testing purposes, however it is permitted to maintain a swarm with a single manager, and even if the swarm manager fails, the services will continue to run, but scaling of services and nodes will not be operational. Manager node are capable of load balancing the swarm their are managing themselves, however we override this feature with the addition of the load balancer node described on the previous section. The reason for this action is our aim to create a tool that would handle more than one swarms and would add security to the entire system.

In our implementation, Elixir is not allowed to create any extra manager nodes, but only workers. Therefore, it is up to the provider of the Web application to choose how many manager nodes he is willing to have to online for managing a single swarm cluster with as much fault-tolerance on the system as possible. Elixir's task is to recognise how many manager nodes are currently online, which of them are currently managing the same swarm, and make turn these manager nodes into monitored hosts, by installing them the Zabbix agent in the form of a containerized application. Elixir does not create a swarm cluster by itself, it only extends the managing limitations on existing swarms, so the manager nodes will already have the Docker Engine installed if they are to function as managers properly. Later, Elixir will create separate NGINX configuration files, one for each swarm, based on the number of manager nodes, and will pass the manager node's IP addresses and ports of application on these files, before restarting the load balancing service.

3.3 Worker

A worker node will act, every time, as the Elixir commands through its swarm's manager node. A worker does not have any authority or special commands in the swarm that it belongs to, it cannot increase the number of its containers running for swarm services on its own, and it cannot start a service on the swarm on its own. Every single action that the worker must take for the good of the swarm, must be ordered through its swarm's manager. "You can create a swarm of one manager node, but you cannot have a worker node without at

[13] `https://docs.docker.com/v17.09/engine/swarm/how-swarm-mode-works/nodes/#manager-nodes`

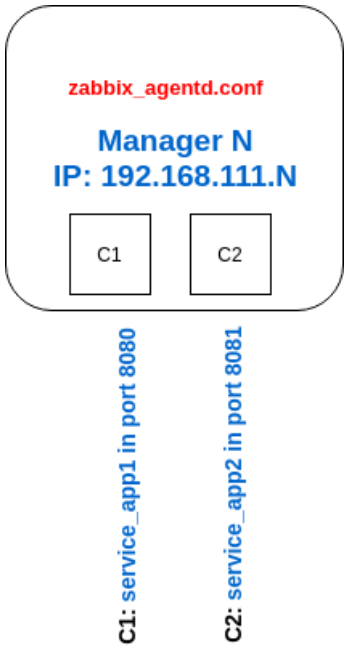


Figure 12: Manager's structure

least one manager node."[14] After a worker node is created, Elixir installs to it the Docker Engine and then the Zabbix agent, in the form of a containerized application. Then, Elixir overrides the Zabbix agent's configuration file in order to assign a monitoring style (Active or Passive) and write the IP address of the Zabbix server. After all these steps are completed, Elixir will write this worker node's IP address and port of application on the NGINX configuration file and restart the load balancing service, so that the load balancing can proceed and the worker node can commence work for its assigned service.

[14] https://docs.docker.com/v17.09/engine/swarm/how-swarm-mode-works/nodes/#manager-nodes

4 Elixir's Implementation

In this section we will describe the necessary operations Elixir is offering, but this we will be looking at the code. Fig.13 is a flowchart illustrating all the steps Elixir has to take, while in operation, in order to manage the swarms of the provider's system and maintain a well balanced environment. Elixir begins with creating the necessary triggers based on the threshold provided by the provider. Elixir supports both Disk usage based triggers, as well as CPU usage based trigger. The load balancer node is already online and its features are ready (Docker Engine, Zabbix server), so there is no need to create it and install them from scratch. After these initial steps, Elixir search for all the manager nodes currntly online, divides them into groups accodring to which swarm they manage, and stores the information of only 1 manager for each swarm, as the main swarm's main manager. Based on the number of main managers, it will also create an equal number of NGIXN configuration files, in order to start the load balancing services. Later, Elixir proceeds to perform periodical health checks on the main manager of each swarm, keeping track of the its trigger values. When the limits of these triggers are surpassed the appropriate reaction by Elixir will follow (auto-scaling). Last but not least, Elixir will wait an entire minute before revisiting the same swarm for health checking again. The triggers are checking for average usage values in a time frame of 5 minutes, hence if Elixir does not wait that full minute, its decision might be affected by the manager's average usage value, that will not have been able to adjust to the after scaling system yet.

