Politecnico di Milano

Facoltà di Ingegneria Industriale e dell'Informazione

POLITECNICO
MILANO 1863

Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Containerization of components for complex applications

Relatore: Di Nitto Elisabetta

Tesi di Laurea di: Gianmarco DELLO PREITE CASTRO
matr. 905131

Anno Accademico 2019-2020

Dedico questa tesi a tutti coloro che mi sono stati di aiuto e supporto nell'ultimo anno.

Ringraziamenti

I would like to thank, first of all, my teacher Elisabetta Di Nitto for being extremely helpful and patient with me during this thesis work. I would also like to thank my parents, for giving the opportunity to follow this universtiry course and being of the utmost support during all of it, helping me in the high and in the low moments of this journey. I thank my sister and my brother for, just like my parents, being extremely supportive. Lastly, I would like to thank Multi-Wing int. and my ex boss Søren for allowing me to take an internship during the work of this thesis, and for being very understanding of my situation, and Motorola Solutions int. and my boss Jon, from my current company, for believing that I would be able to finish my studies on time and giving me the opportunity to start working while on the last steps of my thesis.

Nærum, 25 Novembre 2020

Gianmarco Dello Preite Castro

Sommario

Questa tesi ha come scopo analizzare il problema deployare applicazioni complesse in infrastrutture eterogenee. Negli ultimi anni il cloud ha guadagnato molta attenzione ed è diventato una parte sostanziale della nostra vita quotidiana, essendo responsabile di offrire molti dei servizi che utilizziamo e conservando grande parte dei nostri dati. Come sviluppatori, dobbiamo a volte deployare applicazioni nel cloud. Questo può ovviamente essere fatto manualmente, ma è molto impegnativo. Questo è il motivo per il quale molti tool per automatizzare il processo sono apparsi. Il processo di automatizzare il ciclo di vita di una applicazione cloud non è semplice, ma ha molti vantaggi rispetto a farlo manualmente. Dalla portabilità alla ripetibilità. Con soltanto alcune righe di codice è possibile deployare una applicazione nel cloud. Esistono attualmente una dozzina di tool per automatizzare questo processo. Introdurremmo un progetto Europeo in sviluppo, SODALITE [6], che ha come scopo migliorare lo stato dell'arte, attraverso l'introduzione di un DSL (domain specific language) basato su TOSCA [7] e Ansible [1]. Questa tesi si propone di analizzare e risolvere alcuni dei problemi legati ad uno step essenziale nel processo di automatizzazione di questo processo, cioè quello di containerizzare i componenti prima che possano essere messi nel cloud. Ci concentreremo in una tecnologia usata per questo, Docker [9].

Organizzazione

Questa tesi è organizzata come segue.

- Nel Capitolo 1 Introduciamo il contesto e il problema ed introduciamo la soluzione proposta.

- Nel Capitolo 2 Presenteremo una descrizione del backgroud, nella quale andremo ad analizzare più a fondo il contesto del problema
- Nel Capitolo 3 Introduciamo i principali problemi trovato durante lo sviluppo di questa tesi, e una soluzione proposta per essi.

La tesi si conclude con il Capitolo 4 dove sono illustrati i risultati ottenuti e sono presentati alcuni possibili sviluppi futuri.

Table of Contents

Chapter 1

Introduction

This thesis aims at analyzing the problem of deploying complex applications on heterogeneous infrastructures. Over the past few years the cloud has been gaining a lot of attention and has become a substantial part of our day to day life, from hosting most of the services we run, to even storing most of our data. As software developers, we sometimes need to deploy applications to run on the cloud. And that can obviously be done manually but is a very time-consuming task. This is why recently many tools for automating this process began to arise. The process of automating the lifecycle of a cloud application isn't simple, however it has many advantages over doing it manually. From portability to repeatability. Through a few lines of code, it is possible to deploy an application on the cloud. It is also possible to modify such lines of code to deploy a similar component, and the operation can be repeated as many times as necessary within seconds. There are currently dozens of tools for automating this process. We will introduce a current ongoing European funded project, SODALITE [6], that aims at improving the current state of the art, through the introduction of a DSL (domain specific language) built on top of TOSCA [7] and Ansible [1] in order to simplify the process of creating and writing TOSCA blueprints and Ansible scripts. This Thesis aim is to analyze and solve some problems related to one crucial step in the process of automating such process, that is containerizing components before they are put on the cloud. We will mainly focus on one technology used to do such task, that is Docker [9].

