POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Fine-Grained Adaptation of Cloud Applications with Containers

DEIB
Dipartimento di Elettronica,
Informatica e Bioingegneria del Politecnico di Milano

Relatore: Prof. Luciano Baresi
Correlatore: Dott. Giovanni Quattrocchi

Tesina di Laurea di:
Dmitrii Stebliuk, matricola 823716

Academic Year 2015-2016

Dedicated to my family and everybody who supports me through thesis and master degree period

Abstract

Nowadays Modern Web applications are often deployed and executed on a Cloud infrastructure which provides a convenient on-demand approach for renting resources and easy-to-use horizontal scaling capabilities. The workload of Web applications is continuously changing over time and unexpected peaks of requests can happen, making the system unable to respond. For this reason the autonomic adaptation is an emerging solution to automatically adapt the resources allocated to the application according to the incoming traffic, cpu-utilization, and other metrics. Our ongoing autonomic initiative is based on the MAPE architecture (Monitor-Analyze-Plan-Execute). This thesis focuses on the Execute component.

While the state of the art solutions focus on adjusting the number of Virtual Machines allocated to the application, the containerization, a novel kind of virtualization that takes place at the operating system level, is emerging and is becoming popular. Containers are linux processes that can run sandboxed on a shared host operating system. This means that each container does not contain an entire operating system making this technology more lightweight and faster to boot comparing to Virtual Machines.

The contribution of this thesis is the implementation of the Execute component that exploits the usage of both Virtual Machines and containers enabling a faster and finer-grained adaptation and multi-layer adaptation. We consider not only the adaptation at the infrastructure layer, but we also adapt the middleware software that enables the execution of application specific code as application servers, DBMS and so on. We have implemented two approaches for the "Execute" component: the monolithic and the hierarchical one. The former consists of a centralized architecture where only monitoring sensors are distributed among the nodes, the latter consists in completely distributed architecture where all the MAPE components are replicated at each level of the hierarchy (container, VM, cluster of VMs, etc.).

Acknowlegements

I would like to thank Politecnico Di Milano professors of the courses I took for the dedication to their work and interesting, cutting-edge material they were teaching us.

Also I would like to thank my supervisor Giovanni Quattrocchi for guiding and supporting me all other my thesis production period.

Contents

Chapter 1

Introduction

Nowadays software can consist of many different parts that are running together. The functional and non functional requirements are subject to continuous changing and the service infrastructure should be capable to support these changes. A concrete example of automatic infrastructure management is Amazon's Auto Scaling, which manage when and how an application's resources should be dynamically increased or decreased. This paper describes implementation of alternative solution that help to solve autoscaling problems.

1.1 General idea

This paper describes Autonomic Systems management using different types of virtualization: virtual machines and containers. This Autonomic System adaptation is based on the MAPE framework: "M" stands for monitoring, "A" for analysing, "P" for planning and "E" for execution. In other words the system is monitored, analysed and the adaptation plan is produced that is executed by some driver depending on what virtualization technique we choose.

1.2 Autonomic system

Autonomic system is self-adapting self-managing system with distributed resources, that can adapt to unpredictable changes while hiding intrinsic complexity to operators and users. IBM was on of the first companies who suggested this kind of systems and it has set forth eight conditions that define an autonomic system: The system must

1. know itself in terms of what resources it has access to, what its capabilities and limitations are and how and why it is connected to other systems.
2. be able to automatically configure and reconfigure itself depending on the changing computing environment.
3. be able to optimize its performance to ensure the most efficient computing process.
4. be able to work around encountered problems by either repairing itself or routing functions away from the trouble.
5. detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.
6. The system must be able to adapt to its environment as it changes, interacting with neighboring systems and establishing communication protocols.
7. rely on open standards and cannot exist in a proprietary environment.
8. anticipate the demand on its resources while keeping transparent to users.

Even though the purpose and thus the behaviour of autonomic systems vary from system to system, every autonomic system should be able to exhibit a minimum set of properties to achieve its purpose:

1. Automatic: This essentially means being able to self-control its internal functions and operations. As such, an autonomic system must be self-contained and able to start-up and operate without any manual intervention or external help. Again, the knowledge required to bootstrap the system (Know-how) must be inherent to the system.
2. Adaptive: An autonomic system must be able to change its operation (i.e., its configuration, state and functions). This will allow the system to cope with temporal and spatial changes in its operational context either long term (environment customisation/optimisation) or short term (exceptional conditions such as malicious attacks, faults, etc.).
3. Aware: An autonomic system must be able to monitor (sense) its operational context as well as its internal state in order to be able to assess if its current operation serves its purpose. Awareness will control adaptation of its operational behaviour in response to context or state changes.

1.3 MAPE The IBM Autonomic Framework

The IBM Autonomic Computing Initiative codified an external, feedback control approach in its Autonomic Monitor-Analyze-Plan-Execute (MAPE) Model. Figure 1 illustrates the MAPE loop, which distinguishes between the autonomic manager (embodied in the large rounded rectangle) and the managed element, which is either an entire system or a component within a larger system. The MAP loop highlights four essential apsects of self-adaptation:

1. **Monitor**: The monitoring phase is concerned with extracting information - properties or states - out of the managed element. Mechanisms range from source-code instrumentation to non-intrusive communication interception.

2. **Analyze**: is concerned with determining if something has one away in the system, usually because a system property exhibits a value outside of expected bounds, or has a degrading trend.

3. **Plan**: is concerned with determining a course of action to adapt the managed element once a problem is detected.

4. **Execute**: is concerned with carrying out a chosen course of action and effecting the changes in the system.

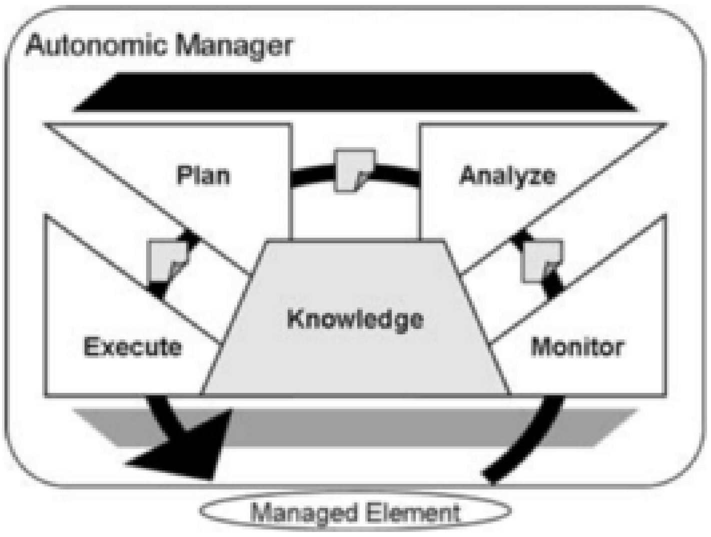


Figure 1.1: The IBM Autonomic MAPE Reference model

1.4 Cloud and coarse-grained virtualization

Modern web applications more often are run in the cloud. Running in the cloud means that application is deployed not on real physical (bare-metal) machine, but on the virtual machine where virtual infrastructure can be configured on the cluster of machines. This approach has advantages to classic bare-metal one. First of all it is scalability: as machine is virtual we can do vertical scaling (allocate CPU cores or RAM) as we want, moreover if the cloud infrastructure works on the cluster, we can provide resources that one real machine just do not have. The second advantage is again scaling: we can easily create new virtual machines: this is called horizontal scaling. And the last advantage is on-demand computing or pay-as-you-go, when you do not need to pay upfront, but only for the resources you use.