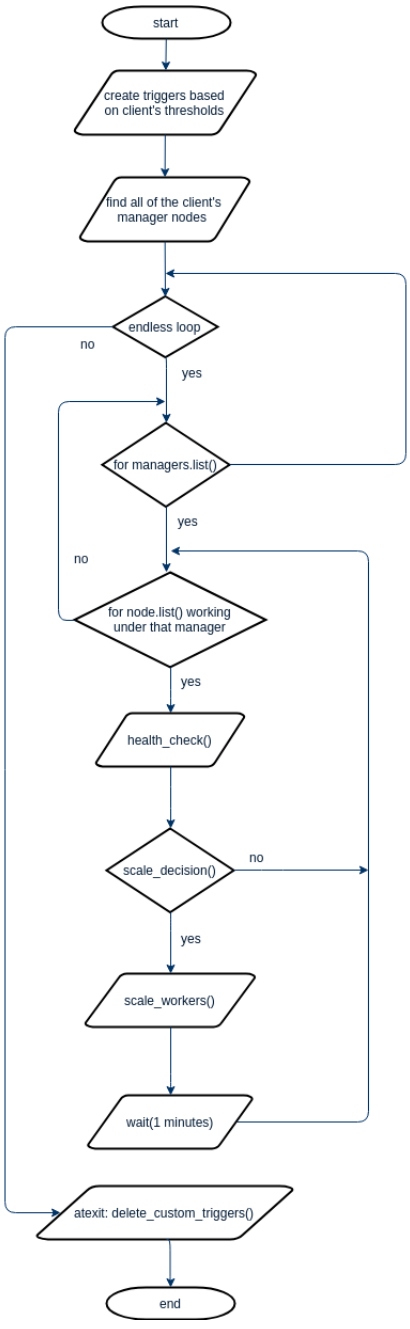


Figure 13: Elixir's logic flowchart

In listing 2 we present an example of how the request for creating a trigger should look like. Elixir uses these requests only once every time it is brought online, in order to set up the proper triggers for CPU monitoring and/or Disk monitoring according to the users thresholds of choice. Elixir will also delete these triggers, if it is shut down by the user. The "description" is the alert message that the infrastructure provider will receive, in order to understand what is the worker nodes problem, "expression" is the macro expression which checks whether the trigger is satisfied or not, and "dependencies" is a parameter added only if our custom trigger requires the values of an other trigger to function properly. We are allowed to create more than one triggers every time with a single trigger create request. All the requests are stored into json-rpc files.

```
{
    "jsonrpc": "2.0",
    "method": "trigger.create",
    "params": [
        {
            "description": "Processor load is too low on {HOST.NAME}",
            "expression": "{Template OS Linux:system.cpu.util[,idle].avg(10m)}>80",
            "dependencies": [
                {
                    "triggerid": "17367"
                }
            ]
        }
    ]
}
```

Listing 2: Trigger create example

4.1 Find All Manager Nodes

This is an important part of Elixir, since it works as an agent a layer above Docker Swarm, which means Elixir is responsible for managing containerized applications already deployed on Docker Swarm by a provider, not deploying them by itself. In order to achieve that level of control, Elixir must first find all the managers enlisted on the provider's computer. For that, Elixir is using the equivalent of the Docker CLI command "docker node ls", from the available commands of the Docker SDK, as presented in algorithm 1. The response of this command for each node determines, whether this node is a manager or not and to which swarm it belongs to.

In listing 3 we present an example of how the request to Zabbix should look like, in order to create a host. First we define the method that we need to use from the API and then we define the parameters for this method. Parameter "host" is the name we want to assign to the monitored server, "ip" is the server's IP, "groupid" will add the created host on a specific group of servers, that we define, where there will be specific triggers and monitoring rules. For our work,

Algorithm 1 Finding All Manager Nodes Algorithm

```
1: procedure FindAllManagers
2:     nodes ← AllSystemNodes
3:     for node in nodes do
4:         response ← node.cmd(NodeList)
5:         if response contains 'Manager' then return managerlist.add(node)
```

every single new host created is added to the 'Linux Servers' group, since all of our worker nodes will be Linux servers. The "port" value must always be 10050, since this is the port that Zabbix uses.

```
{
    "jsonrpc": "2.0",
    "method": "host.create",
    "params": {
        "host": "Linux server",
        "interfaces": [
            {
                "type": 1,
                "main": 1,
                "useip": 1,
                "ip": "192.168.3.1",
                "dns": "",
                "port": "10050"
            }
        ],
        "groups": [
            {
                "groupid": "50"
            }
        ],
        "templates": [
            {
                "templateid": "20045"
            }
        ]
    }
}
```