Outline of the Thesis

This thesis is structured in the following way:

- In Chapter 1 We introduce the context and the problem, we then introduce a proposed solution.
- In Chapter 2 We will provide a background description, in which we further analyze the context of the problem.
- In Chapter 3 We introduce the main problems found during this thesis work, and a proposed solution for them.

Finally, in Chapter 4 we illustrate the obtained results and present possible future works.

Chapter 2

Background and Workflow

2.1 Introduction

In the background and workflow section, we will discuss what are the technologies available for the deployment of complex applications on heterogeneous infrastructures. A complex application is an application that is divided into many multiple components that need to interact and communicate among each other, each with their own purpose and set of computations. A complex application could be deployed on different infrastructures, which brings us to what heterogeneous infrastructure means. It means that the infrastructure in which the application is to be deployed is not composed of homogeneous elements, so for example it could be a combination of services in the cloud, on an HPC (high performance computing), embedded systems and so on.

In figure 2.1 we illustrate what is, in general, the life cycle of a cloud application. The most important steps, regarding this thesis work being, Deploy, Monitor and Manage which are strictly related to what is discussed. Deploy being the central problem of this thesis, monitor and manage being steps done ideally by the orchestrator.

2.2 Technologies

The relevant technologies for this thesis work that allow us to deploy complex applications on heterogeneous infrastructures, and that work together with each other are: Ansible [1], TOSCA [7] and Docker [9].

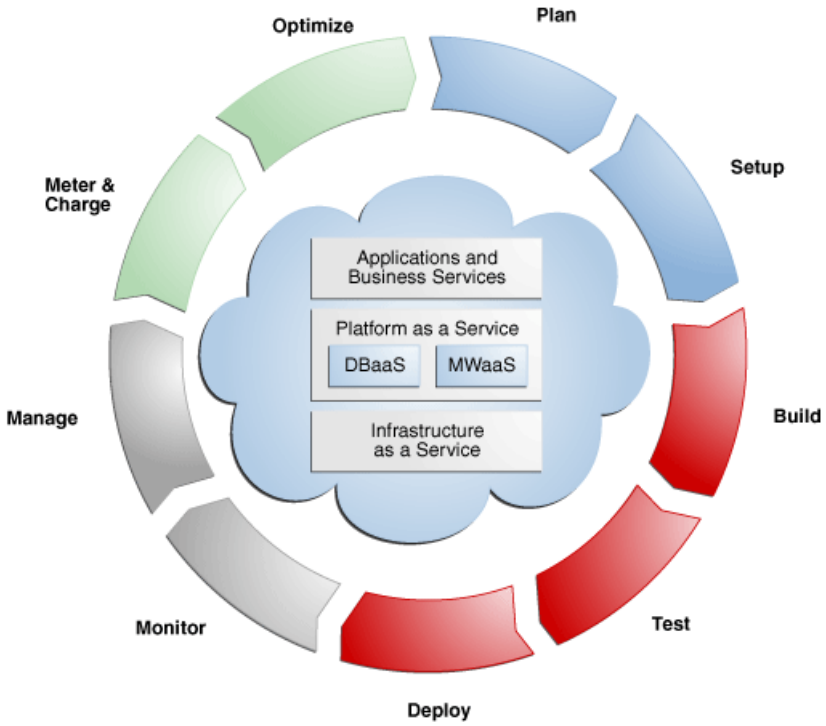

Figure 2.1: Life cycle of cloud application

2.2.1 Ansible

Ansible is an IT automation engine able to automate cloud provisioning, configuration management, application deployment and intra-service orchestration. It uses the YAML language, and is able, through code, to specify how an application is managed on different infrastructures. What this means, is that we are able to write Ansible code that is able to manage the lifecycle of an application, from deployment to undeployment.