Current implementation uses Amazon Elastic Cloud Computing as cloud provider. Also it is supported Vagrant. Vagrant is software that is higher-level wrapper around virtualization software such as VirtualBox, VMWare, KVM, Linux containers and Amazon EC2. We do not use Vagrant and Amazon EC2 drivers for vertical scaling, but only for the horizontal one: creating/deleting new virtual machines. This scaling is considered as coarse-grained, because Amazone EC2 VM and VirtualBox VM takes quite long time to create and use heavy virtualization technologies: like full-virtualization, paravirtualization or hardware-assisted virtualization.

1.5 Containers

Container virtualization is operating-system-level virtualization method where the kernel of an operating systems allows for multiple isolated user-space instances, instead of just one.

On Unix-like operating systems, one can see this technology as an advanced implementation of the standard chroot mechanism. In addition to isolation mechanisms, the kernel often provides resource-management features to limit the impact of one container's activities on other containers. In this work we use docker implementation of container virtualization. It uses modern Linux features like LXC containers and cgroups for managing resources.

Containers have some advantages comparing to classic full-virtualization: they have very low or zero overhead, because they are running without emulation and just send system signals to operation system kernel. Moreover containers are much faster to create/delete as they do not need to start / stop operation system that can take significant amount of time. On the other hand containers are not considered as classic virtualization killers, that will take its place. Current approach is to use classic virtualization for clouds, and run container above classic virtual machines, so the containers virtualization is used above the classic one.

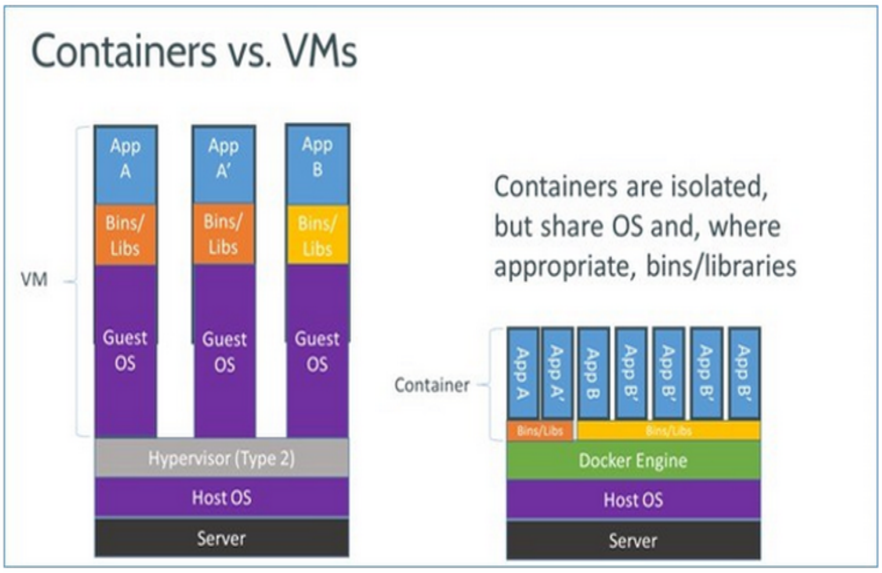


Figure 1.2: Comparing VM to container

1.6 Contribution

The main contribution was implementing the "E" executor part of MAPE framework applied to autonomous system. Different implementation cases were researched with different virtualization techniques and implementations. First of all executor accepts as the input the plan from the "P" planner. It was considered two different implementations of executor: "monolithic" and "hierarchical". The difference between them is that the monolithic executor manages virtual machines on its own, it decides when it is required to create/delete virtual machine, the input plan goes to the main node, which orchestrates agent nodes (virtual machines) with docker containers. The hierarchical executor manages only containers on the virtual machines and the plan is built for each VM, so the executor does not need to estimate the number of virtual machines or try to solve allocation problem.

Also the executor considers different types of virtualization: coarse-grained one (full-virtualization, hardware-assisted or paravirtualization) and the operation system-level virtualization (containers). Drivers for vagrant and amazon elastic cloud were implemented for coarse-grained virtualization. The driver for docker was implemented to support the light-weight containers virtualization.

Also some configuration adjustment should be implemented for continuously changing environment. For example after scaling JBOSS application from using 1 CPU and 2gb RAM to 8 CPU and 16gb RAM, we want to run instead of 4 JBOSS threads, 32 threads. This requirement was implemented as "scale-hooks" which run on each docker container creation/update. Also sometimes configuration adjustment should be more complex, taking the same example with JBOSS resources update, load balancing node needs to change weights or to add new nodes to its configuration. So load balancing node should wait for finishing of create/update operations on each JBOSS node and after that trigger its adjustment. This type of adjustment we call "tier-hooks" and for them should be specified dependency, for example load balance node depends on JBOSS tier. After each change of JBOSS tier, the tier-hooks run for the nodes that depend on the changed tier.

1.7 Thesis structure

The rest of the paper is organized as follows. The second part describes the objective, functional, non functional requirements and shows use-case diagram. The section 3 presents the solution design, problems, approaches, decision made and the architecture (components). The implementation part discusses algorithms, distributed vs monolithic approaches, Vagrant driver, AWS driver, docker driver and shows class diagrams and sequence diagram. And the last section is the conclusion and future works.

Chapter 2

Requirements

2.1 Introduction

The following requirements describe the system called **Executor**. The main goal of the **Executor** is to provide required infrastructure and resources to the controlled micro-services. With the term micro-service we understand some 1-tier or multi-tier application. We consider only 2 types of resources: CPU cores and available RAM.

The required infrastructure and resources are provided by creating Virtual Machines and allocating containers on them. Each container is considered to store 1 tier of the application. If tier requires more resources than 1 VM has then this tier would be represented by more than 1 container on different VMs.

The input of the executor is the **topology** and the **plan**. The **topology** describes each application and its tiers: static information which can be changed only by sending "change topology" request. The **plan** says how many resources are required for each tier.

The plan can be of two different types: the **monolithic** and the **hierarchical** one. The **monolithic** plan says just how many resources it needs for each tier and the **Executor** considering current **allocation** tries to decide how many VMs it needs to create / delete and how it should allocate containers on all VMs to satisfy all **plan** requirements. Current **allocation** is information about currently used VMs, containers and tiers running on them.

The **hierarchical** plan takes VMs management on its own and specifies tiers resources demand for each VM separately. It says that on this VM, this tier needs this number of CPU cores and this number of available RAM.

Also a continuously adjusted system requires some triggers to be run on the adjustment. The system requires two types of triggers: one that runs on each container after scaling (container create / update) and another that runs after some dependee tier (container) is scaled. We will call these triggers **hooks**. For example the first **scale hooks** are used when we need to adjust a number of threads considering how many resources container has. The second **tier hooks** are used by load balancer, which may need to change weights for tiers after dependee containers are changed. Also **tier hooks** can be used to provide information about dependee tiers: for example JBOSS tiers need the IP address of the DB tier.

