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 and the 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"[3]. 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 classic deployed ones:

- They do not rely on the machine or OS they are running, just the engine

[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

- 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"[4] (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 Industry

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. SLURM

SLURM is "an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters"[2]. It aims to organize and schedule tasks on multiple nodes; these tasks can also be defined as OCI-compliant containers. SLURM was crated to be executed on supercomputers or clusters of computers; in fact, SLURM focuses on communication between daemons and tasks through message passing framework as MPI and queue-managed resource access. Even if it can integrate containerized workloads, it is not suited to be deployed in a cloud environment rather than a computation center.

1.4.2. 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.

1.4.3. Kubernetes

Kubernetes (often called "k8s") is a container orchestration system, and is the *de facto* standard for container orchestration. It provides an all-in-one system to manage containerized applications:

- It provides an abstraction over the container level (the Pod) that is used to define the service provided rather than the container itself
- It includes different ways to persist data and state across containers; this gives a kubernetes cluster the capability to hold an entire application, from data layer to presentation layer
- Kubernetes clusters embeds security and access control by enforcing the already existing isolation features of a containerized environment and by providing a set of tools that easily control the access to the cluster itself (the Ingress controllers)
- It allows developers to define the redundancy for every single service defined, so to set an "horizontal scaling width" beforehand to handle faults and heavy loads

Kubernetes has been designed for applications that are designed as microservices, and it

provides the tooling for administrate single services, replication, and scaling in such an environment. As already stated in (1.2) the level of abstraction provided by kubernetes is efficent when horizontal scaling must be automated, but also removes some of the controllability of a containerized system. Moreover, resource management within kubernetes deployment in the cloud is very difficult: all the needed resources (as computational resources, memory resources, storage and networking ones) must be available at any time so that kubernetes can manage them, since the available k8s deployments at the moment are coupled to the underlying virtual machine. As an example, on the major cloud service providers, during the selection of the resources to allocate to a k8s cluster it is explicitly required to select a virtual machine size which will be the host of the deployment; this allows cloud providers to fix the resources available to each deployment. This also forces said resources pool to be always active, and in the case of a heavy deployment this drives up the costs.

1.4.4. Serverless Approach

Functions-as-a-Service is a cloud computing execution model that allocates computing resources on demand. Popular FaaS services include AWS Lambda, Azure Functions and GCP Cloud Functions. This execution model can be seen as a "nanoservice" approach, where an application is further divided into single funtion calls rather than services, but it has a very operational connotation: the term "serverless" is usually used to describe a type of *application deployment* that minimizes code deployment time, since the whole infrastructure is completely abstracted and managed by the cloud provider. The problem with a serverless approach is the actual computing power available: let alone the high level of abstraction (which again removes some control over the infrastructure and the actual resources allocated) the currently available solutions all impose limits both in terms of power usage, measured in CPU allocated, and in runtime: this renders impossible to use such an approach in the scenario described.

2 | Design and Testing Phase

2.1. MapNCloud Original Architecture

This thesis project extends and develops the original Distributed Software Development project, which have been created during the first semester of the 2021 / 2022 academic year. The original project aim was to create a system which could offer photogrammetry services, so the rendering of a tridimensional model from a set of images of the subject, leveraging the container technology to encapsulate the rendering engine. In this section I describe the original architecture of that system and in the next ones I will explain how that architecture has been redesigned, adapted and extended in this thesis project.

2.1.1. System Design

The MapNCloud project was designed as a "three plus one" tiers application:

- a **presentation layer** or front end, which has been developed as a web site and offers a graphical interface to the system
- a **business layer** or back end, which embodies all the business logic of the application
- a **data layer**, the database which persists the data needed by the application
- a **computation layer**, which is a separated back end module that is tasked with the most computational heavy routines (as the photogrammetry pipelines)

Another module is present in the application, and it is the ticketing service. It is not considered a whole tier, in fact its only job is to manage the queues of tasks that the backend generates, and let the computational layer components access them in the right order with the right schedule.