Listing 3: Host create example

4.2 Add Node Logic

After the initial and mandatory steps of Elixir, it is time for the periodical health checks performed on the swarm's manager. The swarms must have already been initialised and their manager nodes assigned, all by the provider. All the data from the health checks are stored into the MySQL database of the Zabbix server. Elixir gets notified that there is something wrong, only if any of the declared triggers gets violated. When this moment comes, Zabbix server sends the trigger descriptions of the violated triggers, to Elixir so it will know which action to

perform (up-scale, down-scale or nothing). The actions performed, when up-scaling has been triggered, are illustrated on algorithm 2. At first Elixir must come up with a name for the new worker, which it does based on the number of already existing worker nodes in the swarm and then orders the creation the new worker through a mixture Docker-Machine and Docker SDK implementation. Then, Elixir gets the join-token from the manager node, and with the help of the Docker SDK gives this token to the worker node so it can join the swarm. Lastly, Elixir turns the newly created worker node on a monitored host, with the use of the Zabbix API host-create method,restarts the NGINX load balancer service, after it writes the new IP on the NGINX configuration file, and then scales the service up. Below, we present a flowchart diagram 14 with all the basic information about Elixir's add-node workflow.

Algorithm 2 Add Worker Node Algorithm

```
1: procedure AddWorker
2:     worker.WorkerName ← create.WorkerName()
3:     docker.createworker(worker)
4:     manager.swarm(GetJoinToken)
5:     worker.join(GetJoinToken)
6:     host.create(worker)
7:     loadbalancer.add(worker.getIP())
8:     loadbalancer.update()
9:     manager.scaleUp(service)
```

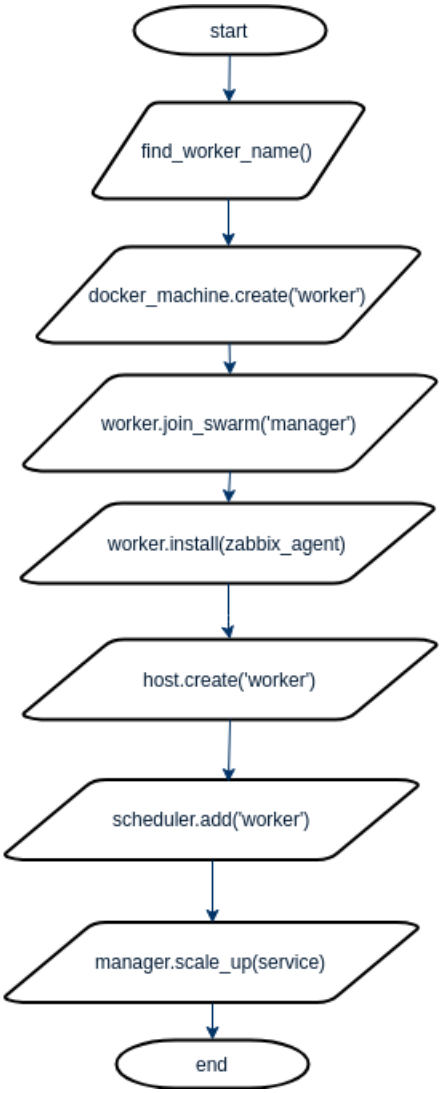


Figure 14: Add node logic flowchart

4.3 Delete Node Logic

In case the trigger description indicated a problem for down-scaling, Elixir will follow the necessary actions as they are illustrated on algorithm 3. At first Elixir creates a list with all the worker nodes of the swarm, and the proceeds to pop the last one created off the list. Elixir, then, removes this worker from the swarm, after it puts it on Drain mode, and deletes the host from the Zabbix server, after getting the workers IP address, once again with the use of Docker SDK commands. Lastly, the IP of the worker node gets deleted from the NGINX configuration file and the load balancing service gets rebooted. Only then, the service of gets scaled down. Below, we present a flowchart diagram 15 with all the information concerning Elixir's basic delete-node workflow.