2.2 Specific requirements

This section contains all requirements for the Executor: functional, non-functional and constraints. Each requirement is described in the following sections:

Requirement ID	Uniquely identifiers requirement
Title	Gives the requirement a symbolic name
Description	The definition of the requirement
Priority	Defines the order in which requirements should be implemented. Priorities are designated (highest to lowest) from 1 to 3. Requirements of priority 1 are mandatory; 2 represents ”nice to have” features , and 3 represents optional features.
Risk	Specifies the risk of not implementing the requirement. It shows how critical the requirement is to the system as a whole. The following risk levels are defined over the impact of not being implemented correctly. • **Critical (C)** It will break the main functionality of the system. The system cannot be used if this requirement is not implemented. • **High (H)** It will impact the main functionality of the system. Some function of the system could be inaccessible, but the system can be generally used. • **Medium (M)** It will impact some system features, but not the main functionality. The system can still be used with some limitation. • **Low (L)** The system can be used without limitation, but with some workarounds.

2.2.1 Functional requirements

Requirement ID	FR-0
Title	The user should set topology of the system
Description	The topology should include: • Infrastructure description – Driver used (Vagrant, AWS) – AWS autoscaling group name – Credentials (Optional) • Max VMs value • Application list • Tiers list of each applications • Scalability of the tier • Docker image of the tier • Scale hooks • The tier dependencies list • Tier hooks •
Priority	1
Risk	C

Requirement ID	FR-1
Title	The user should emulate a monolithic plan execution
Description	The user should provide the monolithic plan to the system and get the result as list of actions executed: VMs created / deleted, containers created / deleted / updated, scale hooks run and tier hooks run
Priority	2
Risk	M

Requirement ID	FR-2
Title	The user should execute a monolithic plan
Description	The plan should describe the required resources for all the tiers of the application.
Priority	1
Risk	C

Requirement ID	FR-3
Title	The user should see the current allocation
Description	The allocation should include • VMs IP addresses • VMs containers • container resources (CPU cores and RAM)
Priority	2
Risk	M

Requirement ID	FR-4
Title	The user should see the docker information about running containers
Description	The user should see output of ”docker inspect” command for each container on each VM. The interested information is ”cpuset” and RAM used by the docker container.
Priority	2
Risk	M

Requirement ID	FR-5
Title	The user should emulate a hierarchical plan execution on a VM
Description	Each VM should accept a hierarchical plan and emulate its execution, showing updated / created / deleted containers and scale-hooks that should be executed.
Priority	2
Risk	H

Requirement ID	FR-6
Title	The user should execute a hierarchical plan
Description	The user should execute a hierarchical plan on the VM
Priority	1
Risk	C

Requirement ID	FR-7
Title	The system should distinguish the scalable and not scalable tiers in the topology
Description	The maximum number of containers for the not scalable tier is 1. For the both scalable and not scalable tiers the minimum number of containers is 0.
Priority	3
Risk	L

Requirement ID	FR-8
Title	The system should run scale-hooks after containers create / update actions
Description	Each VM after creating / updating containers check the topology if scale-hooks specified for this tier and run them. Scale-hook is a bash script that get as an input 4 integer arguments: initial CPU cores usage, initial RAM memory units usage, new CPU cores usage, new RAM memory units usage.
Priority	2
Risk	M

Requirement ID	FR-9
Title	The system should run tier-hooks after creating / deleting / updating of dependee tier containers
Description	The topology should specify an optional dependency between tiers. The type of dependencies supported is: 1-to-n (a dependency from a not scalable tier to a scalable), 1-to-1 (a dependency from a not scalable tier to a not scalable), n-to-1 (a dependency from a scalable tier to a not scalable). The dependency n-to-n (from a scalable tier to a sclalable tier) is not required. The tier-hook is a bash script that gets as an input 3 string arguments: a dependent tier name, a dependee tier name and a new allocation JSON stringified.
Priority	2
Risk	M

Requirement ID	FR-11
Title	The hierarchical plan should processed through the master node
Description	Besides the fact that the hierarchical plan can be run directly on the agent node, we should have a 1 entry point (the master node) to manage dependencies and tier-hooks.
Priority	2
Risk	M

Requirement ID	FR-11
Title	The hierarchical plan should be executed only when messages from each planner have arrived
Description	The hierarchical planner sends the new plan from each container, but we can not proceed immediately these plans due to dependencies. So plans should be processed considering dependencies only after the executor received plans from each container (tier).
Priority	2
Risk	M

2.2.2 Non functional requirements

1. (NFR-0) A monolithic executor should not create / use VMs, if it is possible to allocate resources on less number of VMs.
2. (NFR-1) An allocation should use resources (CPU cores and RAM) equal to a demand (required in a plan).
3. (NFR-2) A move from a previous allocation to a new one should have minimized weight of actions. Possible actions: VM create, VM delete, VM use, container create, container delete, container update, container use. The weights can be tuned, but the order of weights should be:
   VM create > VM use > VM delete and
   container create > container update > container use > container delete
4. (NFR-3) A monolithic plan that requires about 100 tiers and 50 VMs should be emulated in less than 10s
5. (NFR-4) All the requests should be closed by the authorisation.
6. (NFR-5) Actions of the plan should run in the order of dependencies. Nodes that are dependees should be run first. A dependency graph should be taken into the account.
7. (NFR-6) Containers should be created before the delete action, otherwise the application will not have enough resources in the moment between containers were deleted and before they are created.

2.2.3 Use case diagram

The use case diagram shows main user actions. It is considered that for the monolithic plan the user runs these actions on the main node, but for the distributed executor on the agent (the virtual machine)

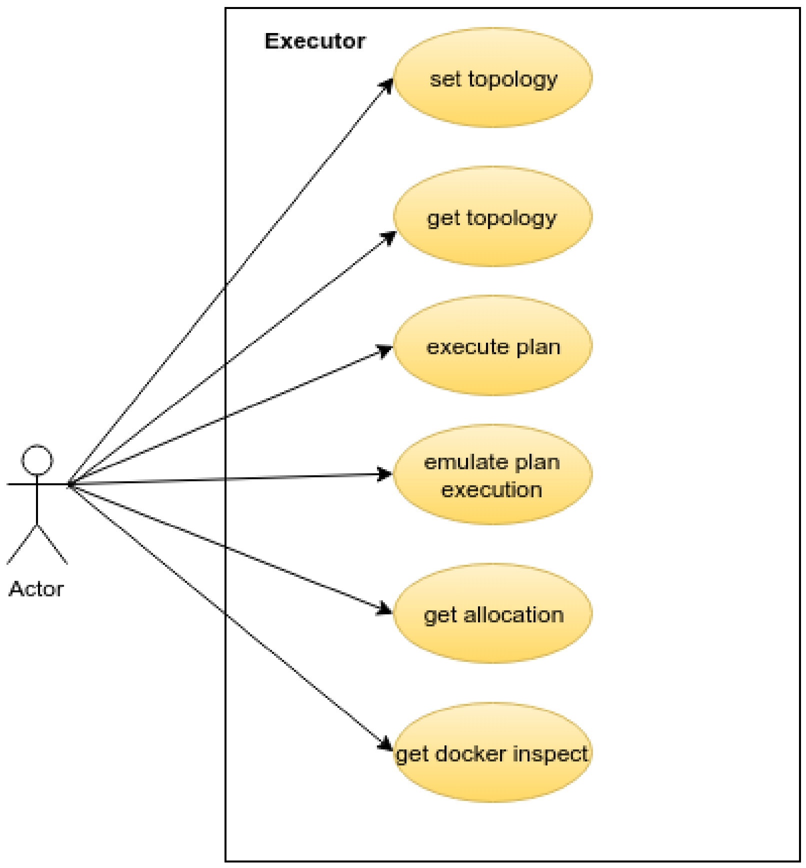


Figure 2.1: Use-case diagram

Chapter 3

Solution design

The executor contains of two parts: the main node and the agents. The agent runs on the created VMs, manages starting / updating / removing of the docker containers, executing of the distributed plan and provides information about current allocation and docker inspect output. The main node manages executing of the monolithic plan, and also provides current allocation and docker inspect output. The main node provides the allocation by sending requests to each known agent and aggregating answers.