2.2.2 Tosca and xOpera

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard that aims at simplifying the management of complex applications. In TOSCA we define service templates that consist of multiple different components, and the relationship between them. The latest specification of TOSCA also relies on the YAML language, in order to have a simple language to specify the interactions between multiple cloud components. TOSCA is a central component of orchestration tools, such as Cloudify and xOpera.

xOpera [10] is an orchestrator being developed by xlab that is compliant with the latest version of the TOSCA YAML 1.2 specification, and currently being updated to be compliant with the 1.3 specification. An orchestrator, such as xOpera, is a tool that automates the management of complex applications. xOpera specifically allows us to run TOSCA blueprints in order to control the whole lifecycle of a cloud application, from deployment to undeployment.

2.2.3 Docker

Docker is a tool that allows users to manage containers. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another [9]. Which means that with these tools we are able to create very small and lightweight containers (much lighter than a virtual machine) that have everything they need to execute a single (simple or complex) task. A big difference between a container and a virtual machine (other than the size) is that containers run on top of the underlying operating system, without the need to simulate a new virtual environment.

2.3 Summary

In this chapter we have discussed some of the available technologies that exist that help us manage complex applications in heterogeneous infrastructures, and how they interact between each other in order to allow us to deploy such applications in such infrastructures. It is important that we note that these are just a few of the technologies available. Furthermore we can conclude that the management of complex applications on heterogeneous infrastructures is not an easy task, and even something as simple as trying to deploy one component may require a lot of work.

Chapter 3

Problems and Solutions

3.1 Introduction

In this chapter we will discuss the main problems faced when containerizing components of a complex application. We will be using as a study case the containerization of the components of one of SODALITE's [6] use cases, the Snow Use Case [2].

3.1.1 Snow Use Case

The Snow Use Case is a project being developed by Politecnico di Milano, that aims at understanding what is going to be the situation in the future for water supply, by analyzing pictures of snowy mountains. The project is divided into many small components, each of which have a very precise task, from collecting data, to extracting relevant information from the data, etc. It enables the perfect opportunity to understand the step of containerizing components, in the bigger automation of the deployment to the cloud process. Figure 3.1 illustrates the pipeline for the components of the Snow Use Case.

3.2 Steps to create all the necessary files

In this section we will describe what are the steps suggested to generate all the code and files needed to be able to deploy an application to an heterogeneous infrastructure.

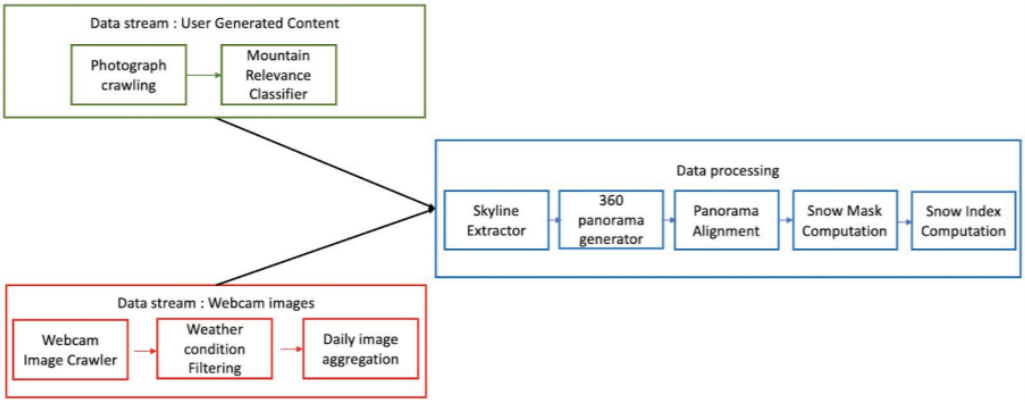

Figure 3.1: Snow Use Case Pipeline

The current workflow for working with all these tools combined is described as follows: Initially we have an application we would like to deploy, it could be a simple application composed of a single component, or it could be a complex application composed of multiple components, possibly relying on different technologies, that somehow need to communicate with each other. One of the crucial steps is to identify, if necessary, on which type of infrastructure each component is to be deployed. It could lead to some differences when following the next steps. We will focus on the steps necessary to deploy components to the cloud.

