Master Project Report

September 4, 2016

The Global Living Application Platform (GLAPP)

Fabio Isler (09-115-965)
Man Wai Li (14-705-156)
Dinesh Pothineni (14-707-988)
Riccardo Patane (09-177-270)

supervised by
Prof. Dr. Harald C. Gall
Dr. Philipp Leitner

University of Zurich UZH

s.e.a.l. software evolution & architecture lab

Abstract

This master project takes the idea of a so-called global living cloud application (GLA), a bio-inspired notion of cloud applications, and presents a platform called GLAPP to deploy and constantly optimize the deployment of such GLAs. The work included the development of a frontend to serve the users of this platform, a backend to effectively deploy those GLAs, and a control loop to constantly monitor and adapt this deployment. The goal of GLAPP is to go some steps further than conventional cloud application platforms by supporting a deployment across several cloud providers and by offering a constantly optimized deployment with regards to user-defined policies. Examples of such optimizations are scaling actions or moving certain parts of a GLA across regions in order to satisfy the declared policies. This report introduces the underlying concepts of a GLA and GLAPP, and provides details on the design, the architecture and the concrete components of GLAPP.

Chapter 1

Introduction

Cloud computing has created a paradigm shift in the last few years, by making infrastructure available at lower costs and with higher efficiency of operations. These solutions are increasingly being adopted by enterprises and developers, as they can provision a huge amount of resources to scale on-demand in order to meet their business needs. In cloud computing, resources such as CPU processing time, disk space, or networking capabilities, are rented and released as a service. Today, the most important model for delivering on the cloud promise is the Infrastructure-as-a-Service (IaaS) paradigm. In IaaS, virtual computing resources are acquired and released on demand, either via an Application Programming Interface (API) or web interface. Along with great flexibility of being able to get new resources on demand and pay for what you use, new problems arise. Selecting a cloud service provider can often be a quite challenging decision for the developer or a company, so is being able to monitor and evaluate these infrastructure resources on a regular basis. With so much of variance in cost and performance, it is imperative that one would look for a reliable deployment, monitoring solutions to strike a balance with application requirements. Furthermore, the complexity and skill required to manage multiple layers of application, data, middleware and operating systems can be very high.

Understanding run time performance and behavior of various application components in realtime can enable us to take advantage of arbitrage opportunities that exist between different machines/regions or even other cloud providers. For example, an application can take advantage by moving closer to its users based on timezones or traffic to improve response time. Currently there is no flexibility to move freely between various cloud providers without great development effort and cost, however such an ability to move freely between providers enables us to benefit from cost and performance differences of cloud providers. We propose a cloud middleware that can not only take care of application deployment to the cloud, but also constantly monitor and trigger necessary adaptations to benefit from these opportunities. This middleware also enables developers to specify their intended goals in terms of high level policies to govern the application behavior. The middleware can then break down these policies into low level objectives, in order to trigger adaptations by changing the state of application when required.

1.1 Global Living Cloud Applications

The aim of this project is to develop a platform for what we call *global living cloud applications* (GLAs). In a nutshell, GLAs are a bio-inspired notion of cloud-native applications. GLAs live in the cloud, and are able to migrate between data centers and cloud providers automatically, based on changes in cost and performance of cloud offerings, changes in customer behavior or requirements, or other factors.

The bio-inspired terminology applies for the different levels of components of a GLA:

- **GLA**: The GLA itself is a collection of organs that form the whole application.

- **Organ**: An organ consists of one or more cells and is responsible for a service.
- **Cell**: A cell is the lowest-level component of a GLA. There are two kinds of cells: The ones that provide user-defined functionality of the organ, and platform-defined cells like reverse-proxies for exposed ports.

1.2 GLAPP

In order to manage these GLAs, we introduce a platform called GLAPP (**G**lobal **L**iving **AP**plication **P**latform). It allows a developer to deploy multiple GLAs on whatever cloud she has access to and sets a centralized mechanism in place to constantly monitor and manage all the GLAs. The platform supports heterogeneous environments, so a GLA can live and move across different providers, regions, instance types, etc.

GLAPP is an open-source platform which can be deployed easily on a server. Set-up instructions can be found on github[1], as well as all the source code that was developed in the course of this project[2].

1.3 Outline

This report serves the purpose of giving an overview of the whole project. In chapter 2, we give some background information about design and implementation decision as well as the architecture of GLAPP. Chapter 3 introduces all the components GLAPP consists of in more detail. In chapter 4, we present the case study we had in mind and demoed in our final presentation. Chapter 5 discusses possible extensions and future work on the project. The report is concluded in chapter 6.

[1]https://github.com/glapp/gla-sails
[2]https://github.com/glapp

Chapter 2

Background & Architecture

2.1 Basic Design Decisions

We decided to build GLAPP on three pillars, as visualized in figure 2.1: A backend server that handles everything related to the deployment of the GLAs and therefore also talks to the host infrastructure, a frontend to serve the users of the platform and give a visual overview over the state of their GLAs, and a control loop called MAPE that monitors the deployed GLAs and triggers adaptions via the backend server. These components will be introduced in section 2.3 and discussed in detail in chapter 3.

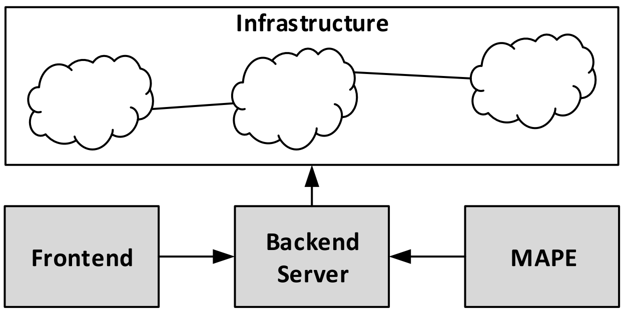

Figure 2.1: Main components of GLAPP

Several design decision had to be made considering those main components and the infrastructure. This section introduces some of these decision processes.

2.1.1 Deployment: Containerization with Docker

Containerization vs. other virtualization methods:

We set out to create a platform to manage GLAs. Speed of deployment is therefore an important criterion when considering the underlying technology. While both virtual machine and containerization are virtualization technologies that provision shared cloud computing resources, virtual machine focuses on virtualization of infrastructure resources and containerization focuses on virtualization of an application [Pah15]. Being a lightweight virtualization technology, containerization gets rid of the guest OS layer that exists in traditional virtual machine architecture. Instead, applications are deployed on top of a container engine which runs on the host OS. In addition, application binaries are stored in a package along with the necessary libraries in the case of containerization. This contrasts to the full OS required for a virtual machine. These lightweight runtime and packaging provides a fast deployment that is a crucial aspect of GLAPP. With this containerization technology, GLAPP can deploy application components in seconds when the application images are present on the host. In addition to the fast deployment, containerization has a much smaller footprint compared to virtual machines. A normal virtual machine image has a size in gigabytes due to the presence of a full OS. In contrasts, container images used in our case study take up only a few hundred megabytes of space as a container image includes only the application binaries and required libraries.

Docker vs. other containerization technologies:

In containerization, there are various models and technologies. For example, both Docker[1] and OpenVZ[2] are containerization technologies but they use different models. However, Docker containers mostly function as an application or a service while OpenVZ containers function as a virtual private server (VPS) for deploying additional application stacks on top of it [Ope16]. Similar to OpenVZ, LXD[3] is another machine containerization technology. As GLAPP aims at managing GLAs, Docker provides the most direct way of containerization of applications. With other containerization technologies like OpenVZ or LXD, the platform would need to manage both the application deployment on containers as well as deployment of those container on cloud infrastructure. This would add another layer of overhead and complexity.