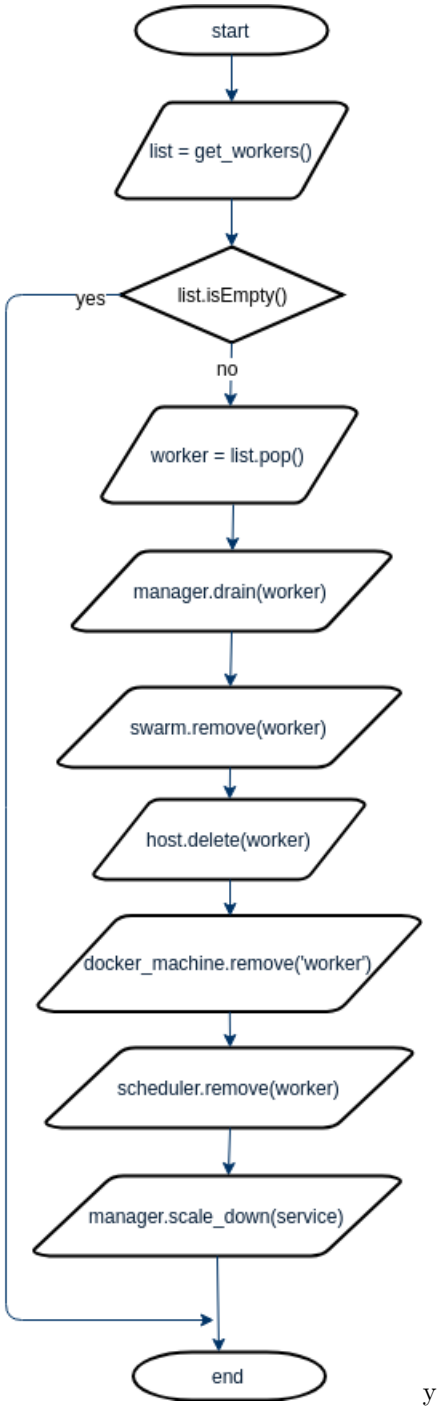
Algorithm 3 Delete Worker Node Algorithm

```
1:  procedure DeleteWorker
2:      list[] ← manager.getWorkers()
3:      if list is NotEmpty then return
4:          worker ← list.pop()
5:          manager.swarm.remove(worker)
6:          manager.service(ScaleDown)
7:          host.delete(worker)
8:          loadbalancer.remove(worker.getIP())
9:          loadbalancer.update()
10:         docker.delete(worker)
11:         manager.scaleDown(service)
```

In listing 4 we present an example of how the request to Zabbix should look like, in order to delete a host. After defining the method, the only other parameter Zabbix requires is the monitored server's host-id.

```
{
    "jsonrpc": "2.0",
    "method": "host.delete",
    "params": [
        "13"
    ],
    "auth": "038e1d7b1735c6a5436ee9eae095879e",
    "id": 1
}
```

Listing 4: Host delete example



y

Figure 15: Delete node logic flowchart

5 Performance Evaluation

The basic goal of the experimental procedure that we followed, is to prove that our agent works correctly and that it has the desirable effect on the system, which means achieving lower response time for a great concurrency number through automatically scaling up the worker nodes in the system, and vice versa for the respective case. In order to acquire the necessary results, we selected to use a CPU intensive testing application. We expand on the subject of this testing application and we analyse the results of our evaluation in the following subsections.

5.0.1 Testing Application Deployment

As we already mentioned on the above paragraph, we choose to use a CPU intensive application in order to carry out these tests with meaningful results. That is why our application is as an input, an edge from a given graph, . The graph is already created inside the Web applications code, since our goal is to create a realistic scenario for stressing the system's CPU usage, not create an elaborate Web application. We developed this Web app on Flask, a microframework for Python and later containerised it, created an image out of the running container, uploaded the Docker image on DockerHub, and from there distribute the Web app service among all swarm nodes with Elixir. The containerization and distribution of this Web app would not be possible without creating the proper docker-compose.yml file. Information about the correct structure of this file, can be found on the internet, and will not be included in our thesis, since they are not part of our research.

Every single added node will have the same characteristics as the manager in line with the horizontal scaling policy. That means that every single node (manager or worker) owns 1 CPU, 20GB Disk and 2GB RAM as available resources. The nodes have small sized flavor, and use 'base_ubuntu_18.04' as their base image. There reason, they use this image instead of earlier versions of ubuntu, is for , since the latest version Docker Engine might appear to be malfunctioning when installed on earlier ubuntu releases.