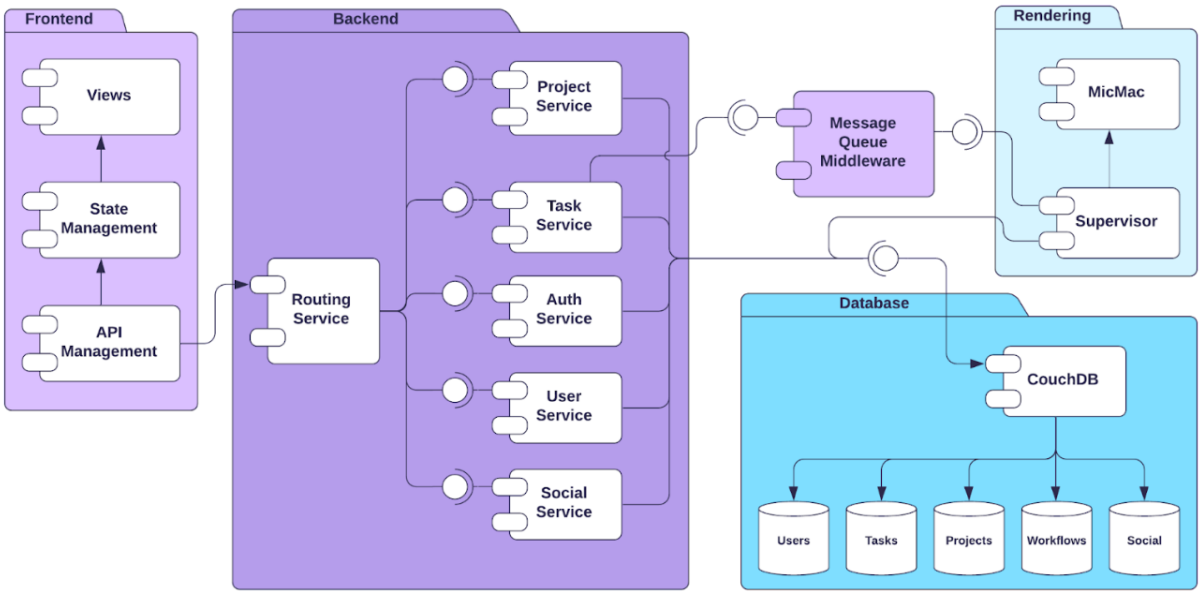

Figure 2.1: Original MapNCloud component diagram, with internal modules detailed.

Front End

The presentation layer of the application was developed as a web application; this exposes a graphical user interface for the services offered by the system. It is a JavaScript application (developed with the VueJS framework) which interacts with the backend via HTTP calls; this allows it to be considered, by the cloud provider, a static web application, which is easy to deploy and has a low resources demand.

The front end architecture follows the Model View ViewModel pattern, which separates the actual GUI elements from the logic of the user experience in View and ViewModel respectively, and lets the ViewModel reflect the changes on the Model; this is a common pattern to adopt as it provides a clear organization of functionalities and responsibilities among the internal software submodules.

The front end is deployed as a static web app in the cloud provider web hosting dedicated service. It is directly built and made available from the code repository (through a CI/CD pipeline) reacting at every code change on a specific branch. The cloud provider also takes care of the SSL certificates (to ensure crypted communication) and the global availability through content delivery networks.

The front end module is the one that is the least affected by this thesis work; it is only extended to include an administration section.

Back End

The backend module is composed of two main software modules, which are distinct applications:

- the actual MapNCloud backend, a TypeScript application that exposes a REST API with all the functionalities of the application. This module is packaged as a Docker container; in this scenario, the possibility to bundle together code, configuration files and additional resources in a single Docker image makes deployment faster and easier, also providing all the isolation properties offered by the containers alone.
- a queue managing system, which acts as the ticketing system for the tasks issued to the computation layer. This is a RabbitMQ (a popular open source queue management system) instance, communicating via AMQP[1] with the other modules. The choice to separate these modules was taken to ensure the reliability of the whole system, and to enhance the maintainability of it (RabbitMQ offers out of the box monitoring services, which help to diagnose and follow the instance execution. These were to be reimplemented from scratch if the queuing module were implemented together with the application). This module is also deployed as a Docker container, but in this case the choice was made to reduce the amount of infrastructure needed for the application: the alternatives were deploying a single container group with RabbitMQ and the computational layer together or provisioning a whole virtual machine just for installing a whole instance of RabbitMQ.

The backend is the module that is most affected by this thesis work: in the next section the major changes will be analyzed, which affect the internal organization of components, the use of the tools offered by RabbitMQ and the positioning of the module in the overall architecture.

Data Layer

The Data Layer is the component of the application that persists the data generated by the system, organizes it and make it available to the backend itself to distribute it. It is in charge of storing, as a example, the source files for the photogrammetry processes, the output 3D models, and all the data (stored as hierarchical text data) which describe the users, the projects and the tasks generated by the users themselves.
The DB choice was CouchDB, a non relational database produced by Apache and is written in the same highly parallel technology of RabbitMQ; this also simplifies the deployment of clusters instead of single instances. The choice of non-relational was made to help in the earlier stage of development, where the flexibility of such an approach helped

us in prototyping. As per the queuing system, this module is packaged, configured and deployed as Docker container.