2.1.2 Orchestration: Docker Swarm

For GLAPP to manage the GLAs in response to the needs of application or changing environment, GLAs need to be run in cluster across different cloud computing resources. Docker Swarm[4] is the orchestration tool used to manage this cluster of containers. An orchestration tool is used to control the deployment of containers across the cloud infrastructure to satisfy various objective such as load-balancing. While there exists other tools such as Kubernetes[5] that provides orchestration functionalities, Docker Swarm provides additional capability to manage hosts of different cloud service providers. For example, Docker Swarm can also create and delete hosts and manage the members in a cluster. This enables extension possibility of MAPE to manage not only the containers, but also underlying cloud computing resources.

2.1.3 Rule-Based Adaptations vs. Markov Decision Process

Markov Decision Process (MDP) and rule-based adaptations can both be used to determine an appropriate adaptation action to change the application from an undesired state where policy is violated to a desired state where the policy is fulfilled. MDP is a reinforcement learning method whereas the rule-based adaptations are represented by a set of rules that are coded into the MAPE. A reinforcement learning problem is defined as the task of learning from interactions to achieve a goal [SB98]. MDP is a reinforcement learning task that satisfy the Markov property [SB98]. For the development of the MAPE loop we used the BURLAP reinforcement learning library[6] of Brown University. The work consisted in:

1. representing the current state of the application (s)
2. define the possible actions (a)
3. define the transitions from the current state to the next state (s, a, s')
4. applying learning algorithm, e.g. Q-Learning or SARSA

[1]https://www.docker.com/
[2]https://openvz.org/
[3]http://www.ubuntu.com/cloud/lxd
[4]https://www.docker.com/products/docker-swarm
[5]http://kubernetes.io/
[6]http://burlap.cs.brown.edu/

Representing the current state resulted in a huge state space. Considering that MDP learns from past rewards of getting to a specific state, we realized that our MDP would be inefficient. The main reason is that it would take too much exploring time until the system could begin to apply the knowledge acquired. Every adaptation the MAPE would trigger would also give the reward for the new state. If we assume that in the normal operation of the system a rule violation should not happen very frequently, then it could last years until the whole state-action space is explored, if at all. A potential solution to this is using function approximation. With function approximation, the system does not need to explore all state-action pairs anymore. Instead, when a new state-action pair is encountered it approximates the Q-value based on a similar state-action pair. BURLAP in its current version 3 supports standard function approximation only for numeric representation of the features of the state. Since we have nominal values (strings) for the representation, alternative function approximation is needed which requires further study and implementation. We decided to not follow that path further since it seems to exceeds the scope of this project. Details of this problem is discussed later on in section 3.3.4.

As a result, rule-based adaptations was used in the final implementation. Instead of a reinforcement learning approach, a heuristic is used for the determination of the adaptation action. Compared to the Q-Learning algorithms with its lookup table, the Rule-based adaptation is fast in finding an adaptation action to address the violated policy. The action generated by the rule-based adaptation will be effective when the rules are well-defined. In contrast, MDP requires the convergence of Q-values in order to generate effective adaptation actions. At the same time, the heuristic rules used are easy to understand for further tweaking. However, rule-based adaptation also has its disadvantages. For instance, it does not scale well when the policy becomes more complex, or when new types of metrics are introduced. In such case, MDP will scale well as the Q-value will converge over time. In other words, even with additional metrics and complex policy, once Q-Learning have the converged Q-values, it is expected to generate effective adaptation action. A further downside of rule-based adaptation is that the system does not react or has an undefined behavior in case the MAPE encounters an unknown situation.

2.2 Implementation Decisions

2.2.1 Backend: Node.js

The backend has been implemented with the *Node.js*[7] runtime. More specifically, we used a framework called *Sails*[8]. Although it is labeled as a complete MVC framework, we mainly used it to implement the server-side functionality and to provide a clean REST API for the other GLAPP components.

There are two reasons why we chose Node.js and Sails in particular. For one thing, it was the backend framework some of us were most experienced in, which made it easier to focus on the actual project without having to work our way into a new framework. For another thing, Node.js comes with the well-established packaging system *npm*[9], where we found very helpful packages for our problem domain. As an example, *dockerode*[10] was the module which we used to implement the Docker-specific functionality, and it proved to be very well maintained, adding new functionalities whenever a new Docker release was published. Since we were about to work with relatively new technologies like Docker, it seemed reasonable to count on packages which we knew will be adapted to changes in these new technologies.

[7]https://nodejs.org/
[8]http://sailsjs.org/
[9]https://www.npmjs.com/
[10]https://www.npmjs.com/package/dockerode

2.2.2 GUI: AngularJS

This component of the application facilitates users to interact with rest of the platform. Since the whole backend already provides REST endpoints for all the services, the platform is fairly relaxed in choosing a framework for the frontend. AngularJS[11] was particularly chosen for two important reasons. Since we had already decided to use Node.js for the server component, it helps a great deal in terms of productivity to be able to use the same language for other components. The team could easily switch between the server and front-end while developing features. This can greatly aid us to iterate faster on the feedback received.

Some of the benefits of using the Angular framework are as follows. Data binding is the foremost advantage in the application, meaning any dynamic fields in the interface can be watched and automatically updated as the new data comes in. Directives for DOM manipulation and two way data binding makes a lot of known hardships a breeze with Angular. Angular code can be very modular, reusable with easy-to-test features. The template system provides great reusability of various components. We used Bower and NPM for package management, as they provide comprehensive set of libraries for all the application needs. Routing is built-in to angular, which makes a single page application possible. This can make an application run extremely smooth and fast, once its loaded on the browser. Directives are very useful to define custom HTML tags and reuse the most frequent features in the views. In addition to this, modularity, abstraction of services makes a strong case for Angular. Last but not least, it is supported and maintained by Google, which can guarantee a long term road map and future development of the framework.

2.2.3 MAPE: Java

For the MAPE, we decided to use Java. Java is a widely used programming language with good interoperability and decent support by other system or libraries used in this project. One of the important aspect of MAPE is the ability to easily interface with other components and systems inside or outside the GLAPP platform. Another consideration is the availability of third party libraries that GLAPP can leverage to perform common computation using well-known algorithm such as various planning and learning algorithms used in Markov Decision Process (MDP).

A set of comprehensive functionality is provided through Java API and third party libraries. In addition to functionality included in Java standard API such as HTTP connection used for interfacing with the sails backend and monitoring system, useful third party libraries are also available. For instance, the GSON library[12] provides an easy API to create and manipulate JSON objects that is used for data exchange between MAPE and sails backend as well as between MAPE and the monitoring system. Most importantly, the BURLAP library[13] provides not only a set of planning and learning algorithm for Markov Decision Process in reinforcement learning, but also a framework for further extending the processing capability through custom implementation of various components including learning algorithm and approximation function. The availability of these functionalities and third party libraries makes Java a compelling language in developing MAPE component.

2.3 Detailed Architecture

As introduced, the platform consists of 3 different parts/blocks: a frontend, a server backend and a control loop. The frontend provides an interface for developer to interact with the middleware to deploy and manage her GLAs. The server backend provides the management functionalities of the middleware including cloud infrastructure management, application deployment and access to the application status information. Lastly the control loop is the com-

[11]https://angularjs.org/
[12]https://github.com/google/gson
[13]http://burlap.cs.brown.edu/

ponent responsible for enabling the management of GLAs by the platform itself. It follows the MAPE (Monitoring, Analysis, Planning and Execution) principle. Possible execution actions are moving cells between different cloud instances (migration), duplicating/splitting cells of the GLA (mitosis), or removing cells.