3.1 Problems

3.1.1 Bin packing problem

The monolithic approach given current allocation and the monolithic plan (resource demand for each tier) should do creation / deletion of VMs and allocation containers on them. This problem is similar to well-known Bin packing problem. A VM for us is a bin and a container is an object which we need to pack in the bin. The container has two dimensions: CPU cores and RAM, so this is 2D bin packing problem. The main difference to bin packing problem is that we have initial allocation (initial packing) and we should not only minimize number of bins used, but also number of movements required to reach the final allocation from the initial one.

The linear optimisation problem was formulated to solve this 2d bin packing problem:

$$\text{minimize} \sum_{i \in I} \sum_{j \in J} (w_{0ij}\alpha_{0ij} + w_{1ij}\alpha_{1ij}) + \sum_{i \in I} (w_{2i}\beta_{0i} + w_{3i}\beta_{1i})$$

subject to

$$\begin{array}{lllr}
(1) & \sum_{j \in J} x_{ij} \le m_{0i}\beta_{0i} & & \forall i \in I \\
(2) & \sum_{j \in J} y_{ij} \le m_{1i}\beta_{0i} & & \forall i \in I \\
(3) & x_{ij} \le m_{0i}\alpha_{ij} & & \forall i \in I, j \in J \\
(4) & y_{ij} \le m_{1i}\alpha_{ij} & & \forall i \in I, j \in J \\
(5) & \sum_{i \in I} x_{ij} = d_{0j} & & \forall j \in J \\
(6) & \sum_{i \in I} y_{ij} = d_{1j} & & \forall j \in J \\
(7) & m_{1i}x_{ij} \ge y_{ij} & & \forall i \in I, j \in J \\
(8) & m_{0i}y_{ij} \ge x_{ij} & & \forall i \in I, j \in J \\
(9) & \alpha_{0ij} + \alpha_{1ij} = 1 & & \forall i \in I, j \in J \\
(10) & \beta_{0i} + \beta_{1i} = 1 & & \forall i \in I \\
 & \alpha_{0ij} \in \{0,1\} & & \forall i \in I, j \in J \\
 & \alpha_{1ij} \in \{0,1\} & & \forall i \in I, j \in J \\
 & \beta_{0i} \in \{0,1\} & & \forall i \in I \\
 & \beta_{1i} \in \{0,1\} & & \forall i \in I \\
 & x_{ij} \in \mathbb{Z} & x_{ij} \ge 0 & \forall i \in I, j \in J \\
 & y_{ij} \in \mathbb{Z} & y_{ij} \ge 0 & \forall i \in I, j \in J
\end{array}$$

where

I is the set of VMs,

J is the set of Tiers,

α_{0ij} is "tier_is_used" binary variable that is true if we allocate container for tier[j] on VM[i], α_{1ij} is "tier_is_idle" binary variable that is true only if α_{0ij} is false, the equation (9) links them together, β_{0i} is "vm_is_used" binary variable that is true if we allocate any container for any tier on VM[i], β_{1i} is "vm_is_idle" binary variable that is true only if β_{0i} is false, the equation (10) links them together, x_{ij} is "cpu_usage" variable: the number of CPU cores that tier[j] uses on VM[i], y_{ij} is "mem_usage" variable: "the number of RAM units (1 unit = 512Mb) that tier[j] uses on VM[i], (1) is CPU availability constraint and constraint for activation of "vm_is_used" variable, constant m_{0i} is the number of CPU cores on VM[i] (maximum allowed value for x_{ij}), (2) is the same as (1), but for the RAM units, constant m_{1i} is the number of RAM units on VM[i] (maximum allowed value for y_{ij}) (3) and (4) are activation of α_{ij} constraints similar to (1) and (2). (3) is for CPU cores, (4) is for RAM units, (6) is the RAM units demand equation, where d_{1j} is the RAM units demand from the plan for the tier[j], (5) is the same as (4), but for the CPU cores, d_{0j} is the CPU cores demand from the plan for the tier[j], (7) says that if tier[j] uses some RAM on VM[i], then it must use some CPU cores, (8) is the same as constraint (7), but other way round: if tier[j] uses some CPU cores on VM[i], then it must use some RAM units.

This formulation of the problem requires us to know the number of VMs beforehand, which we do not know. So the upper bound number of VMs is calculated and provided to the ILP solver. VMs that are used in the initial allocation are already created, but there are also VMs that are empty and are only going to be created if there will be some container on it.

Considering the initial allocation is done by constants $w_{0ij}, w_{1ij}, w_{2i}, w_{3i}$. Where w_{0ij} is the constant for using tier[j] on VM[i], so knowing initial allocation we can set this constant to the cost of the container creation if there is no tier[j] on VM[i] in the initial allocation, and the cost of the container update if there is tier[j] on VM[i] in the initial allocation. Here is a pitfall that if in the initial allocation container was used on the VM, and we use it in the new allocation, we can not differentiate if the value of CPU cores and RAM units used has changed (we need to run container update command) or is not changed (we need do nothing).

The weight constant w_{1ij} is the cost of removing container of the tier[j] from the VM[i], this weight should be 0 if the tier[j] was not on the VM[i] in the initial allocation.

The weight w_{2i} is the cost of VM[i] creation. if VM[i] is already used in the initial allocation this constant is 0.

The weight w_{3i} is the cost of using the VM[i]. This cost is 0, if the VM is not created yet, and it was not used in the initial allocation.

Considering NFR-2, the weights for actions should follow these rules:

1. container delete < container update < container create
2. VM delete < VM use < VM create
3. VM delete + container delete + container create <
   VM use + container update

The rule number 3 is introduced, because it is possible a situation when for example we can remove some VM[p], this VM had in the initial allocation 1 container. The weight of vm_usage=30, the weight of vm_deletion=25, the weight of container_update=10, the weight of container_deletion=5, the weight of container_create=15. So to continue to use this VM the objective function will have $vm\_usage + container\_update = 30 + 10 = 40$, and to remove this container in the worst case we need to create container in another place: $vm\_deletion + container\_deletion + container\_create = 25 + 5 + 15 = 45$. So ILP minimizing the objective function will choose not to remove the VM[p], but keep it. This is against our requirement NFR-0 and we want to keep the minimum number of VMs possible.