Finally, we use Apache bench, a tool for benchmarking a Hypertext Transfer Protocol (HTTP) Web server. This software tool will be used to stress our application with multiple concurrent users, in order to creating a realistic stressing (CPU intensive) scenario for the system and for Elixir. Then we will measure the performance of system, based on the response time per a number of requests. An argue can be made for not choosing Disk usage instead of the CPU usage (that we chose) for the testing of our application, since 100% Disk usage means that worker node can no longer perform any more additional tasks. We actually consider both of these parameters to be of great importance for the well being of every worker node and we strongly believe that in future works,

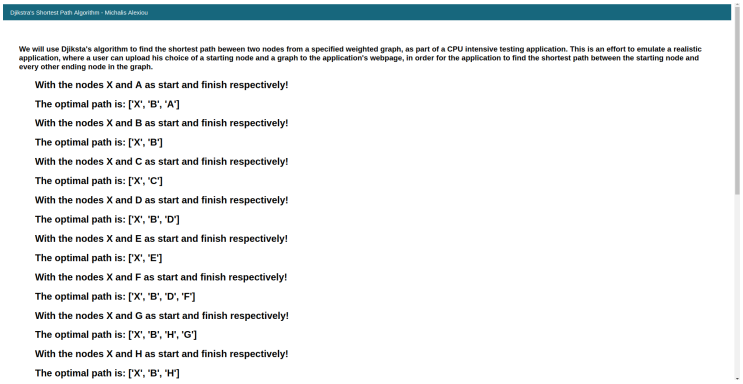


Figure 16: The HTML layout of the testing Web application we created.

the decision to scale up or down should be made based on the metrics gathered by both. However, the purpose of our evaluation process is to prove that Elixir is capable of functioning properly against heavy workload and perform the necessary auto-scaling on time, rather than studying which parameter would prove to be better for auto-scaling or which would be the better threshold for making the decision to scale up or down.

5.0.2 Experimental Procedure

Our first test is showcased in Fig.17 and in Fig.18, where we illustrate the response time of the system and the CPU percentage of the manager node respectively during the testing. As we have already mentioned, the chosen CPU percentage threshold is an input for the agent and can be changed according to the user's desires. Having said that, for this particular test we choose to use the 20% CPU usage mark as the threshold to trigger the up-scaling in our system, in order to speed up the already time consuming process of automatic up-scaling the system. With the Elixir agent online and managing our system, we begin with a single node and workload of 400 concurrent users. This will prove enough to push the CPU percentage levels of the manager node beyond the 20% threshold and in that way will trigger Elixir's up-scaling procedure. It is important to note that Elixir's monitoring protocols are checking for average CPU levels in a time frame of 5 minutes, so that it can avoid any possible spikes that can occur by mistake during Zabbix's monitoring of the manager node or during the manager node's operation in general. The dashed lines indicate when the workload of the system is increased to more concurrent users, each color stands for a different concurrency. The red dashed lines indicate the exact moment on the system, when the up-scaling (add node logic) was triggered. The red dots in Fig.17 indicate the moment when the new nodes were created. It is important to notice the drastic drop in response time right after the creation of

the new node. This is the moment where the NGINX service of the loadbalancer reboots, in order to include the IP of the newly created worker node, for the rest of the testing process, with the result of causing a momentary downtime. It is also of equal importance to remember, that monitoring the CPU usage of the manager node is enough, since it will give the same CPU usage as of any other worker node, because of the load balancing algorithm we implemented (round robin)

In Fig.17, we observe that the response time of the system keeps on rising, because the continuous workload of 400 concurrent users leaves little time for the system to balance itself. However, after the addition of next worker node in the system, we can see that levels of CPU usage as well as the total response time of system experience a very small drop, before stabilizing at about 55% and 4000 miliseconds respectively. This is a logical outcome, since we will now have 2 nodes working on for the system (1 manager, 1 worker), but we have also increased significantly the number of concurrent users in order to push the system for another up-scale. After the creation of the second worker and the reboot of the NGINX load balancing service, we lower the workload to 600 concurrent users, a small enough number to not cause any further up-scaling, but not small enough to trigger an accident down-scale. During this period, we find the system stabiling its response time at about 2000 miliseconds and the manager's CPU usage at about 25%.