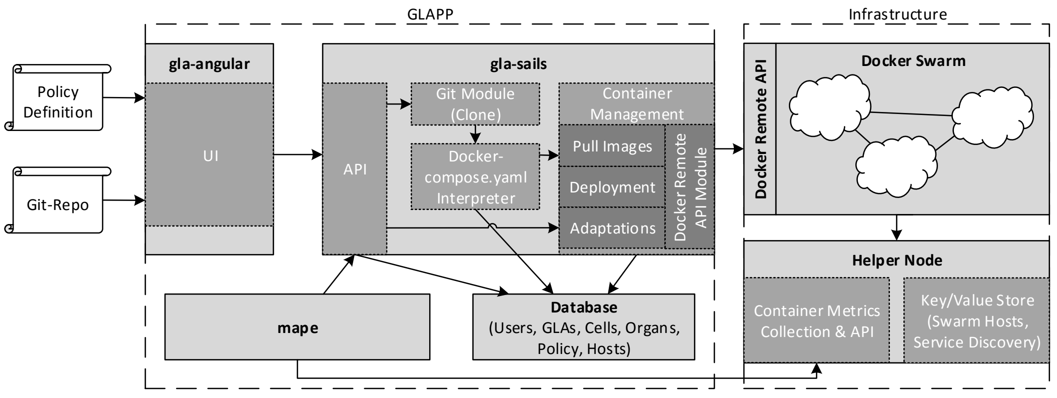

Figure 2.2: Detailed Architecture of GLAPP and the Infrastructure

Figure 2.2 shows the detailed architecture and building blocks of GLAPP. The platform is designed to be modular and interoperable from the very beginning. Every component is connected to each other through a series of API calls, so that every critical component in the platform can be modified without large dependency chains, or even completely replaced with a better one if needed. A simple user interaction flow through the platform architecture looks as follows. Once the developer decides to use the platform, they interact with the user interface (UI) to create a new global living application (GLA). One can create a new GLA by simply providing the URL address of their code repository. If it's a valid repository, the front-end module passes the repository URL to the sails server through its API. The Git Module in the sails server then clones this repository to find the compose file. The platform presumes that this repository contains a docker compose file, which is a means of specifying configuration of main components in their application. The structure of this compose file specifies the important services in the application, the sources of each of their images, various environment variables, ports to be exposed and dependencies to make them run smoothly. The Docker-compose Interpreter module will perform this task of analyzing the compose file and storing the necessary information into the database. It creates the high level definition of the GLA, with its main organs and other important information.

The Database module is responsible for storing every piece of information related to the application, users, except the cell images themselves. This is handled by the docker remote API, which is a part of the Container Management module. The Container management is an important liaison between the infrastructure and the sails server. It is responsible for pulling images mentioned in the compose file and storing them in the infrastructure, in a ready-to-deploy state. It also executes the initial deployment and any future changes to deployment in terms of adaptations. Each image pulled from the compose file is stored as an elementary cell, from which future cells of the GLA are created. Each cell stored in the infrastructure can be re-used if any other GLAs has a specification of the same cell. For example, if your GLA has a postgres database mentioned as a cell this will be stored and made readily available for any other GLAs using a postgres cell. In this way, we cut down the waiting time for images to be pulled down to a fraction of what it would usually take and many GLAs can be created and deployed instantaneously in the cloud.

The Infrastructure is the bottom layer in GLAPP platform. Our architecture uses the docker swarm technology underneath to create a cluster of servers across various cloud providers with a common internal network routing in place. This allows us to have servers in any part of the world, or with any provider like Google, Amazon, DigitalOcean and still be able to talk to each other as they are part of one entity. In the cluster, there is one swarm master and multiple swarm-agents which talk to each other through the swarm master. We deployed a key value store in one of host machines to keep track of the host information and provide service discovery as it will be described later in section 3.4.2. The whole infrastructure cluster with multiple hosts and the application services running inside these host machines are organized into one major module. The sails server communicates with the infrastructure through the Docker Remote API.

Additionally we also use Prometheus[14], a powerful monitoring solution and a time series database to feed our application metrics data. In order to make this happen, we run a Prometheus agent on every host machine, which periodically monitors all the containers that are currently running on a particular host to feed respective data into this time series database. We use cAdvisor[15] for this function. It is a daemon, that collects information related to the resource usage, network statistics and performance of the running containers in the machine. It has native support for docker containers, which makes it a perfect bid for the job. The Prometheus server scrapes data from all agents every few seconds. The best part is, Prometheus not only collects the data, but also provides a powerful query language to aggregate and query the metric data. This monitoring module is also extensible with external application metrics, as a REST API is built into the Prometheus tool. The GLAPP platform facilitates a way to feed third party application metrics into the platform, with the help of this.

During the creation of a new GLA, a developer can specify high level application policies to guide their GLAs. For example, a policy related to cost can be set as not to exceed 1 USD per hour, irrespective of the load or the number of cells running at the moment. It's then the job of the GLAPP platform to continuously optimize the deployment strategy, so that this policy is never violated. This means that it should strike a balance between cheaper instances, high availability in all regions for the users while ensuring other goals if there are any. This is where MAPE comes into the picture. It serves as the nucleus of the platform. As soon as the application is deployed, MAPE looks for policies set on the GLA. If there are any, it then retrieves the related application metric data from the Prometheus server. The analysis module then compares the policies with metric data to detect if there are any current violations. In the event of a violation, the planning module then recomputes the optimal deployment strategy and necessary adaptations to achieve that goal. The execution module then executes the planned adaptations by communicating the specifics to the sails server, which then executes the actual changes in the infrastructure. The planning module is the crucial step here, and this is extensible. That means that one can swap this part with a different planning algorithm of their choice to optimize the deployment. As introduced, MAPE currently supports a basic form of markov decision process using Q-learning and a rule based algorithm. This whole cycle of monitoring, analysis, planning and execution is repeated until the application reaches a healthy state. It ensures the application is always healthy and adheres to the set policies. More details on the individual components and their implementation follows in the coming chapter.

[14]https://prometheus.io/
[15]https://github.com/google/cadvisor

Chapter 3

Components

3.1 Backend Server

The backend is mainly responsible for the deployment and is the central component that defines the data structure of the platform. In order to understand the main functionality of the backend server, this section introduces the data models, the REST API, the services and the tests of this component.

3.1.1 Data Models

Figure 3.1 provides a simple ER-like overview over the way how the server stores the data about users, applications, etc. The mentioned attributes are exemplary, but represent the most important ones to understand the whole model. It is apparent that the model is tightly coupled to the introduced notion of a GLA - an application which consists of organs, which again consist of cells.

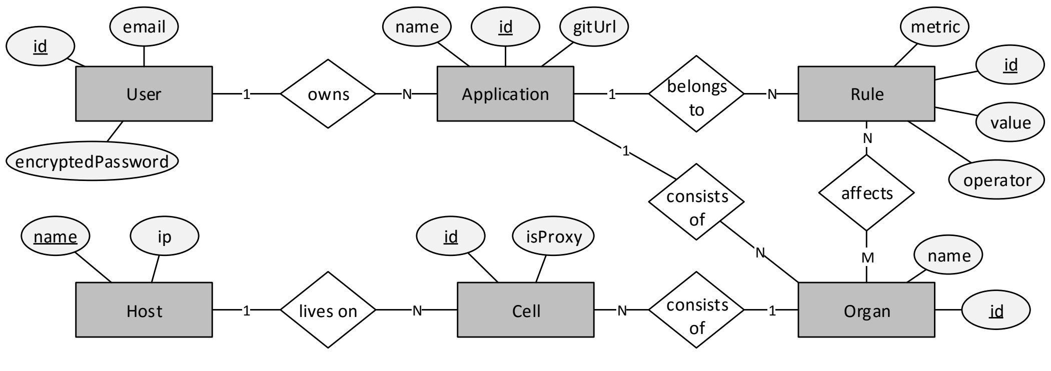

Figure 3.1: Simple overview over the models and their relations

Sails is built upon an ORM called *waterline*[1], which allowed us to manage the data on a relatively high level. That means that by defining the models and their relations and by setting up a connection to any database in the configuration of Sails, the data could be stored and retrieved by simple function calls. We decided to use *MongoDB*[2] as a database.