Also NFR-2 requires weight of "container use" be less than "container update", but I could not found the easy way to differentiate them in ILP. As we have initial allocation as constants, for example c_{ij} is CPU cores used by container[j] on VM[i] and m_{ij} is the same for RAM units. The new allocation in the ILP are variables x_{ij} and y_{ij} for CPU cores and RAM units accordingly. So to differentiate "container use" from "container update", we need either $x_{ij} + y_{ij} \neq c_{ij} + m_{ij}$ or $abs(x_{ij} + y_{ij} - c_{ij} - m_{ij}) > 0$ or some "if" constraint that all makes our ILP non linear. The effective workaround to solve this issue is considered out of scope of this paper.

3.1.2 Scalability

The tier can be scalable or not. The not scalable tier can have only 1 or 0 containers. If the tier is absent in the plan, it will have 0 containers, otherwise it will be 1, whilst the scalable tier can have any number of containers ≥ 0 allocated on different VMs. In the ILP formulation not scalable containers are not considered, so the current workaround is too

3.1.3 Limitation of the ILP formulation

This ILP formulation has some limitations:

1. This formulation of the problem requires us to know the number of VMs beforehand, which we do not know. But we can make upper-bound estimation.

2. This formulation does not differentiate between "container update" and "container use" (do nothing with container).

3. Weights should be chosen wisely to satisfy NFR-0.

4. Bin packing problem is NP-hard

5. We have 1 optimal solution and can not choose between several different solutions

3.1.4 Hooks

Hooks are scripts that trigger on the tier / container scaling. The goal is to adjust the application's configuration after the allocation is changed.

We tried to extract and describe use cases of adjusting required:

1. Jboss needs to adapt the number of threads considering the number of CPU allocated.
2. LoadBalancer needs to be provided with the list of Jboss containers and resources allocated to adapt it weights.
3. Jboss needs to be provided with the DB address on container creation / change.

The adjustment 1 should be run for the container after it is created or updated. We call this adjustment "scale-hook" or "on_node_scale" hook. The script should be provided with 4 arguments: previous CPU cores allocated, previous RAM memory units allocated, and new CPU cores and RAM memory units allocated.

In the 2nd adjustment the LoadBalancer tier depends on the Jboss tier. So the dependencies should be specified. Also the dependent node (LoadBalancer) should wait until all the containers of the dependee are processed (JBOSS). After that the adjustment script should be run on all the containers of the dependent node (LoadBalancer). We call this adjustment "tier-hook" or "on_dependency_scale" hook.

The 3d adjustment should be run only on the container start. To simpliy the first version of the application we decided not to implement this adjustment for the dependecy 1-to-n. Only for the dependencies n-to-1 and 1-to-1 after the dependee tier is processed, the ip of the dependee tier is provided to the dependent node on the creation (parameter "add-host").

This creates some limitations: the dependee container should be processed before dependent containers. For example, if the first plan will have only dependent containers, in the next plan it will be added dependee container, and in the last plan the dependence between them will be specified, then "add-host" feature will not be propagated and the dependee containers will not know about the dependant one.

Similar problem if we decided to remove the not-scalable tier from the plan, the dependee containers will have not valid "add-host" link and even if later we put the "not-scalable" tier back to the plan, it will have another IP address.

So considering these limitations we suggest one of the two workarounds.

The first is to manage the DB dependencies out of the topology and provide the IP address in the topology as "docker parameter".

The second is just use docker "add-host" feature as it. So if the dependent container is removed, or the dependency is added, the user should be aware that "add-host" will not be specified.

3.1.5 Dependencies

Considering the requirement FR-9 only 1-to-1, 1-to-n and n-to-1 dependencies are supported. The dependencies between scalable tiers are not supported. The dependencies are only used to manager tier-hooks and . The flow for container allocation and hooks execution should be:

1. get tiers dependency graph (G)
2. get set of Tiers without outgoing dependencies (T)
3. run container create / update / delete for each tier in T
4. run scale hooks if needed for each tier container in T
5. run tier hooks if needed for each tier container in T
6. remove T from G
7. if G is not empty go to 2

3.2 Approaches

Besides the ILP approach to solve container allocation problem discussed above it was also tried another approache: Constraint Satisfaction Problem.

3.2.1 Constraint Satisfaction Problem

CSP is defined as a triple <X, D, C>, where X is a set of variables, D is a set of the respective domains of values, C is a set of constraints.

In our context we have 2 matrices of variables as X:
x_{ij} is a matrix that specify how many CPU is used by Tier[j] on VM[i],
y_{ij} is a matrix that specify how many RAM units are used by Tier[j] on VM[i].

The domain D for us is all possible values of CPU cores and RAM units. For example if we have AWS t2.medium with 2 CPU cores and 8 RAM units, we have:
d_{0ij} is $\{0, 1, 2\}$ for CPU cores,
d_{1ij} is $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ for RAM mem units.

For a set of constraints C we should consider similar to ILP:

- availability constraints on each VM for CPU cores and for RAM mem units
- demand constraint considering the plan
- 0 CPU cores with RAM > 0 or other way round

3.2.2 Limitation of the CSP

This CSP formulation has also some limitations:

1. Comparing to ILP, changing domain to continuous values may cause problems.
2. The solution of the CSP is a list of different solutions, that can differ only by a permutation. We can have a factorial number of different solutions which we need to look to found the optimal one considering the initial allocation.
3. The problem is NP-Hard and the complexity growth is much faster than the ILP.

The CSP formulation has one insuperable problem that it does not scale. If we have 10 Tiers, 10 VMs, and 16 CPU cores and 64 RAM mem units, the CSP solution does not complete in appropriate time.

3.3 Architecture

The architecture consists of two main components: Executor Main Node and Executor Agent. The main node accepts a monolithic plan, orchestrates agents and has a whole picture of VMs allocated. The agent node can accept a distributed plan to execute, can run / start / stop docker containers, provides the allocation information and output of docker inspect command.

These executor nodes are "Execute" part of the autonomous system MAPE framework. The Monitor component measures different metrics of applications that are running in the containers on the Executor agents, and provide this information to the Analyze component. The Plan component together with the Analyze component produce a new plan (a resource demand for tiers) which is sent to Executor.