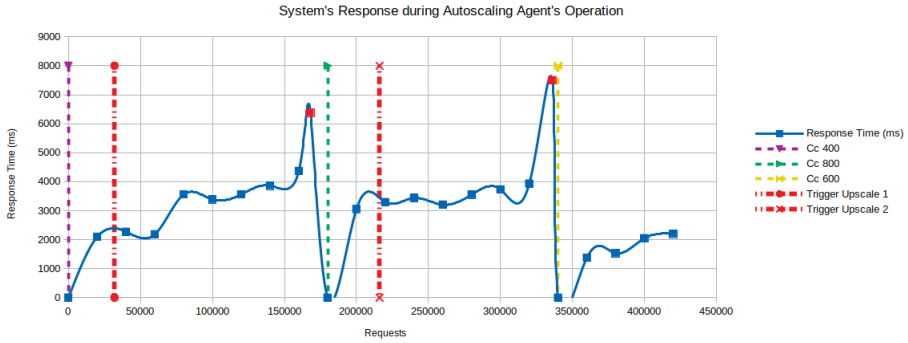


Figure 17: System's response during the auto-scaling test.

In Fig.19, we explore another realistic scenario where the level of concurrent users decreases gradually, every time after a new node has been deleted from the system, setting up in that way a chance to test our work in an environment once again with a more dynamic workflow. For this test, we choose to use a threshold of 15% CPU usage as the down-scale trigger limit. In Fig.20 we present the managers CPU usage during the same test, and how it fluctuates

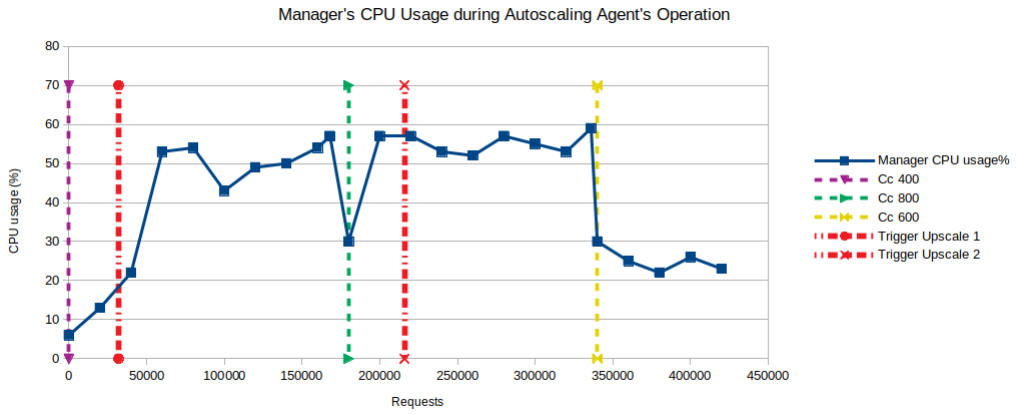


Figure 18: Manager node CPU usage percentage during the auto-scaling test.

up and down again during the auto-scaling process.

From these experiments, we observe that the both response time of the system and the CPU usage of the manager keeps on dropping, because of the enforcement of a workload of 400 concurrent users, after a much higher initial workload of 800 concurrent users that scaled the system into 3 nodes (1 manager, 2 workers) in the first place. We notice the levels of CPU usage and total response time of system stabilizing at about 13% and 2500 miliseconds respectively, right before the down-scaling trigger is enabled. This time the red dots on the response time Fig.19 indicate the moment when the extra worker is deleted. After the deletion of that worker, the reboot of the NGINX load balancing service causes, once a again a drop at the response time since the system will be temporarily down. We proceed with deceasing the number of concurrent users to 200, causing yet another down-scaling. After of the last worker node in the system is deleted, and the only working node is a manager, we reboot the load balancer and this time, enforce the small workload of 50 concurrent users, just to keep the system busy and the Elixir functioning. Even though the manager's CPU usage in 20 is stabilizing below the appropriate limit (about 11%) and there should be a call for down-scaling, Elixir will never delete a manager node, because that would destroy the swarm and bring the Web application offline. During this period, we find the system stabiling its response time at about 1400 miliseconds.

Finally, we have included 2 diagrams 21 and 22 with the response time results, while testing the system with 3 nodes (1 manager, 2 worker nodes) and 1 node (1 manager), but this time we turn off the Elixir agent. The purpose of these tests is to showcase that the produced results during the previous tests were