[1]https://github.com/balderdashy/waterline
[2]https://www.mongodb.com/community

3.1.2 REST API

The backend is supposed to serve the frontend and the MAPE component with data and to provide an interface for triggering adaptations to the deployment. To achieve this clean interface, there is a REST API in place. Table 3.1 lists and describes the endpoints of this API. Since our components are completely modularized and the communication only happens via this interface, the remaining components like the frontend or the MAPE are easily interchangeable.

3.1.3 Services

In the Sails framework, services are used to handle specific, API-independent functionalities. For most of the internal logic apart from the actual API of the backend server, we created the following two well-encapsulated services:

- **DockerService**:
  The DockerService is responsible for all the low-level, docker-specific functionalities. Many of the requests mentioned in table 3.1 trigger quite complex actions. For example, when a user requests to deploy an application, the server generally handles the following steps: (1) Creating an overlay network on the hosts, (2) Creating all the necessary cells in this network, and (3) Creating a reverse proxy cell for each organ that has an exposed port.

  To avoid such low-level logic in the API controllers, it makes sense to encapsulate this in a service. The DockerService handles the connection to the Docker infrastructure by the mentioned npm package *dockerode*. It takes care of everything that is concerned with Docker, consisting of the following categories:

  – Initialization: Establishing the Docker infrastructure connection, extracting the organs from the initial docker-compose.yml file

  – Deployment: Handle the overlay network, deployment (creating the containers and the reverse proxies as cells)

  – Adaptations: Moving containers, adding/removing containers, clearing all containers of a certain application

  – Infrastructure: Get host information and parse it in order to save it in our database

- **PrometheusService**:
  The PrometheusService is responsible for a clean provisioning of analytic data for the frontend. It establishes the connection to the Prometheus API and allows the fetching of any data the user requests. Since this data is cell-level data, it makes sense to have aggregation functions in place. In the current version of this service, only *average* is implemented, but it's programmed in a pipeline-like way so various alterations and aggregations of the data can be added and chained to the service call.

3.1.4 Tests

It is crucial for the whole platform that the backend's interface works as expected. To assure this, we wrote tests of all the API endpoints in different scenarios, based on the Node.js testing framework *Mocha*[3]. Since most requests trigger functions of the services, these tests cover the majority of code in the backend. How to run the tests is described in the README.md[4] of the backend component.

[3]https://mochajs.org/
[4]https://github.com/glapp/gla-sails

Table 3.1: Endpoints of the REST API

Endpoint	**Description**
User	
POST /user/signup	Used by the frontend to sign up a user.
PUT /user/login	Used by the frontend to log in a user.
GET /user/logout	Used by the frontend to log out a user.
GET /user/confirm-login	Helper endpoint needed for the frontend to assure a user stays logged in on a refresh of the page.
Application	
GET /application/getUserApps	Used by the frontend to get high-level information about applications of a specific user.
GET /application/details	Used by the frontend to get detailed information of a specific application.
POST /application/add	Used by the frontend to add an application.
POST /application/remove	Used by the frontend to remove an application, including clearing up its existing cells on the hosts.
POST /application/deploy	Endpoint used by the frontend to deploy an application on the infrastructure.
POST /application/undeploy	Used by the frontend to clear up an application's existing cells on the hosts.
POST /application/rename	Used by the frontend to rename an application.
Organ	
POST /organ/scaleUp	Used by the MAPE to scale up an organ. Constraints can be defined to place the new cell on a specific host, e.g. in the region "US".
POST /organ/scaleDown	Used by the MAPE to scale down an organ. A specific cell can be specified to control what cell is going to be destroyed.
Cell	
POST /cell/move	Used by the MAPE to move a specific cell from one host to another.
Host	
GET /host/info	Used by the MAPE and the frontend get information about the current infrastructure.
GET /host/prometheusUrl	Used by the MAPE get obtain the URL of Prometheus.
Rule	
GET /policy	Used by the MAPE and the frontend to get the list of rules for a specific application.
POST /policy/set	Used by the frontend to set a new or replace an existing rule.
POST /policy/remove	Used by the frontend to remove a rule.
Analytics	
GET /analytics/organCpu	Used by the frontend to retrieve organ-level data for the cpu usage graph.
GET /analytics/organMemory	Used by the frontend to retrieve organ-level data for the memory usage graph.
GET /analytics/events	Used by the frontend to retrieve application-level data for the events graph.

3.2 GUI

The user interface is primarily responsible for facilitating developers to create, deploy and manage their GLAs on the platform. It's responsible for showcasing all the critical data of the application in an intuitive manner. The following sections will explain in detail about the technology used under the hood, features of the platform and internals of the source code components.

3.2.1 Technology

As mentioned earlier, user interface is built using Angularjs, a popular framework to built efficient single page applications. This project used the material angular[5] library for the interaction design as a whole and to build responsive layouts that can work on most screen sizes. Bower[6] is used as a package manager for all the view-related libraries and NPM is used side by side for the necessary libraries. We integrated gulp, a stream building system to utilize the power of node streams and perform fast builds. Gulp[7] not only makes development easy with live reloads of the changes in source code, but also offers an efficient pipeline to integrate all the controllers, assets, services, etc., into a production-ready build. The choice of bower is due to its ability to resolve dependencies between packages efficiently. It is paramount to not maintain multiple copies of the same library, which can bog down the application in due course. Bower can prevent such problems by resolving the packages in a flat manner. Major libraries we used in this component angular-material, angular-ui-router, angular-messages, angular-sails, d3, nvd3[8], angular-visjs[9], toastr and many more minor libraries. Each library was chosen based on the level of adoption, revision history and the track record of open issues at the time. Wrong choice of libraries can often lead to additional development cost and break down the code which is undesirable. We also used a set of libraries with npm package manager including, lodash, eslint-angular, gulp-uglify, gulp-minify etc., to achieve our objectives. AngularJS offers dynamic synchronization between models and views through its two-way data binding capabilities. This is particularly important to present dynamic information and smooth user interactions. Decoupling DOM manipulation from application logic is a great boon for development, as we could focus on writing the important logic and not worry about the rendering. Using dependency injection, we could integrate many services and functions with the views easily. Also two way binding relieves the server from template related responsibilities, as these are rendered in html according to the details specified in the model. As soon as there is a change in data model, Angular's scope service will trigger a watch-digest cycle, by which it passes the new changes to the view via controller. The simple routing service allows us to structure the application well, and be able to navigate between the states easily.

3.2.2 Usage of the platform

The interface is organized into two main views at the higher level. One is the apps view and the other is the dashboard view. Users can enter the platform with valid login credentials, else they can create a new account at the time of landing. On entering the platform, they can access the apps page, which is an aggregation of all GLAs created in the platform. Each widget of a GLA on the apps page provides additional information related to it, such as the number of organs that are part of the GLA, recent events, current deployment status etc. Users can also access the current infrastructure details from the hosts tab in the same view. One can see the details of the swarm cluster, swarm agents, including the details of cloud provider, region where the agent is hosted. Hosts page also showcases the information regarding tier, operating system details of the agents and other hardware specs like memory etc. This is rather an administrator view, which may not be accessible by a regular user in an end product.

[5]http://material.angularjs.org
[6]https://bower.io/
[7]http://gulpjs.com/
[8]http://krispo.github.io/angular-nvd3
[9]http://visjs.org

Creating a new GLA is made to be quite simple with GLAPP platform. Users can create a new GLA application by providing the git-repository URL of their application code. One constraint is that this repository should contain a docker compose file. With this basic information they can add a new GLA to their apps collection. The client then forwards these details to the backend which then handles the extraction of cells and organs with the help of the cloning and container management module. Users can see the status of readiness of their GLAs in the apps page. Once the organs of GLA are extracted and prepared for deployment, it is moved to the *ready* state. The user interface differentiates these states with color coding for easy reference. If the GLA is in ready state, users can proceed to the deployment through the dashboard.