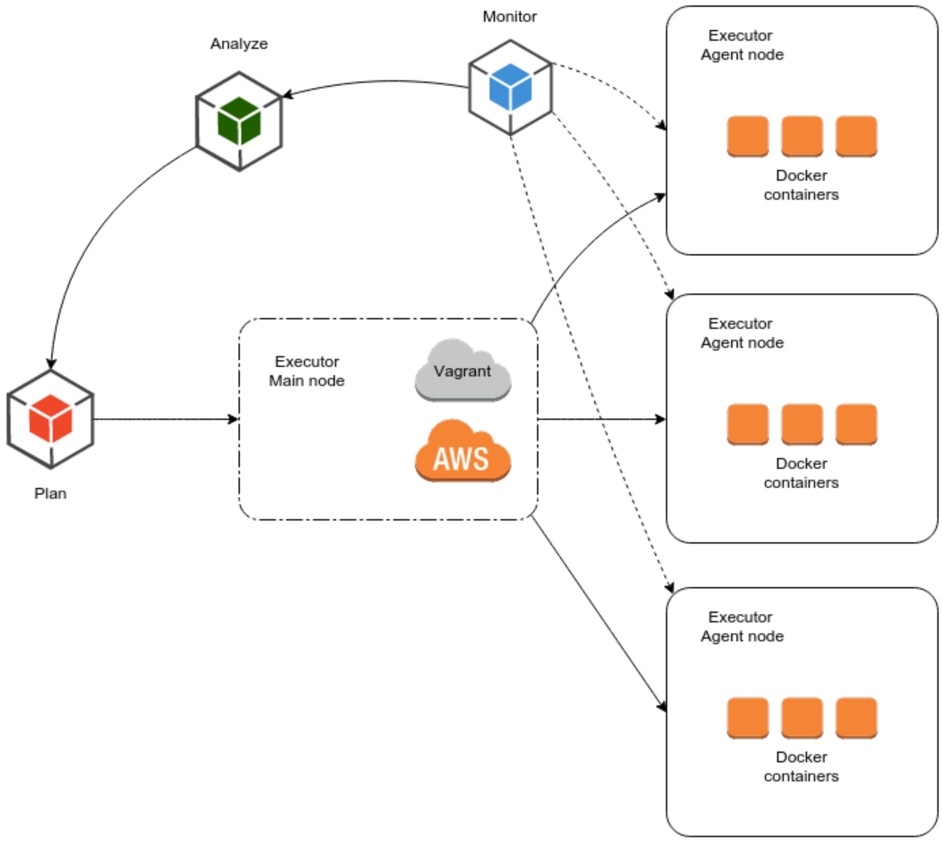


Figure 3.1: Architecture

3.3.1 Components

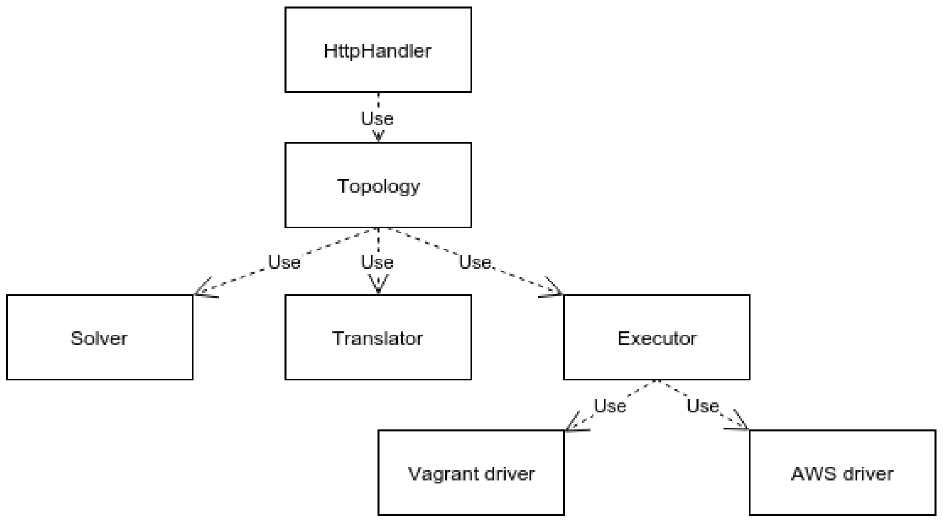


Figure 3.2: Executor main node classes

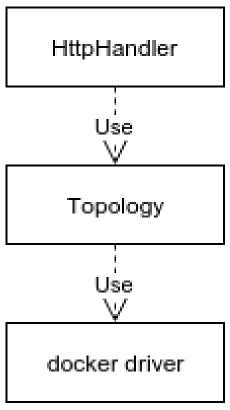


Figure 3.3: Executor agent node classes

Chapter 4

Implementation

4.1 Integer Linear Programming

For prototype the ILP problem formulated in the section 3.1.1 was modeled and solved in AMPL language.

Listing 4.1: AMPL model

```
set Tier;
set VM;

param tier_used{Tier, VM};
param vm_used{VM};
param cpu_demand{Tier};
param mem_demand{Tier};
param cpu_max{VM};
param mem_max{VM};
param use_tier_weight{Tier, VM};
param use_vm_weight{VM};
param not_use_tier_weight{Tier, VM};
param not_use_vm_weight{VM};

var tier_usage{Tier, VM} >= 0 binary;
var vm_usage{VM} >= 0 binary;
var tier_idle{Tier, VM} >= 0 binary;
var vm_idle{VM} >= 0 binary;
var cpu{Tier, VM} >= 0 integer;
var mem{Tier, VM} >= 0 integer;
```

```
minimize cost:
  sum{i in Tier, j in VM}
    (use_tier_weight[i, j] * tier_usage[i, j] +
      not_use_tier_weight[i, j] * tier_idle[i, j]) +
  sum{j in VM} (use_vm_weight[j] * vm_usage[j] +
    not_use_vm_weight[j] * vm_idle[j]);

subject to CPU_availability{j in VM}:
  sum{i in Tier}
    cpu[i, j] <= cpu_max[j] * vm_usage[j];

subject to RAM_availability{j in VM}:
  sum{i in Tier} mem[i, j] <= mem_max[j];

subject to CPU_demand{i in Tier}:
  sum{j in VM} cpu[i, j] = cpu_demand[i];

subject to RAM_demand{i in Tier}:
  sum{j in VM} mem[i, j] = mem_demand[i];

subject to CPU_activation{i in Tier, j in VM}:
  cpu_max[j] * tier_usage[i, j] >= cpu[i, j];

subject to RAM_activation{i in Tier, j in VM}:
  mem_max[j] * tier_usage[i, j] >= mem[i, j];

subject to CPU_RAM_activation{i in Tier, j in VM}:
  mem_max[j] * cpu[i, j] >= mem[i, j];

subject to RAM_CPU_activation{i in Tier, j in VM}:
  cpu_max[j] * mem[i, j] >= cpu[i, j];

subject to link_tier_idle{i in Tier, j in VM}:
  tier_idle[i, j] + tier_usage[i, j] = 1;

subject to link_vm_idle{j in VM}:
  vm_idle[j] + vm_usage[j] = 1;
```

In the executor we solve the problem using *or-tools* and ILP solver by Google. The Solver component accepts the current allocation, the topology and provides as output a new allocation. The new allocation is the input of the Translator component which translates the new allocation to the list of actions to be run. This list of actions is the input to the Executor component.

4.2 API

4.2.1 Main Node

[GET] /api/allocation Returns the allocation JSON of known VMs. In reality, just make calls to VMs and groups answers.

Listing 4.2: Example GET allocation output

```
{
  "52.34.2.170": {
    "rubis_app_server":{
      "cpuset": [0],
      "cpu_cores": 1,
      "mem_units": 2
    },
    "pwitter_app_server":{
      "cpuset": [1],
      "cpu_cores": 1,
      "mem_units": 3
    }
  }
}
```

[GET] /api/inspect Returns the inspect output JSON for known VMs. In reality, just make calls to VMs and groups answers.

Listing 4.3: Example GET inspect output

```
{
  "52.34.2.170": {
    "rubis_app_server":{
     "MemUnits": 2,
      "CpusetCpus": "0",
      "Memory":1073741824
    },
    "pwitter_app_server": {
      "MemUnits": 3,
      "CpusetCpus": "1",
      "Memory": 1610612736
    }
  }
}
```

[GET] /api/topology
Returns JSON with the current topology.