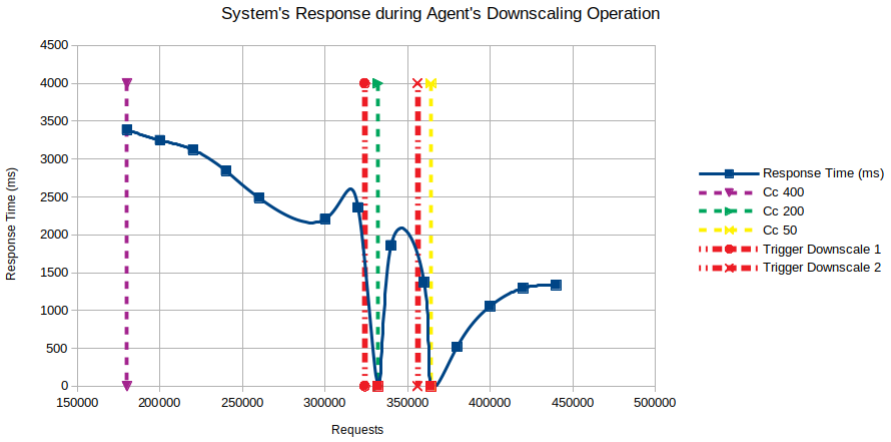


Figure 19: System's response during the down-scaling test.

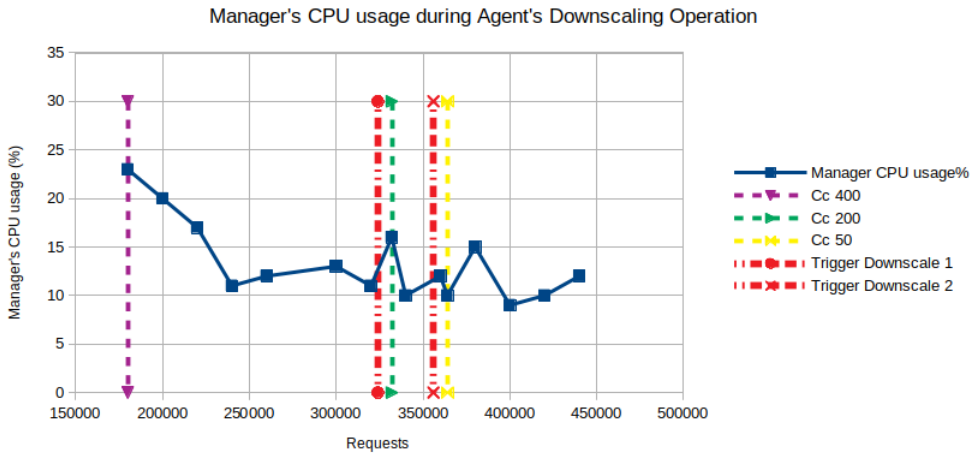


Figure 20: Manager node CPU usage percentage during the down-scaling test.

correct, in the aspect of response time with respect number of worker nodes and concurrent users number. In Fig.21, we can easily see the response time of the system with 3 nodes, stabilizing at about 2000 miliseconds, that is the same value around which the response time of the system with 3 nodes under Elixir's operation stabilizes as well, excactly after the last auto-scaling process is completed. This is a very reassuring outcome for the performance of Elixir, especially since in both tests at this point the number of concurrent users was

600. In Fig.22, we notice that the response time of the system with 1 node, is stabilizing at about 1500 miliseconds, which is a bit higher from the average response time value of the system with 1 node, while Elixir is online, and with 50 concurrent users, but still similar enough to reassure us about the correct function of Elixir.

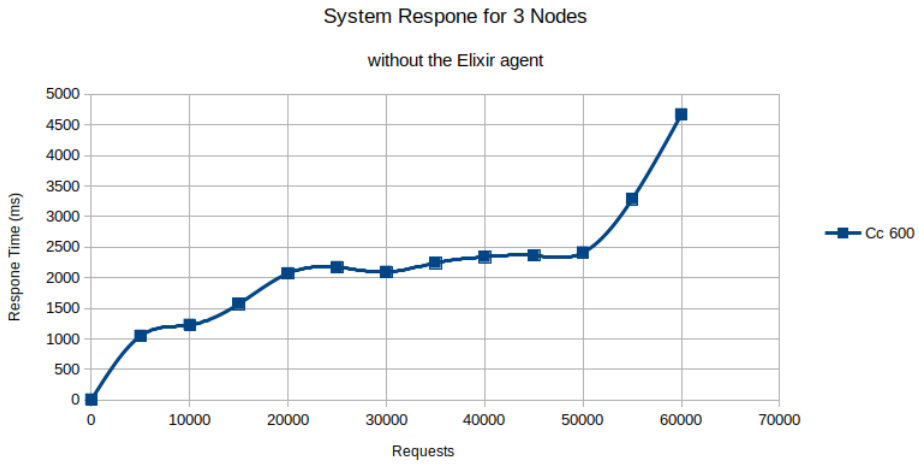


Figure 21: System's respone with 3 nodes and without the Elixir agent.