The first step is to containerize the components of such application, in order to have them able to run on different environments (possibly unknown to us) by requiring as little as possible. We do so using Docker and writing a Dockerfile in which we specify how the application is to be retrieved by the container (it could be with a simple file copy from disk, or a download from the cloud). Following that we install the prerequisites to run the application, such as, for instance, Python, Java, all the necessary libraries and dependencies, etc. Then, if necessary, we make the application able to communicate with the outside, through exposing ports on the container. And finally we install the application (if it requires it). It may be necessary to provide what are called Entrypoints to our countainer, that is a way to execute the container as if it were an application.

The following step is to write Ansible YAML scripts for each of our components, in which we specify tasks to manage the lifecycle of our application, such as for instance deploy and undeploy. In our case, in which the components

have already been containerized, and in which we will use an orchestrator, these tasks will most likely be just the initialization and termination of our containers.

We then write a TOSCA YAML blueprint, that is a description of the nodes of our system. In a complex application this will include calling the Ansible scripts at the appropriate time, as well specifying the ways in which our components communicate with each other.

Finally we use an orchestration tool, such as xOpera, to orchestrate (that is, manage) our application.

In figure 3.2 we illustrate, with an activity diagram, what is described in this section. The central focus of this thesis are the first tasks done in parallel, that is, the containerization of each component. After that is done we create Ansible scripts for each component, with one script for each step in the life cycle of the application. And lastly we create the TOSCA blueprint.

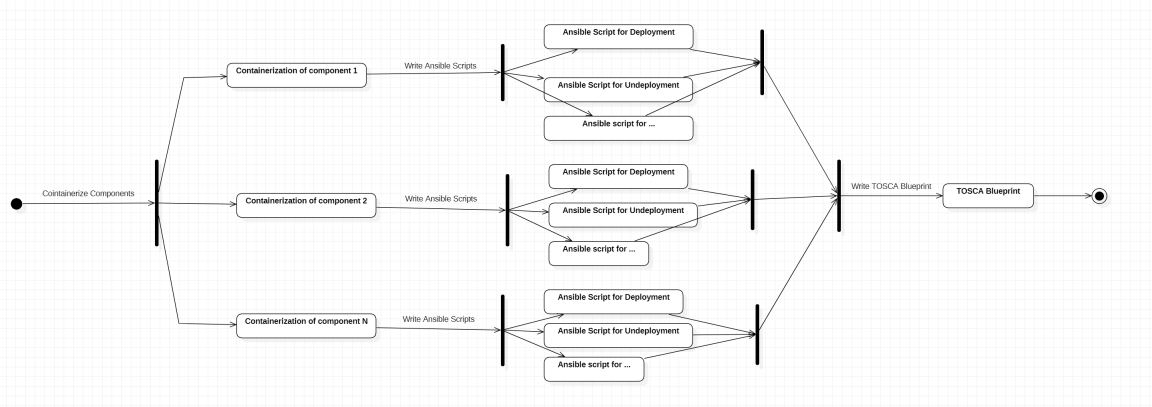

Figure 3.2: Activity Diagram Steps

3.3 Problems

In the following paragraphs we will describe what are the problems we encountered when trying to containerize components of a complex application. The running example which we will use to validate the proposed solution is the Snow Use Case from SODALITE.

3.3.1 Problem 1: Authentication

The first problem we will analyze is the most basic one when dealing with the creation of a dockerfile: getting our component inside the container. The component iself may be a source file, a java project, or a python script, etc. but we should be able to abstract from this. The component being moved inside the container can be done in a multitude of ways. We will introduce and analyze 2 of them:

- copying from the host file system.
- downloading from the web.