The dashboard aims to provide all the application information, metric data and deployment related settings at one place. It is specific to each GLA and can be accessed by simply clicking on the GLA in the apps view. The dashboard is organized into five sub-views named overview, policy, analytics, history and settings.

As shown in the figure 3.2, overview renders the current visualization of GLA with it organs and cells in a network graph. The frontend uses the visjs[10] library with few modifications for this purpose. The large blocks are known as organs and the smaller green blocks are the cells that are part of each organ, while the red blocks are the proxy cells. Proxy cells are also like gateway cells, as they are responsible for carrying the information from and to the other cells in the organ.

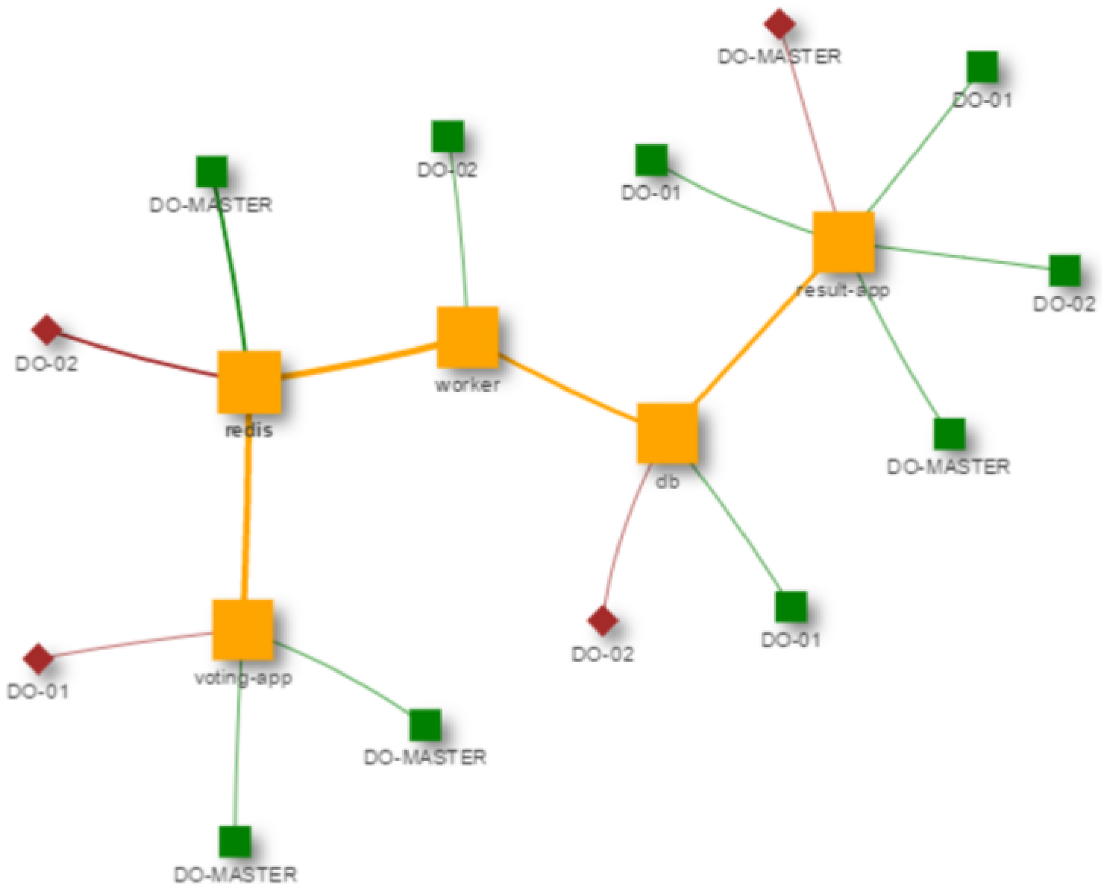

Figure 3.2: Visualization of a GLA in the frontend

Users can switch between list and graph views of the GLA. The current endpoints of the application are also updated from time to time in the same view, so users can check out the live

[10]http://visjs.org/

status of the application. The list view contains a detailed view of each cell, showcasing low-level information including various environment variables and the host information. This view is also intended to serve as a manual mode for users, if they intend to move or scale certain cells of their application. All the capabilities of the platform are offered here in the manual mode.

The policy view in dashboard allows users to create and modify one or more policies at the same time. Each policy can accept a weight and a goal. The goal is a particular metric chosen to be applied on a set of organs, with certain constraints through operators such as *less than*, *greater than* etc. The policy view is responsible for the policy management in the platform. The MAPE module takes these policies into account as soon as they are created or modified.

The analytics page provides a real-time feed of important metrics related to the application such as CPU usage, memory usage and events. The analytics are rendered using nvd3[11] visualization library. In both CPU and memory usage charts, the users can check the data for any specific organ and varying time frames. Data is pulled by calling backend API, which then queries the Prometheus server for the actual data. All charts are updated in real-time. The events chart is particularly interesting, as it shows a real time view of deployment adaptations happening on the GLA.

The history page shows a detailed log of all the application-related events that happened.

The settings view provides options to deploy or undeploy the GLA application. Users can also delete the whole GLA, which proceeds to not only clean up the GLA in database, but also to clean up any traces of the related cells deployed in the infrastructure.

3.2.3 Internals

The source code is organized to be modular and easy to iterate over development cycles. Code related to the client can be accessed through the gla-angular[12] repository. The major part of the code is located in *src* folder which is further organized in apps, components, dashboard, home, infrastructure and landing. Each of these folders and sub-folders is responsible for one complete set of MVC and contains a controller file and the view file inside them. Pathways are organized by the router service. Various build scripts for production, test and deployment are organized in the gulp folder. This client component itself is dockerized as part of larger GLAPP platform, and these settings can be modified in the Dockerfile. Barring the other commonalities, we believe this structure helped us a lot with development and also make future updates easier.

3.3 MAPE

The MAPE is the control loop of the platform and responsible for managing the deployed applications in accordance to the policy. It analyzes the environment information and performance metrics to determine the healthiness of the deployed application and triggers appropriate adaptation action in case of any violation to the user defined policies. The environment information includes details of the infrastructure, application deployment and user-defined policies.

MAPE consists of three parts: one interfaces with other systems to retrieve the necessary information for analysis and dispatch the adaptation action, another part analyzes the application status and determines its healthiness with regard to the policy, and the last part determines appropriate adaptation actions based on the environment information and the computed healthiness.

The MAPE implementation is grouped into several packages:

- Package `ch.uzh.glapp` contains the core classes of MAPE including the `MainLoop` class where the main function is defined. It also includes classes for interfacing with other systems as well as an utility class providing auxiliary functions used in various computations.

[11]http://nvd3.org/
[12]https://github.com/glapp/gla-angular

- Package `ch.uzh.glapp.mdp` contains the classes for MDP using the BURLAP library.
- Package `ch.uzh.glapp.model` and its sub-packages contain the classes for modeling the infrastructure information and performance metrics returned by the GLAPP backend server and the monitoring system.

3.3.1 Interface with other systems

At the start of each control loop, the MAPE interfaces with the sails backend to retrieve environment information, and with the monitoring system to retrieve performance metrics. To conclude the loop, the MAPE communicates again to the sails backend to execute the adaptation action and to get the new state information. As mentioned in the implementation decision section, MAPE uses the HTTP protocol with JSON objects for data exchange. It retrieves everything including infrastructure information, application deployment information, user-defined policies, and performance metrics through HTTP requests. For instance, the infrastructure information includes information about the hosts such as the service provider, instance location and size. Deployment information, on the other hand, refers to the information of each application components including on which hosts they are currently deployed. Furthermore, MAPE also retrieves user-defined policy information through the sails backend. A policy is a set of rules that defines which metrics the application need to comply to its threshold for calculating the healthiness and the weight of each rule relative to the others. Lastly, the MAPE retrieves the performance metrics from the monitoring system. JSON objects are used for all these information retrievals. The use of JSON objects provides a data structure that is flexible to represent different types of information from different interfacing systems. In addition, it allows the MAPE to build an object-oriented view of the whole environment that provides quick access to specific information for various computations.