Listing 4.4: Example topology

```
{
  "infrastructure": {
    "cloud_driver": {
      "name": "aws-ec2",
      "autoscaling_group_name": "monolithic-ex-8cpu",
      "credentials": "/usr/me/utils/aws.properties"
    },
    "max_vms": 10,
    "hooks_git_repo": "https://github.com/n43jl/hooks.git"
  },
  "apps": [{
    "name": "rubis",
    "tiers": {
      "loadbalancer": {
        "name": "Front Load Balancer",
        "max_node": 1,
        "docker_image": "haproxy",
        "depends_on": ["app_server"],
        "on_dependency_scale": "reload_server_pool.sh",
        "max_rt": 0.1
      },
      "app_server": {
        "name": "Application Logic Tier",
        "docker_image": "polimi/rubis-jboss",
        "depends_on": ["db"],
        "on_node_scale": "jboss_hook.sh",
        "on_dependency_scale": "reload_connections.sh",
        "ports": ["80:8080"],
         "entrypoint_params": "-w 3 -k eventlet"
      },
      "db": {
        "name": "Data Tier",
        "max_node": 1,
        "docker_image": "mysql",
        "on_node_scale": "mysql_hook.sh",
        "max_rt": 0.2,
        "ports": ["3306:3306"]
      }
    }
  },{
    "name": "pwitter",
    "tiers": {
      "app_server": {
        "name": "Application Logic Tier",
        "docker_image": "pwitter-web",
        "ports": ["8080:5000"],
        "entrypoint_params": " /opt/jboss-4.2.2.GA/bin/run.sh --host=0.0.0.0
        --bootdir=/opt/rubis/rubis-cvs-2008-02-25/Servlets_Hibernate -c default"
      },
      "db": {
        "name": "Data Tier",
        "max_node": 1,
        "docker_image": "mysql",
        "on_node_scale": "mysql_hook.sh",
        "max_rt": 0.2,
        "ports": ["3307:3306"]
      }
    }
  }],
}
```

[PUT] /api/topology
Set the topology. The payload is the same as the example output of the [GET] /api/topology method.

[PUT] /api/translate
Returns JSON with the actions list required to execute provided in the payload plan.

Listing 4.5: Example of the input payload

```
{
  "rubis": {
    "app_server": {
      "cpu_cores": 1,
      "mem_units": 3
    }
  },
  "pwitter": {
    "app_server": {
      "cpu_cores": 1,
      "mem_units": 4
    }
  }
}
```

Listing 4.6: Example of the output

```
{
  "actions":[
    "update container \"rubis_app_server\"
      on the vm "52.34.2.170" set cpu_cores=1 and mem_units=3",
    "update container \"pwitter_app_server\"
      on the vm "52.34.2.170" set cpu_cores=1 and mem_units=4"
  ]
}
```

4.2.2 Agent

[GET] /api/allocation

Returns JSON with the current allocation of the containers on the VM. The difference with the "inspect" command is that "inspect" is the output of "docker inspect" command, while the "allocation" is taken from the app runtime state. e.g. We create using the API some container "Jboss". Both the "allocation" and "inspect" will return 1 container. After that we stop / start our Executor Agent Node. The "allocation" will be empty, but the "inspect" will return 1 container.

Listing 4.7: Example GET allocation output

```
{
  "rubis_app_server":{
    "cpu_cores": 1,
    "mem_units": 2
  },
  "pwitter_app_server":{
    "cpu_cores": 1,
    "mem_units": 3
  }
}
```

[GET] /api/inspect

Returns JSON with the current docker containers running on the VM.

Listing 4.8: Example GET inspect output

```
{
  "rubis_app_server":{
    "MemUnits": 2,
    "CpusetCpus": "0",
    "Memory":1073741824
  },
  "pwitter_app_server": {
    "MemUnits": 3,
    "CpusetCpus": "1",
    "Memory": 1610612736
  }
}
```

[GET] /api/topology
The output is the same as for the main node.

[PUT] /api/topology
The input and the output is the same as for the main node.

[PUT] /api/translate
Returns JSON with the actions list required to execute provided in the payload plan.

Listing 4.9: Example of the input payload

```
{
  "rubis": {
    "app_server": {
      "cpu_cores": 1,
      "mem_units": 3
    }
  },
  "pwitter": {
    "app_server": {
      "cpu_cores": 1,
      "mem_units": 4
    }
  }
}
```

Listing 4.10: Example of the output

```
{
  "actions":[
    "update container \"rubis_app_server\" set cpu_cores=1 and mem_units=3",
    "update container \"pwitter_app_server\" set cpu_cores=1 and mem_units=4"
  ]
}
```

[PUT] /api/topology
Set the topology. The payload is the same as the example output of the [GET] /api/topology method.

[PUT] /api/execute
Returns JSON with the actions list required to execute provided in the payload plan.

Listing 4.11: Example of the input payload

```
{
  "rubis": {
    "app_server": {
      "cpu_cores": 1,
      "mem_units": 3
    }
  },
  "pwitter": {
    "app_server": {
      "cpu_cores": 1,
      "mem_units": 4
    }
  }
}
```

[PUT] /api/run/tier_hooks

Returns JSON with the actions list required to execute provided in the payload plan.

Listing 4.12: Example of the input payload

```
[
  {
    "app": "rubis",
    "dependent": "app_server",
    "depends_on": ["db"],
    "allocation": "..."
  }
]
```

[PUT] /api/docker/run

Runs the docker container specified in the payload.

Listing 4.13: Example of the input payload

```
{
  "name":"pwitter_app_server",
  "cpu_cores": 2,
  "mem_units": 3
}
```

[PUT] /api/docker/remove

Removes the docker container specified in the payload.

Listing 4.14: Example of the input payload

```
{
  "name":"pwitter_app_server"
}
```

[PUT] /api/docker/update

Returns JSON with the actions list required to execute provided in the payload plan.

Listing 4.15: Example of the input payload

```
{
  "name":"pwitter_app_server",
  "cpu_cores": 1,
  "mem_units": 2
}
```

4.3 Topology

There are restrictions for the topology:

1. Ports are specified in the form (host VM port : guest container port). All hosts ports specified for all tiers should be unique, even for different applications.
2. The script name specified in the "on_node_scale" should be located in the repository specified in "hooks_git_repo" in the folder "scale_hooks".
3. The script name specified in the "on_dependency_scale" should be located in the repository specified in "hooks_git_repo" in the folder "tier_hooks".
4. The max_node=1 means that the tier is not scalable. It will not be moved from one VM to another and it must have the resources demand less than it is available on one VM.
5. For tiers with the max_node $>$ 1 is allowed to have dependencies only to tiers with max_node=1 (n-to-1). n-to-n dependencies are not allowed.
6. Dependencies between tiers from different applications are not allowed.
7. Circular dependencies are not allowed

Also tier names should be unique inside the app, and the app name should be unique too. Any tiers of any apps can be on the same VM, so uniqueness of the container name is imposed by using as container name the compound name: "APP_NAME_TIER_NAME".

4.4 Hooks

To adjust to the new allocation they should be provided with input parameters.

The "on_node_scale" hook requires as the input parameter the previous resources and the new. So there are 4 input parameters: previous CPU cores, previous memory units, new CPU cores and new memory units.

The "on_dependency_scale" hook requires the IP address of the dependee tier and new allocation resources. So the input parameters are: the dependee tier name, the dependent tier name and stringified allocation JSON.

