POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

WIP Title: Dynamic resource allocation in the cloud for compute heavy tasks in a containerized environment

Tesi di Laurea Magistrale in
Computer Science and Engineering

Author: **Elia Ravella**

Student ID: 967243
Advisor: Prof. Raffaela Mirandola
Co-advisors: Name Surname, Name Surname
Academic Year: 2021-22

Abstract

Keywords: Cloud, Containers, Dynamic infrastructure

Contents

1 | The Problem, the State of the Art and Current Available Solutions

1.1. Introduction

1.2. The Problem

Scalability is one of the key design point that must be taken into consideration when developing a software. If a system cannot scale in power when the userbase or the load requested changes it slows down, making the response times growing for each request and compromising the overall performance of the application. The most intuitive approach to scalability, which is also the most common in cloud environments, is horizontal replication. With horizontal replication I mean (throughout all this thesis) the addition of identical software modules alongside the already existing ones to share the load; to do so, different incoming requests are routed to different modules when they arrive. The replicas being identical (and usually stateless) ensure that each request is carried out in the same way. The policies for deploying replicas can be either static or dynamic (based on the predicted load during the day or measuring the real time traffic incoming, for example).

This approach has seen a wide adoption in the industry and is the *de facto* standard to tackle scalability problems, especially in web environments. The horizontal replication approach gives applications the flexibility they need in reacting to the load that is applied, and is especially effective when the application is divided in submodules that can be individually scaled.

What happens when the requests that an application must serve change also in nature, and not only in volume? Horizontal replication works well when the load is mostly uniform

(and for the majority of web applications, it is) and can be analyzed in a one dimensional fashion as "the number of requests". When requests set in motion heavy computational pipelines, as image processing or complex mathematical problems, but the interface they are served is shared with all the other *light* requests[1] then a single request can weight, in terms of resources it needs to be carried out, very differently from the others. In this scenario, horizontal replication is harder to put in place effectively: if requests weight differently it is not possible to just share them equally among replicated servers[2] because in some cases a server will receive a much more higher share of heavy requests and be stuck executing them while other servers will be idling because they received only light requests. Horizontal replication, as it is implemented now, cannot face efficently this scenario. The proposed solution uses dedicated ephemeral workers to execute *heavy* requests, and organises and schedules them with a ticketing middleware.

1.3. Containerized Environment and High Performance Computing

Containerization Virtualization is "the act of creating a virtual (rather than actual) version of something at the same abstraction level, including virtual computer hardware platforms, storage devices, and computer network resources"[1]. In cloud environments virtualization is the most used tool to provide isolated services, as compute capabilities, storage capabilities or networking. There are different kinds of virtualization, depending on the layer virtualized: some solutions just virtualize the hardware and let the user install a full fledged OS over it, other virtualizes all the technological stack from the hardware to the OS level, leaving to the user (in most of the cases, a developer) only the problem of developing the application s/he wants to ship, without the need to care about hardware limitations or operative systems settings.

One of the most used virtualization technique in the industry is containerization: when a software is said to be *containerized* it is packaged in a format that encapsulates also all its dependencies, configuration files and variables and OS settings. This package, the container, must be run through a container engine which provide the communication with the underlying software and OS to execute the software. Famous container engines are Docker, Podman, chroot or rkt. Containerized applications have several benefits over

[1] this is not an impossible scenario: what a REST API exposes are a list of "light" HTTP requests that can trigger all kind of operations on the server they are executed on

[2] here *servers* is used to describe logical backend modules, not physical machines

classic deployed ones:

- They do not rely on the machine or OS they are running, just the engine
- If the container has been built correctly, the configuration of the application is already done
- Packaging all their dependencies makes them independent from the other application installed on the system: two containerized applications running on the same engine could be using the same library but at different (and even incompatible) versions
- The container acts also as an isolation mechanism, that keeps the application from interacting with other systems: this improves the security of the container applications

If an application is containerized, moreover, it is very easy to deploy several instances of the same application working at the same time: this kind of deployments improve scalability and reliability of the application.

High Performance Computing With "high performance computing" is usually addressed the field in computer science that studies computational heavy problems and develops solutions to solve them via techniques as extreme parallelism, clusterization and high performance networking. In the early stages of this project I focused also on HPC in order to understand if a virtualization approach as the containers one is suitable for such kind of tasks. The problem was the additional layer of virtualization added by the container engine: since most of the times HPC software relies heavily on low-level procedure calls, the additional virtualized layer could degrade too much the performance. However, as stated in the paper "Exploring the support for high performance applications in the container runtime environment"[2] (which original aim was to compare native performance with containerized ones) the optimization of engines is reaching a level that can offer near-native performance for HPC-like loads, at least in not-extreme scenarios.

1.4. State of the Art

There are several solutions available on the market that provides flexible infrastructure management and are built to automate the management of the infrastructure of an application. These systems can be seen as schedulers (so software components that organize when a task is executed and on which resources) with some additional features as the capability of actually *allocate* the resources needed or the automated management of the

interfaces between resources and components.

1.4.1. Shifter

Shifter is a simple scheduler which aims to utilize the container format in an HPC environment. It allows the user to specify the load in a docker image, then Shifter automates the conversion of that image to an HPC format and the scheduling of such task. Shifter is *not* an extension of Docker or the Docker engine, nor aims to automate the infrastructure, instead it just provides an additional interface (which is container compatible) to an already existing HPC platform.

1.4.2. SLURM

1.4.3. Kubernetes

1.4.4. Serverless Approach

2 | Design and Testing Phase

2.1. MapNCloud Original Architecture

Here I talk about the original deployment of the MapNCloud service. I plan to add a subsection explaining in detail the tech stack.

2.2. Problems Addressed

1. database choice and API modification
2. queue monitoring
3. resizable backend containers
4. cloud provider integration

At the end of this section I will present the "final" design draft

2.3. Testing and Validation

HERE I will introduce the "diffusion analysis" to justify the test parameters

1. CouchDB testing
2. RabbitMQ testing
3. Cloud providers options, pros and cons
4. technological limitations (docker-compose, load balancers)

I will also present the real "final" Architecture that will be deployed here, with cloup provider's technological names and services

3 Implementation

3.1. Frontend

3.2. Backend

3.3. Database

3.4. Messaging Middleware

3.5. Computational Layer

3.5.1. Renderino

Bibliography

[1] Virtualization — Wikipedia, the free encyclopedia. URL `https://en.wikipedia.org/w/index.php?title=Plagiarism&oldid=5139350`. [Online; accessed 28-July-2022].

[2] J. P. Martin, A. Kandasamy, and K. Chandasekaran. Exploring the support for high performance applications in the container runtime environment. 2017.