Copying from file system

The first option at our disposal is to just copy the component from the host file system into the docker file. This can be done with the docker COPY instruction [3]. It does precisely what the name indicates, it copies a file from the host into the container.

The advantages of using this method are:

- it is incredibly simple, and relies on a single command.
- we have full control over what gets put in the container and where.
- authentication is often simple.

The disadvantages of using this method are:

- we need to have the component in our host machine when creating the container.
- updating the component may become hard, as it will be necessary to redo the copying process whenever there is an update.
- it may be inconvenient to have the component on the host file system, especially if it is, for example, very large.

Downloading from the web

The second option at our disposal is to download the component from the web. This in itself can mean many things, for example a direct download from a web service, a clone from a code repository, etc. We chose to focus on the case of cloning code from a repository, as this was the used method in the work analyzed by this thesis. The process requires to first have the URL of the repository in question and then to execute the cloning procedure, through for example, a git clone.

The advantages of using this method are:

- it can be done anywhere at any time.
- updating the component is simple.
- it does not require the host to have a copy of the component.

The disadvantages of using this method are:

- the sequence of commandd to execute the download can be more complex.
- there may be issues when downloading a file, for example connection errors.
- there may be restrictions on downloading very large files.
- it is necessary to have a service somewhere hosting the download procedure.
- authentication can be complex.

Authentication

The problem that can be encountered in both cases, and the subject for analysis in this section, is that of authentication. Often when trying to access a resource that is in some way of restricted access, it is necessary to provide some sort of authentication. The issue arises when trying to automate the process, by creating our container through a dockerfile, as it will be needed to provide the authentication credentials. As mandated by common cybersecurity knowledge,

storing authentication credentials in clear text is a terrible idea. Therefore it is necessarry to introduce some sort of security measure.

The proposed solution is the utilization of docker secrets [5]. Docker secrets are a way of managing sensitive information that should not be transmitted over a network or stored unencrypted. What secrets do is make sure that the sensitive data is stored encrypted. When the data is needed, but the selected service that has access to it, the data is decrypted, used and the disposed of. Much more detailed information on secrets can be found on [5].

In the Snow Use Case, both approaches were tested. At first the acquisition of the components was being done through a clone from a private GitLab repository, using docker secrets. Such approach proved to be very inconvenient, and particularly difficult, so at a later stage it was changed to a copy from host file system, which required no authentication.

3.3.2 Problem 2: Understanding and installing prerequisites

The second problem we will analyze is that of understanding what are the prerequisites for the containerized component to work, and installing them on the container.

Let us assume we were able to copy the component into a container, the next obvious step is to check if it runs. We will in most cases be greeted with a negative answer. As an example, a Dockerfile starting with FROM Ubuntu will introduce just the bare minimum tools for us to have an Ubuntu environment, which in most cases is not enough to run an application. This is the step in which a small understanding of the application to be installed is necessary. Such information can usually be found by reading the applications documentation or asking the developers, for instance. If we are able to do either one of those, the process becomes simpler. Most documentation will define what are the prerequisites to run the application (for example having the latest java or python version and having a selection of libraries).

Once those are established, the process moves forward. We add to our dockerfile the commands necessary to install all the needed tools. (insert snippet possibly). Once we believe we have all of them in place, we add the command to run our application, and our dockerfile should be ready. We try to execute it,

and this is when we may be greeted with messages that we do not expect. (add screenshot of possible error). For example that we are missing a crucial library, even though we made sure to include all the needed libraries installation in our dockerfile. This is usually due to the big difference there is between a container and a host environment. An Ubuntu host environment includes a multitude of things an Ubuntu container environment does not. Aside from that, a single command may produce different results in a container than it would on a host environment, given the differences between the two. Aside from this possible scenario, we may be greeted with a worse one, in which we have 0 knowledge about the prerequisites of the application.