3.3.2 User-Defined Policy

The user-defined policy is a key element for the MAPE. It consists of a set of user-defined rules that governs the application deployment. A rule is specified with the input in the following fields:

- The **metric** field defines the performance indicator that the current deployment will be measured against.
- The **organ** field defines one or more organs to which the rule applies.
- The **threshold** field defines the upper or lower bound that the metric value should be compliant to.
- The **operator** field defines how the metric value should be compared to the threshold. (e.g. *less than*, *bigger than*) For example, a *less than* operator means the metric value needs to be less than the specified threshold.
- The **weight** field defines the weight of a rule relative to others when multiple rules are specified.

Based on the metric selected, a rule can be classified into one of the following three types:

- A **resource utilization rule** limits the resource utilization to the specified threshold. An example would be to restrict the CPU utilization to be lower than 80%.
- A **cost rule** limits the total cost incurred for running the cloud infrastructure.
- An **application metric rule** specifies the range of application metric that the application need to comply to. In current implementation, the click rule is an example of application metric rule, which will be introduced in chapter 4.

3.3.3 Healthiness Computation

Based on the rules defined in the policy, a healthiness value is computed. For the i-th rule, the healthiness of a cell is defined as the ratio of the difference between metric value and threshold to the threshold normalized with the weight of the rule:

$$\text{healthiness}_{cell} = \frac{\text{difference between metric value and threshold}}{\text{threshold}} \times \frac{w_i}{\sum_{i=1}^{n} w_i}$$

where w_i is the weight of the i-th rule and n is the total number of rules.

A positive healthiness value means the metric value is compliant and a negative healthiness value means the metric value is non-compliant. For example, assuming there is only one rule, if the rule specifies the *CPU utilization* need to be *less than* a threshold of 0.8 (80%) and the current CPU utilization is 0.6(60%), the healthiness value will be:

$$\begin{aligned}\text{healthiness} &= \frac{0.8 - 0.6}{0.8} \times \frac{1}{1} \\ &= 0.25\end{aligned}$$

The healthiness value of a rule is the aggregated value of all cells' healthiness values.

After the healthiness values of the each individual cell is computed based on the policy and performance metric, all violating cells within an organ are taken into account for an evaluation. When the number of violating cell exceed a defined threshold, the rule will be considered as violated. Information of each violating cell along with the healthiness value will added to a list of violations. This list of violations will be further processed in MAPE to determine an appropriate adaption action.

3.3.4 Markov Decision Process

A Markov Decision Process (MDP) will be triggered to find an application deployment that fulfills the policy requirement whenever a rule violation is detected. In the first implementation of MAPE, MDP as a reinforcement learning process was used to solve the problem of determining adaptation action in response to the rule violation. The output of solving this problem is an adaptation action or a series of adaptation actions that transit the application from a state where policy is violated to another where the policy is fulfilled. Before starting the MDP, the environment and application deployment need to be modelled as states. At the same time, adaptation actions are modelled as transition actions in MDP that change state from one to another. BURLAP library and its framework is used for modelling and transition action definition. Classes in the package `ch.uzh.glapp.mdp` are implementation according to the BURLAP framework for learning agent[13].

- `MapeWorld` class defines the domain of optimization problem.
- `MapeCell` class defines the state representation for MDP.
- `MapeActionType<action type>` classes define the transition actions. These classes perform a pre-condition check on the current state and generate applicable actions for that state.
- `MapeAction<action type>` classes define the actual action that can be selected by the learning algorithm as output of MDP.
- `MapeEnvironment` class defines the functions to observe the real world environment and to execute the action output from MDP.
- `BasicBehaviorMape` class sets up the whole environment for MDP and specifies the learning algorithm used.

[13]http://burlap.cs.brown.edu/tutorials/bpl/p4.html#qlearn

When the state and transition action objects are constructed, the learning algorithm from the library can be applied.

Q-Learning algorithm was first applied in a simulated environment. A simulated environment was used to verify correctness of the constructed model and the defined transition action. It also served a second purpose for understanding MDP process by generating detailed log of MDP execution. This log provides a sequence of transition actions and Q-value corresponding to each transition that allows fine tuning of Q-learning parameters in the later stage of development.

While the outcome of MDP in a simulated environment is close to expectation, concern was raised regarding the convergence of Q-value that learning algorithm relies on to determine the optimal action in a given state. More importantly, the problem of large states spaces affects the viability of MDP. The model and transition actions were further adjusted in subsequent iterations. After several testings of MDP in the simulated environment, we concluded that a reduction of number of states is required for applying MDP. To reduce the number of states, function approximation is needed. Function approximation methods including both on-policy SARSA and off-policy Q-Learning with gradient descent were explored. The gradient descent method requires numeric vectors for representing features in states. For modeling the GLA, attribute of both the organs and the cells are used as the features. More precisely, these attributes are the number of cells in an organ, the cloud service provider, region and hardware specification of the hosts that cells reside. These features consist of nominal values and as a result both SARSA and Q-Learning methods cannot be used. However, since the provided function approximation methods from current BURLAP library are not applicable to nominal feature, further exploration of function approximation and implementation are needed. The possibility of using MDP with alternative function approximation method will be discussed in the chapter 5 Future Work.

3.3.5 Rule-based Adaptation

For the final implementation, rule-based adaptation is used in MAPE for the determination of adaption action. Although MDP is not used in the final implementation, the BURLAP framework is retained. As a result, changing to rule-based adaptation involves mainly the implementation of a rule set to replace the learning algorithm while the majority of previous implementation for other functionalities are carried over. The implementation for state representation and environment interaction is reused. For the rule-based adaptation, the rule set is implemented in the `MapeActionTypeHeuristic` class. The `MapeWorld` class, which defines the problem domain, is also updated to use this `MapeActionTypeHeuristic` class.

With the use of a rule set, MAPE first analyzes the violation information from the previous stage of healthiness computation. If there are multiple violations, the list of violations are sorted in descending order of the healthiness value. Then the MAPE will act on the violation with the most negative healthiness value. This ensure that the violation that has the largest impact on the overall application healthiness is addressed first. The violations are classified into three types: resource utilization violation, cost violation, and custom metrics violation.

Resource utilization violation is handled by moving the violating cell to a host with better specification (i.e. a host in a higher tier). In case a higher tier host is not available - if e.g. the cell is already running on a highest tier host - duplicating/splitting cell will take place.

Cost violation is a specific type of custom metric where the adaptation action is to scale down the deployment. Whenever there is a cost violation, details of all cells belonging to the organ's specified policy are taken into account. Then the rule set will command MAPE to find a host with the cheapest cost and trigger an adaptation action to move a cell to that host.

The click is the custom metric that we implemented. It is handled differently from both resource utilization violation and cost violation as it scales up/out or down based on the magnitude of the healthiness value. Since click metric represents the amount of click from the application frontend in a specific region, a small amount of click indicates the application and its corresponding cell is under-utilized while a large amount of click indicates the application and its responding cell is over-utilized. In this regard, the healthiness value associated with click violation is the number of clicks to number of serving cells ratio. When the ratio for a given region is lower than scale-down threshold, an adaptation action will be triggered to reduce the number of cells in that region. On the contrary, when the ratio is higher than the scale-up threshold, an adaptation action will be triggered to duplicate/split cell in that region.