Computation Later

The "fourth layer" is the one that carries out the heavy computational tasks issued by the backend. It is composed by several independent *workers*, which are simply containers, that are separated from the backend to ensure that it does not get stuck on a single request. These containers' internal architecture is rather simple: the chosen engine (during the original development MicMac) is wrapped in an external TypeScript application which divides the various phases of the pipeline in "Stages" and so extends the capabilities of the of the original engine: these Stages can be configured by the user (they are part of the data model memorized in the DB, in fact).
The computational backend has to interact with the queuing system to retrieve the tasks and with the DB to get the resources and output the computed results. This "circular" dependency between backend modules is highlighted here because will be the cause of some major change in the backend architecture. The container technology here is leveraged to ensure a fast spin-up time for a worker (which will be significantly longer for a VM, considering also the complex installation process of such engines) and the reusability of the images, which allows for a fast resize and scale of this layer of the architecture.

2.2. Problems Addressed

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

2.2.1. Scalability

The MapNCloud system was designed to ensure a detachment between the "business logic" backend and the "computational" backend, in order to ensure that that the full application does not get stuck on a request by offloading longer tasks to a different independent module. This module (2.1.1) is the main bottleneck of the system, and though the one I focused on to improve its overall scalability.
The first approach considered was to use an automated orchestration software as Kuber-

netes (1.4.3) or a task scheduler as SLURM (1.4.1) to manage the mapping of tasks on available computational units; as explained in the respective sections, these approaches were not viable, due to either an high infrastructure cost or a poor cloud compatibility. The system was required to be able to handle different traffic paces, with different requests weights, being at the same time cost effective[1].
The target solution needed to dynamically spawn workers, each one tailored to the actual traffic detected or to a forecast on traffic (based on past data or statically, using time of day as index for example). Each worker had to be spawned with a different set of resources, to better handle the traffic. As an example, if a great number of requests suddenly arrive all together, each one with an high number of input files, some resourceful workers needed to be spawned; instead, if the number of unacknowledged tasks grows steadily but slowly, a smaller[2] worker unit can be created to handle the accumulating tasks. This kind of reasoning *must* be applied also in the other direction: especially in the cloud, systems that allocates a significant number of hardware resources are very expensive; if heavy workers (i.e. some modules with high CPU or GPU count) are idling, or the system can satisfy the incoming traffic without them, in order to be cost effective the system should be able to identify the unnecessary workers and kill them. The final design uses an automation tool provided by the cloud vendor to expose a REST interface that allows an alerting module (built in the queuing system) to notify congestion situations, and to also shape the worker spawned in terms of CPU count, GPU count and memory available.

2.2.2. Cloud Provider Integration

The choice of the cloud provider was discussed in the first steps of the project, and after some initial performance tests Azure by Microsoft has been chosen. These simple tests were conducted to see if there was a significant difference in performance between the two vendors: the results are available in the dedicated appendix. In the choice of the cloud vendor other factors have been relevant, as the ease of integration between internal services, general support provided by the vendor itself, migration problems and developer familiarity with the tools.

[1] which translates, in cloud environments, to deallocate unused resources
[2] smaller in the number or size of resources allocated: number of CPUs, amount of RAM and optional GPUs

2.2.3. Queue Monitoring

The queue module is the one that is "congestion aware". The workers are designed to interact with it by dequeuing a message (which contains all the task-related information), processing the task and only when the task has been successfully executed and completed acknowledge the message to the queue. This provides a straightforward metric to both understand congestion of the queues and of the workers: the combination of unacknowledged messages and the ones entirely not delivered to clients. A majority of unacknowledged messages would suggest that the workers are underpowered, while a majority of not delivered messages would suggest that too few workers are deployed. These kind of metrics and indexes must be extracted from the queuing system in order to manage the number of workers present.
Luckily, RabbitMQ embeds a monitoring tool (called Prometheus) which automatically organizes the queues data and exports them in a text format. An additional module of Prometheus, called AlertManager, handles the notifications and interacts with external services; this fits perfectly our use case, allowing us to group together various pieces of information and deliver a precise alert (which will be encapsulated in a HTTP request) that actively reshapes the backend to align to the congestion status.
In the early design steps, it was considered to use a cluster of RabbitMQ instances rather than a single one, to improve on reliability and resilience. The cluster additional copies of RabbitMQ would have provided an higher failure tolerance, since the whole cluster could then withstand a failure of an instance, up to "all but one" instances failure. This option was discarded:

- Using a set of instances in place of a single one requires either to inform the other modules of the fallback options or to use a load balancer put in front of the RabbitMQ cluster to interact with it: in the former case all the backend modules would have needed an additional modification; in the latter, an additional software component should have been inserted in the architecture and configured, with the risk of degrading performance
- Tests were conducted to assure if a single RabbitMQ instance could withstand the load imposed by the application, in different scenarios (as single backend workers versus dynamic resizing backend workers). These tests are available in the appendix
- Clustering RabbitMQ nodes, although a very powerful feature of that software and surely fundamental in large distributed message oriented scenarios, requires some additional per-node configuration which would have itself complicated the deployment of the system

2.2.4. API Redesign and Organization

The module that mostly is impacted by this thesis' work is the backend. From the original design (2.1.1) the module has been separated in two main components

1. the **MainBackend**: this module actually hosts the exposed Application Programming Interface the frontend utilizes. The interface exposed is very similar to the original one, the major changes residing in the data access layer[3] which no longer interfaces directly with the database, but instead calls another API to interact with it for tasks as attachments searching and retrieving, data access (which encapsulates authentication) and uploading of resources.
   This piece is software is deployed as an Azure Web App exposed on the internet (which is in contrast with most of the other components of the architecture, which are deployed inside a virtual network) and is packaged as a container. This is the module that encapsulates the business logic of the application. It exposes an interface to the frontend to be utilized and interacts with the OperationalBackend to handle the data (images, models, user data) and the task management functions, as interfacing with RabbitMQ to add a task to be executed. It interacts with the other components just through REST interfaces.

2. the **OperationalBackend**: this module is deployed as an Azure Web App also, and moreover it is assigned at the same hosting and scaling plan of the Main one (which are called Azure App Services); differently from the Main one, though, it is deployed inside a dedicated subnet of the Virtual Network used to isolate the system. This module however is additionally secured, allowing among all incoming requests only the one originating from the MainBackend to actually be processed. This has two major benefits: firstly, making sure that only requests from a "secure" origin can penetrate the virtual network and reach the OperationalBackend, and then to avoid the waste of resources to process "useless" requests.
   Like the Main one, this backend is packaged and deployed as a Docker container; being this the component that also initializes the database and the queue system, and so uses many different configuration and initialization files, the docker packaging mechanism is useful to keep all the app configuration organized together.
   This component has a variety of roles:

   - it exposes a REST interface for the MainBackend that implements the basic CRUD[4] operations for the data models of the applications, acting like a "proxy"

[3] further details on the original architecture available in appendix (B.1)
[4] Create Read Update Delete

of the database

- the same API has a set of endpoints dedicated to the computational layer modules, the "renderini", to operate (as an example) group resources handling
- it interfaces with RabbitMQ via AMQP channels to add tasks for the computational layer to be executed
- it needs to connect to the CouchDB instance in order to expose its functionalities, and it does so through the dedicated driver offered by Apache itself, Nano, which abstracts the native CouchDB REST API and provides an easy to use TypeScript interface to the database data model

This separation has been done for several reasons; one of the first design choices that was made which regarded the internal architecture was to remove the renderini-database interaction, and having instead them asking for the data directly to the backend, breaking the functions loop (2.1.1) to simplify the architecture and make it more fault tolerant. The problem that arose was that a single module, the backend, was taking care of the frontend load and the renderini load, so it would fast become a bottleneck. As per the separation of the computation layer, mentioned in (2.1.1), the solution was to separate the functionalities in two different modules, in order to have two parallel components that do not interfere with each other.
Another reason that led to this separation was to restrict even more the access to the virtual network: Azure allows for web apps to restrict the incoming traffic in a very fine grained manner, and in this case this feature allowed us not to restrict only the port or addresses pool but even the specific client that could interact with the API, securing even more our perimeter (which already restricts inbound and outbound traffic with Access Control List-like policies provided by Azure, called Network Security Groups rules).

2.3. Final Design

3 | Implementation

A | Tests

A.1. Cloud Provider Comparison

A.2. Queue System Stress Test

B | Models

B.1. Original Application Backend Architecture

B.2. Adoption Model

Bibliography

[1] Amqp documentation. URL https://web.archive.org/web/20141010172100/http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html.

[2] Slurm documentation. URL https://slurm.schedmd.com/containers.html.

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

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