Which brings us to the proposed solution, an iterative trial and error process of installing prerequisites. This may seem unorthodox at first, but it produces the necessary results without needing to have an incredibly deep understanding of an application, libraries, docker environments, etc. The process is somewhat straightforward: start by making sure the dockerfile is able to get the component, through whichever method, then install the obvious prerequisites (python, java, etc.). From there enter a cycle in which first we try to run the application, if it works, exit the cycle, if it does not we need to be able to analyze the error message(s) we get. They are a crucial part of this process as they will tell you what went wrong, which is usually a problem with missing prerequisites, like for instance a missing library. From there you proceed by fixing the issue and repeat until the application is able to run correctly. (Possibly add small diagram, error messages, and code snippets)

Another possible solution to this problem we wish to illustrate that exists, is the usage of dependency tracking tools. We will however briefly discuss why they are not the proposed solution in our work. Dependency tracking tools are a current state of the art tool that allows for checking dependencies of a project. However they come with very big drawbacks currently. The major issue which makes them unusable for this thesis work is the variety of languages and tools used in each component of the SnowUC (and many other multi-component projects). The SnowUC components included components written in (but not only) python, java and javascript (with NodeJS). Finding and using the right dependency tracking tool for each of the languages would have been an hard and time demanding process, which brings us to the second problem we want to discuss with dependency tracking: there aren't many available, and they

usually only work for one of the languages. They are also a fairly new concept that hasn't been explored much in the open source world. We were able to identify a software tool that might have been able to solve the issue, dependecy track [8]. It is however an extremely complex tool with many more features necessary to execute our task of finding library dependencies. Further work in the subject could aim at introducing this tool into the workflow in order to further automate it.

3.3.3 Problem 3: Communication between components

The third and final problem we would like to discuss in this thesis is how we need to be aware and manage future communication between components when containerizing them. In almost all possible scenarios, a docker container needs to communicate with the external world, be it to get inputs or provide outputs. In the SnowUC scenario discussed as the main example in this thesis, the components belong in a pipeline, which means in most cases they need to do both: get inputs from the previous component and provide outputs for the next component.

Since in the grand scheme of things the SnowUC is to be deployed on the cloud, the communication between components often relies on the internet, in what is a very simple process. It relies almost entirely on exposing ports which the next component in the pipeline uses to retrieve the necessary information. So the proposed solution, although simple, provides a very powerful mean to manage communications between components. The first step is identifying if it has already been established when the component was created which are the communications it needs to do. In a fully thought out application, this comes at a requirement analysis step, and should already have been planned and implemented once the components are ready for being containerized. The second step is verifying that ports have already been established for the communications, which is in most cases already done. The third step is to make sure those ports are accessible on the outside world, and that is done with a provided Docker instruction called EXPOSE [4]. As the documentation states, that is not enough to publish the port. To do so, the flag -p has to be used on the docker run command to publish and map a port, for example:

```
docker run -p 8081:8080 -v "$PWD/Builds":/home/snow-skyline-alignment/build/
```

```
-it snow-skyline-alignment
```

The fourth step is to verify that it is possible to communicate with the component on that port, usually done by running the component and checking that the results are provided in final address.

The fifth and final step is to verify that the whole pipeline is working. (NB: this step has not been done for this thesis work, as the pipeline for the SnowUC had not been completed at the time of writing and the pipeline relies on higher level orchestration tools).

3.4 Implementation

Finally in this last section we will describe what has been implemeneted. As stated previously we implemented the dockerfiles for a few of the Snow Use Case components, and our work can be found in the public SODALITE repository: `https://github.com/SODALITE-EU/iac-management/tree/master/use-cases/snow-uc/build-images/components`

In that folder of the repository are the dockerfiles for the creation of the docker components, together with a small .sh script to build and run the image itself. The first components to be containerized were the skyline-extractor and skyline-alignment, and those were containerized by a previous research assistant. Later, when I took over the containerization, on a first stage, we containerized the webcam-crawler, the weather-condition-filter and the daily-median-aggregator. Those at first downloaded the components from the Snow Use Case GitLab repository. It was changed at a later stage by one of xLab employees to use a copy from file system. Lastly we containerized on a final step before concluding this thesis works, the flicker-crawler, the snow-index-computation, the snow-mask-computation and the snowwatch-render.