3.3.6 Extensibility

Extensibility of control logic is built into MAPE with use of BURLAP framework. The framework enables swapping or enhancing of various parts of the MDP, including planning and learning algorithms and value approximation functions, by implementing the corresponding interface or extending existing classes. The quick switch from MDP to rule-based adaptation in final implementation is achieved with this inherited extensibility. To replace the learning algorithm provided by the BURLAP library, one can create a class that implements `burlap.behavior.singleagent.learning.LearningAgent` and a class that implements `burlap.behavior.valuefunction.QProvider` interface. The `LearningAgent` class defines the learning agent's behavior in an environment. The `burlap.behavior.valuefunction.QProvider` is responsible for computing the Q-value.

3.4 Other

3.4.1 Monitoring / Prometheus

Prometheus[14] is a monitoring system that can provide both infrastructure and application performance metrics. Infrastructure metrics are provided by exporters, which is a set of library that expose metrics data of the host environments and the Docker containers. Application metrics are provided with the use of custom-built program to expose the needed data in a Prometheus exposition format[15]. The use of Prometheus and its exposition format enables MAPE to access custom metrics such as application performance data or cost metrics of the host from various cloud service providers.

Prometheus is sophisticated with the capability to support different types of query. It provides range queries and instant queries (queries at a single point in time [Aut16]) with support of applying functions on data when formulating a query. This enables the fine-grained retrieval of data that best fits the corresponding performance metric. For instance, the *rate* function allows the direct retrieval of the rate of change in CPU time of a container with configurable data point range, data point interval and duration covered by each data point.

The robust query functionality provides two major advantages. First of all, it shifts part of the computation from MAPE to Prometheus and thus reducing complexity of MAPE in getting specific performance metric for subsequent healthiness computation. Secondly, it allows quick tweaking and testing of query parameters to obtain the most relevant data for the determination of an adaptation action during later development.

3.4.2 Supporting Tools & Applications

In the course of the project, several further small tools and applications were used to complete the task. This section quickly mentions them with a reference to further information sources:

- **Infrastructure scripts**:
  To always work on the same host infrastructure, we soon created handy scripts to set up a complete infrastructure which includes all of the required components. This is all collected in a git repository[16] and can also be used by other people to set up the same infrastructure as we were working with.

[14] https://prometheus.io/
[15] https://prometheus.io/docs/instrumenting/exposition_formats/
[16] https://github.com/glapp/docker-swarm-creation

- **Consul.io**:
  As a discovery service for the Docker swarm hosts, we used the key/value store of Consul.io[17]. It is included in the mentioned infrastructure scripts.

- **Registrator**:
  To register every exposed port of the deployed applications, we used Consul.io in combination with Registrator[18]. In order to register not only public ports but also exposed ports that are only available in the created overlay network, the *-internal* run option of Registrator was used.

- **HAProxy**:
  Registrator was used to create the mentioned reverse proxies. In order to always have an up-to-date reverse proxy with a complete list of active endpoints of a specific organ, we created a HAProxy docker image[19] that used Registrator and consul-template[20] to dynamically adjust the configuration of the HAProxy and restart it with every change.

- **Voting-App**:
  The voting-app is our demo application that will be introduced in more detail in the chapter 4. It is based on an example application by Docker itself[21]. We adjusted the individual components to our needs, e.g. to include custom metrics, and they can all be found on our github page.

- **Metrics-Server**:
  The metrics-server[22] is a small service we created that was used to register the clicks of the demo application. Later on we used it to serve the cost metrics as well. It can be extended to handle all different kinds of metrics that can be scraped by the Prometheus server.

[17]https://www.consul.io/
[18]http://gliderlabs.com/registrator/latest/
[19]https://github.com/Clabfabs/miniboxes/tree/master/minihaproxy, https://hub.docker.com/r/clabs/haproxylb/
[20]https://github.com/hashicorp/consul-template
[21]https://blog.docker.com/2015/11/docker-toolbox-compose/
[22]https://github.com/glapp/metrics-server

Chapter 4

Case Study

4.1 The Demo Application

For development and presentation purposes we used an adapted version of the example-voting-app[1] that Docker used to illustrate the Docker Compose tool. It is basically a web application that lets users vote for either option A or option B. The version of the voting application that we used consists of five components:

- voting-app (Python)
- result-app (Node.js)
- redis
- PostgreSQL
- worker (Java)

A user votes A or B on the voting-app and the vote is sent to redis. The worker takes the vote from the redis in-memory database and stores it into the PostgreSQL database. As the name suggests, the role of the result-app is for displaying the results of the votes. Figure 4.1 is a high level overview of these components.

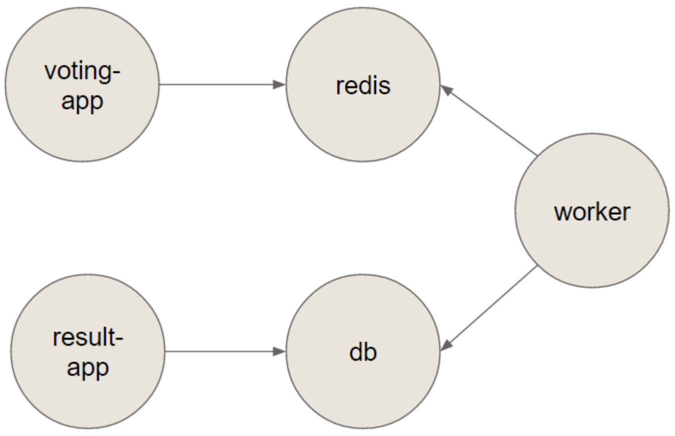

Figure 4.1: The demo application components

The voting-app is a Python-Flask web application that lets the user vote either for Batman or for Superman.For the end of project presentation we also added a button to switch between

[1]https://github.com/docker/example-voting-app

US and EU in order to be able to simulate clicks done in the two regions by registering each click in the the mentioned metrics-server. This was necessary in order to show the ability of the GLAPP platform to adapt the demo application based on the user location. More about the click metric can be found in section 3.3.5.

The result-app is a Node.js web application that retrieves the votes in the relational database PostgreSQL.

The worker is a Java application that inserts a vote into the PostgreSQL database whenever a new vote is available in the redis in-memory database.

The last two components are the standard redis and PostgreSQL databases. All five components are available as Docker containers and everything is wrapped up in the docker-compose file of our gla-demo-app[2].

4.2 Scenarios

For the final demonstration of GLAPP we showed three typical scenarios which the GLAPP platform is able to handle:

1. An Instance-to-Instance scenario, where we stressed the CPU
2. A Provider-to-Provider scenario, where we changed the cost of a cloud provider
3. A Region-to-Region scenario, where we simulated clicks on the voting-app from different regions

4.2.1 Instance-to-Instance

In the first scenario, we used the Linux terminal to enter one voting-app cell in order to use the stress tool[3] to simulate a high CPU utilization. We also set up a policy that lets the MAPE monitor all cells of the voting-app organ. When stressing the cell, the MAPE will take care of it by transferring the cell to another host with more CPU resources assuming that it will alleviate the CPU load.

4.2.2 Provider-to-Provider

For second scenario we had a policy that monitored the total cost of the whole application in terms of USD per hour. The MAPE calculates the current cost and whenever the cost exceeds the policy, it triggers an adaptation. In this case we raised the cost per hour of Amazon Web Services (AWS) instances. To do this we implemented a RESTful API in the metrics-server described in section 3.4.2 that enables us to switch between two cost profiles for AWS. These two profiles are both scraped by Prometheus. As soon as the MAPE detects the higher costs of the AWS machines, it successively migrates affected cells to a cheaper cloud provider until the total cost satisfies the policy again.

4.2.3 Region-to-Region

In the last scenario we showed that the platform is able to follow the users. It moves or creates new cells where the voting-app users are geographically located. We utilized the custom click metrics, as described in section 3.3.5. After ensuring that the click policy is in place we started to click on the voting-app buttons. The system registered the user clicks and spawned new cells in the corresponding region as soon as the threshold of clicks per minute has been hit. When we stopped clicking, the clicks per minute went down again and the whole organ was scaled down to one cell again.