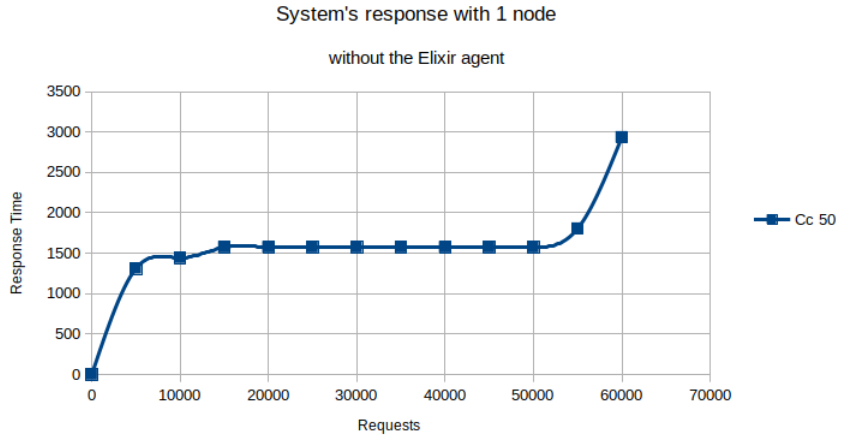


Figure 22: System's response with 1 node and without the Elixir agent.

6 Conclusions

For this thesis, we introduced and implemented a concept autonomous agent, named Elixir, capable of scaling automatically nodes and services orchestrated by Docker Swarm. Elixir's functionality emulates a realistic system architecture, which supports horizontal auto-scaling. Elixir's implementation is based on many different software tools such as Zabbix for distributed monitoring, or NGINX for load balancing. We then tested Elixir through a series of CPU stressing scenarios, where Elixir proved to be capable enough to adapt on a dynamic workload and adjust the system's allocated resources beneficially. So the conclusions about Elixir's contributions can be summarised as follows:

- Elixir provides an official mechanism that can scale up and down services running on Linux servers (worker nodes) and worker nodes automatically, without any need for human supervision.
- Elixir provides a monitoring support for any worker or manager node in the swarm and act accordingly, when there is an internal error in the same.
- Elixir is an agent that can manage multiple applications at once for each provider, while increasing the system's safety at the same time, with the inclusion of a reverse proxy server.

7 Future Work

The following are important issues for future work:

- Find a way to reduce (or even better eliminate) the downtime of the system, caused by NGINX's load balancing service reboot.
- Improve Elixir's speed in order to make it a competitive option against Kubernetes and Amazon EC2. Elixir, for example, gathers data only about managers and stores them in a list, while it does not keep track of any workers. Any time Elixir needs information about worker nodes, it has to obtain it through a command in the manager of the swarm, which costs time. Perhaps, it would prove more beneficial for Elixir, not store this information on a list of workers, when creating the worker. Another upgrade that could help with Elixir's speed, could be to use parallelization, in order to speed up the up-scaling and down-scaling procedures.
- Extend Elixir's capabilities by adding machine learning in order to gain insight as to what would be the best threshold value for triggering the auto-scaling, just like Dr. Sotiriadis suggests in his work.[5]
- Last but not least, define through testing and research, which is the best possible number of manager nodes, managing 1 swarm cluster. In the Docker Documentation there is still no certain answer, only that in reality the number of swarm managers must not be below 3.

References

[1] Bondi André B. *Characteristics of Scalability and Their Impact on Performance*. Proceedings of the 2nd international workshop on Software and performance, 2000.

[2] Christodoulopoulos Christos, Petrakis Euripides G.M., and Sotiriadis Stelios. *Commodore: Fail Safe Container Scheduling in Kubernetes*. Advanced Information Networking and Applications, 2019.

[3] Chaufournier Lucas, Sharma Prateek, and Shenoy Prashant. *Containers and Virtual Machines at Scale: A Comparative Study*. Proceedings of the 17th International Middleware Conference, 2016.

[4] Davidsson Paul. *Concepts and Autonomous Agents*. Department of Computer Science, Lund University, 2004.

[5] Sotiriadis Stelios, Bessis Nik, Amza Cristiana, and Buyya Rajkumar. *Vertical and horizontal elasticity for dynamic virtual machine reconfiguration*, volume PP. IEEE Transactions on Services Computing, 2016.