The dockerfiles themselves consist of a few common commands:

- FROM Ubuntu, specifying that the docker is to be built from linux distribution ubunti.
- COPY, in which we copy from file system the components into the container

- Prerequisites install, in which we install all the prerequisites necessary to install/run the component
- [Optional] Installation of the component, if required.
- [Optional] Entrypoints are provided as comments, and are to be used at a later stage if necessary.

3.5 Summary

In this chapter we have firstly introduced the Snow Use Case, which was used as validation for this thesis work, secondly the suggested steps to follow when attempting to deploy a complex application on a heterogeneous infrastructure. Then we discussed 3 of the problems found during this thesis work, when trying to containerize components for the Snow Use Case. We started by providing a general overview of the problem, and finally used the Snow Use Case as a validation of the proposed solution. The three problems presented were:

- Authentication, that is how do we provide the necessary information to authenticate ourselves when retrieving a component from it's source, without creating any security issues.
- Understanding and installing prerequisites, that is how do we retrieve what are all the necesarry dependencies that a component needs to run, and how do we install them on the container.
- Communication between components, that is how do we allow components to communicate between each other, which is a crucial part of deploying complex applications.

Lastly we presented the real implementation that has been done for this thesis work, providing a link to the repository containing all of it.

Chapter 4

Conclusions and Future Works

In conclusion we can say that the task that would sound somewhat simple, that is of containerizing components for a complex application, proved to be a lot harder than expected. Multiple issues were found, and in this thesis work we discussed the main ones and provided possible solutions for them (Authentication, installation of prerequisites and communication between components). At the end, it was possible to containerize the components after following the correct approach, and their use may become relevant in the context of SODALITE.

The results obtained were many. We were able, first of all, to create dockerfiles that will possibly be used in the SODALITE work and therefore were able to contribute to the overall work that is SODALITE. We were able to study the process of creating the necessary files to deploy a complex application on a heterogeneous infrastructure, understanding also what could be one good approach to follow in order to create all these files. We extended our knowledge in the state of the art when it comes to orchestration in general, from the first steps to the last. And finally we were also able to identify many possible issues that arise in the process of creating dockerfiles, to containerize components of a complex application, and document our findings so that may be useful in the future for ourselves or for others.

There are multiple ways in which this thesis work could be continued. The first being on picking up tools for dependency tracking and trying to find if there is a way to introduce them on the procedure of installing prerequisites, in order to avoid stumbling upon issues regarding missing libraries and such, in order to further simplify the process in the end. The second would be

to complete the following steps proposed in the workflow, that is to write Ansible and TOSCA code to complete the deployment of the application, and controlling it's life cycle. The third and final would be to complete the work of the Snow Use Case by finishing containerizing all the components, and executing them in the pipeline.

Bibliography

[1] Ansible. https://www.ansible.com/.

[2] SODALITE Snow Use Case. https://www.sodalite.eu/water-availability-prediction-mountains-images.

[3] Docker COPY reference. https://docs.docker.com/engine/reference/builder/#copy.

[4] Docker EXPOSE reference. https://docs.docker.com/engine/reference/builder/#expose.

[5] Docker Secrets reference. https://docs.docker.com/engine/swarm/secrets/.

[6] SODALITE. https://www.sodalite.eu.

[7] TOSCA. http://docs.oasis-open.org/tosca/tosca-simple-profile-yaml/v1.2/csd01/tosca-simple-profile-yaml-v1.2-csd01.html.

[8] Depedency track project. https://github.com/dependencytrack/dependency-track.

[9] Docker Website. https://www.docker.com/resources/what-container.

[10] xOpera. https://github.com/xlab-si/xopera-opera.