[2]https://github.com/glapp/gla-demo-app
[3]http://linux.die.net/man/1/stress

Chapter 5

Future Work

In the process of developing GLAPP, we came across several possible extensions that could have been implemented but turned out to be out of scope of this master project. However, GLAPP could be extended in future student projects, so it's still worth mentioning those possible extensions.

5.1 Support Data Mobility

It requires a lot of work to ensure consistent and persistent data management if a cell with data is moved. In the current state, GLAPP ignores the data when moving a cell, so a database will be removed on one host and re-created on another host, which means all data is lost. This is fine for the scope of this project, but as a product it's obvious that GLAPP would have to support data mobility. That is not trivial however, since it would require to e.g. copy a whole database from one host to another host while ensuring that no data is lost in the meantime. It also drastically slows down the movement process.

This is a common problem and therefore there exist libraries and possible solutions for this. Docker itself tries to provide a native solution for it by allowing to turn the data of a container into a tar-file before moving it[1]. However, this doesn't solve most of the mentioned problems. E.g., what if the database should be further filled during the movement process?

An example of a library that presents a possible solution is Flocker[2], which introduces Flocker data volumes that are supposed to be portable and following a container when it's moved.

Another idea would be to just let the data where it is and just create a new data store somewhere else that is somehow "connected" to the first one. This goes to the direction of distributed databases.

5.2 Dynamic Infrastructure

Currently, GLAPP relies on a fixed infrastructure of hosts. We ensured that it's as easy as possible to set it up by providing scripts, but once the infrastructure is in place, it's not supposed to change. However, it's desirable to be flexible when it comes to available hosts in the GLAPP infrastructure. In theory, in a fixed infrastructure, there should be one host for every possible combination of deployment attributes like region, provider, tier, etc. to ensure that a cell can move freely between those attributes.

In a dynamic infrastructure, the hosts would be spawned and killed on the fly, depending on the required resources. So, if a cell should be moved to a DigitalOcean instance in Europe and there is no such host yet, the backend would spawn such a host and add it to the network of the existing hosts. Technically, this is not a big challenge, since most providers offer an API

[1]https://docs.docker.com/engine/tutorials/dockervolumes/#/backup-restore-or-migrate-data-volumes
[2]https://github.com/ClusterHQ/flocker

to automatically spawn and kill instances. Moreover, to join a Docker swarm once an instance is spawned is simple. It would still be a trade-off between costs and availability/speed - if we keep some hosts with a common combination of attributes like a US instance on DigitalOcean with tier 1 alive constantly, movements to this host would be instant.

But the main problem here and the reason that this turned out to be out of scope of this project is the fact that these hosts need to be managed by a separate, higher-level control loop as well. This control loop would have a meta-perspective on the infrastructure, would monitor the metrics of the cloud instances and trigger adaptions in the form of the creation or destruction of instances with certain attributes.

5.3 MDP for Learning

Extensibility is also an important aspect of the MAPE. As discussed in section 3.3.6, the MAPE allows custom learning algorithm to be used instead of the actual implemented rule-based adaptations. On the other hand, while the learning algorithms with standard function approximation in the BURLAP library does not handle nominal feature in states, they can be replaced with other function approximation implementation. One possible method to explore is Free-Energy based Reinforcement Learning (FERL). With FERL, the action-value function, Q, is approximated as the negative free-energy of a restricted Boltzmann machine [SH04]. In a restricted Boltzmann machine, binary random variables are represented as nodes in a bipartite graph [SH04]. The nodes belong to two disjointed sets: visible state and action nodes and hidden nodes with weight on every edge connecting a visible node and a hidden node [SH04, EUD13]. Free energy can be computed as an expected energy, which is determined by the weights, minus an entropy [SH04]. The negative free energy is used to approximate the Q-value ($Q_t = -F_t$) [SH04, EUD13]. Further studies in free-energy based reinforcement learning can be done to determine the viability of using it as an approximation function in MDP to address the large state space problem.

5.4 Decentralized MAPE

Another direction of extension is decentralizing MAPE. Currently MAPE is run in a centralized way such that an instance of MAPE retrieves information of all GLAs deployed and analyzes each of them to determine if any adaptation action is needed. Moving to a decentralized MAPE, one can change MAPE to work on only one GLA and instantiate a separate MAPE instance for every GLA deployed. Since the GLAPP is containerized, this means instantiating a decentralized MAPE instance can be done easily by starting container for its corresponding GLA. GLAs with a decentralized MAPE lead to autonomous GLAs in the sense that it is now equipped with a capability to manage itself. However, further consideration on coordination between multiple MAPE instances is needed, when decentralized MAPE instances also need to manage underlying cloud computing resources.

Chapter 6

Conclusion

Staying true to our initial goal, this project successfully introduced the notion of global living applications (GLA) and a new cloud middleware platform to manage these GLAs. These bio-inspired notion of GLAs are constructed with several organs and cells. We were able to abstract different components in a cloud application to the notion of cells in GLAs. Apart from the macro level design of GLAs, we also introduced several microscopic phenomenons at the cellular level. Cells in GLA can replicate to create two identical copies of themselves, akin to mitosis. Similarly, we showed through on the of deployment scenarios, that cells can move around between cloud providers and all other kinds of environments like region, tier, etc. We also introduced a MAPE control loop into our deployment platform, with capabilities of continuous optimization of GLAs runtime environment. Our platform is designed in a way for users to define certain high level policies to guide their GLAs easily. This policy language is quite simplistic and is extensible with third party application metrics. Our GLAPP platform is designed from the ground up to take advantage of arbitrage opportunities that exist between cloud providers in terms of cost, performance, availability, etc. To this extent we not only succeeded in building such a platform, but we also provided through a series of scenarios and sample GLAs that we can take advantage of these opportunities and adapt the deployment plan from time to time.

The platform is designed in a modular and extensible way, so that the developers can plug in external application metrics to feed the control loop easily. It is also possible to switch the learning module to apply different optimization strategies. With a simple specification of a compose file and details of the code repository, GLAPP is capable of building the images and deploy the GLA application instantaneously. Another successful result is how fast GLAPP can respond to adaptations. It can move to another cloud provider or create new cells in real-time without any downtime for the applications. It is also important to mention about the introduction of red cells, which are responsible for the gateway functionality. They act as proxies for the organ and make sure application endpoints are always reachable. We had built a infrastructure cluster that takes advantage of latest technologies, to push the overall resource utilization and performance to the next level. So overall, we believe the notion of GLAs as living, breathing applications are quite promising and a great first step towards building truly autonomous self-managing cloud applications. GLAPP as a platform succeeded in meeting its objectives and also delivered an extensible platform. In the end, GLAPP goes many steps beyond a conventional cloud middleware that deploys and performs some sort of linear scaling. GLAPP is capable of multi-cloud deployment and offers a multitude of new opportunities in terms of availability and resource utilization to its users. With GLAPP and the introduction of the MAPE control loop, we are able to constantly monitor the overall state of a complex application spread across the world, and to optimize the deployment plan in real-time. This marks a new beginning in the domain of cloud deployments, and we hope further inspire others to build on the approach.

Bibliography

[Aut16] Prometheus Authors. *HTTP API | Prometheus*, 2016 (accessed August 31, 2016). `https://prometheus.io/docs/querying/api/`.

[EUD13] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Scaled free-energy based reinforcement learning for robust and efficient learning in high-dimensional state spaces. *Frontiers in Neurorobotics*, 7:3, 2013.

[Ope16] OpenVZ. *OpenVZ Virtuozzo Containers Wiki*, 2016 (accessed September 1, 2016). `https://openvz.org/Main_Page`.

[Pah15] Claus Pahl. Containerisation and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[SH04] Brian Sallans and Geoffrey E Hinton. Reinforcement learning with factored states and actions. *Journal of Machine Learning Research*, 5(Aug):1063–1088, 2004.