4.5 Algorithms

4.5.1 Processing list of actions

The algorithm figure 4.1 describes the workflow in processing the list of actions for containers and processing hooks. First of all the tiers that depends on other tiers should be processed only after the dependee. To support this we build the dependency graph and process tiers without dependencies first and after remove them from the graph.

The second is the order of processing a tier. We should first create containers, after we can do update and only after that we can do remove. This is done to have more resources than it required for the tiers, otherwise the monitor component may detect that applications needs more resources.

All the hooks (scale-hooks and tier-hooks) are run only they are specified for this tier.

4.5.2 Processing plan

Processing the plan is specified in the figure 4.2. While processing the plan, the plan "app -> tier -> demand" is flatten to the form "app_tier -> demand". If the demand is not equal to allocation than the ILP formulation is built and solved. As the result we get the new resource allocation considering VMs and containers with resources on them. The procedure "process list of actions" is described in the figure 4.1.

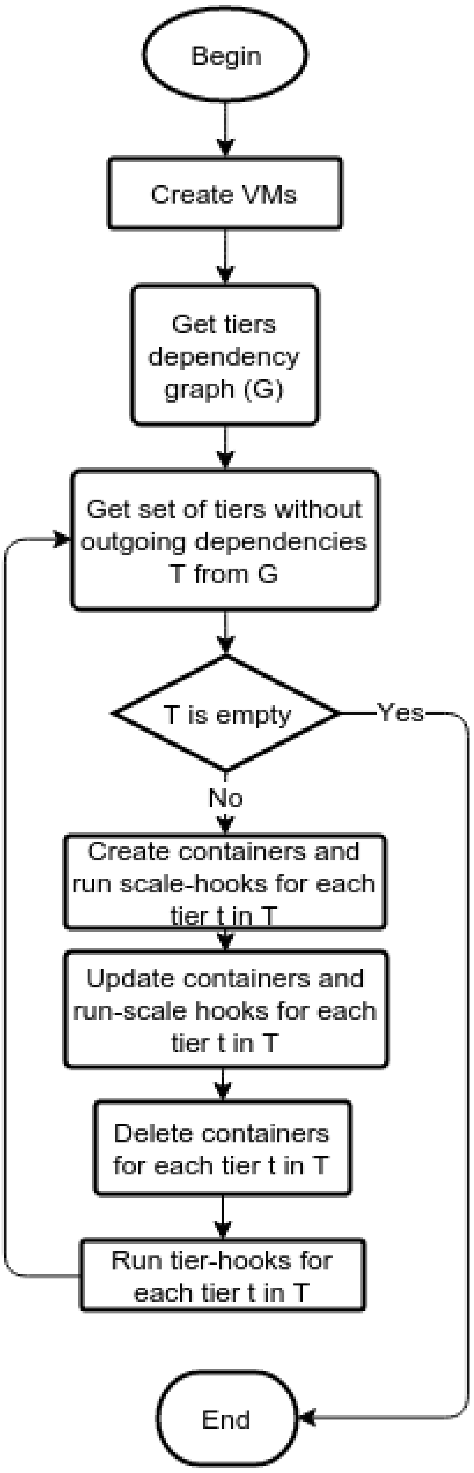
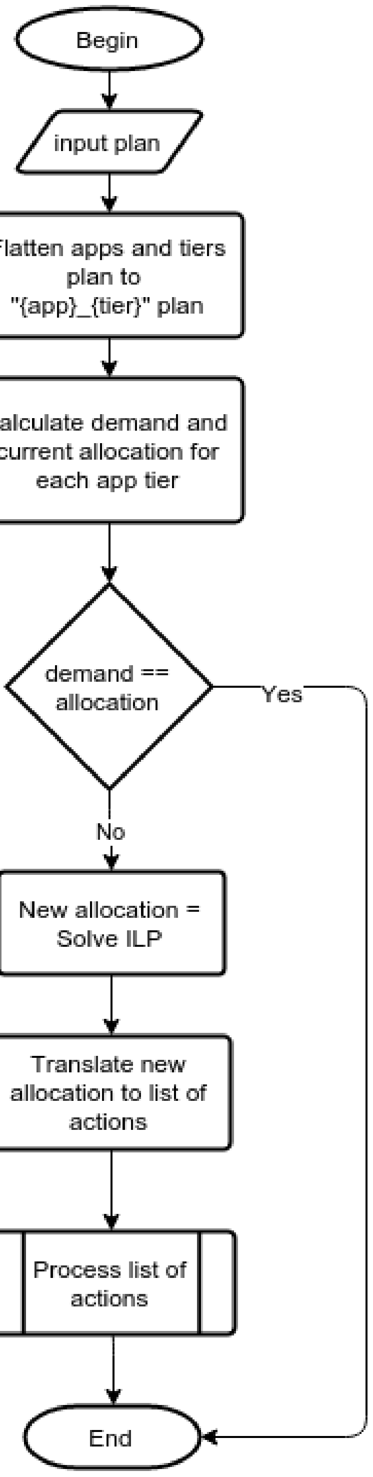

Begin
Create VMs
Get tiers dependency graph (G)
Get set of tiers without outgoing dependencies T from G
T is empty
Yes
No
Create containers and run scale-hooks for each tier t in T
Update containers and run-scale hooks for each tier t in T
Delete containers for each tier t in T
Run tier-hooks for each tier t in T
End

Figure 4.1: Processing list of actions



Begin
input plan
Flatten apps and tiers plan to "{app}_{tier}" plan
calculate demand and current allocation for each app tier
demand == allocation
Yes
No
New allocation = Solve ILP
Translate new allocation to list of actions
Process list of actions
End

Figure 4.2: Processing the plan

4.5.3 Translation

The goal of the translation is to express the new allocation as the list of actions considering the current allocation. The set of possible actions considering we the container with the name "app_tier" and the VM with the IP "127.0.0.1":

- Create VM '127.0.0.1'
- Delete VM '127.0.0.1'
- Create container 'app_tier' on the VM '127.0.0.1' with cpu=2 and mem=2.5gb
- Update container 'app_tier' on the VM '127.0.0.1' set cpu=1 and mem=1gb
- Run scale_hooks for the container 'app_tier' on the VM '127.0.0.1'
- Delete container 'app_tier" on the VM '127.0.0.1'
- Run tier_hooks for the container 'app_tier' on the VM '127.0.0.1'

The algorithm for this translation is:

Listing 4.16: Translation algorithm

```
FOR ALL vm IN current_allocation:
  IF vm IN new_allocation:
    FOR ALL container IN current_allocation[vm]:
      IF not container IN new_allocation[vm]:
        push_delete_container_action
  ELSE:
    push_delete_vm_action
FOR ALL vm IN new_allocation:
  IF vm IN current_allocation:
    FOR ALL container IN new_allocation[vm]:
      IF container IN old_allocation[vm]:
        push_update_container_action_if_needed
        push_run_scale_hooks_if_needed
      ELSE:
        push_create_container_action
        push_run_scale_hooks_if_needed
  ELSE:
    push_create_vm_action
```

```
FOR ALL container IN new_allocation[vm]:
    push_create_container_action
    push_run_scale_hooks_if_needed
```

Scale-hooks run after each container update / create. Tier-hooks can be handled only considering the dependency graph. The executor adds tier-hooks to the list of actions.

4.6 Monolithic vs Hierarchical approach

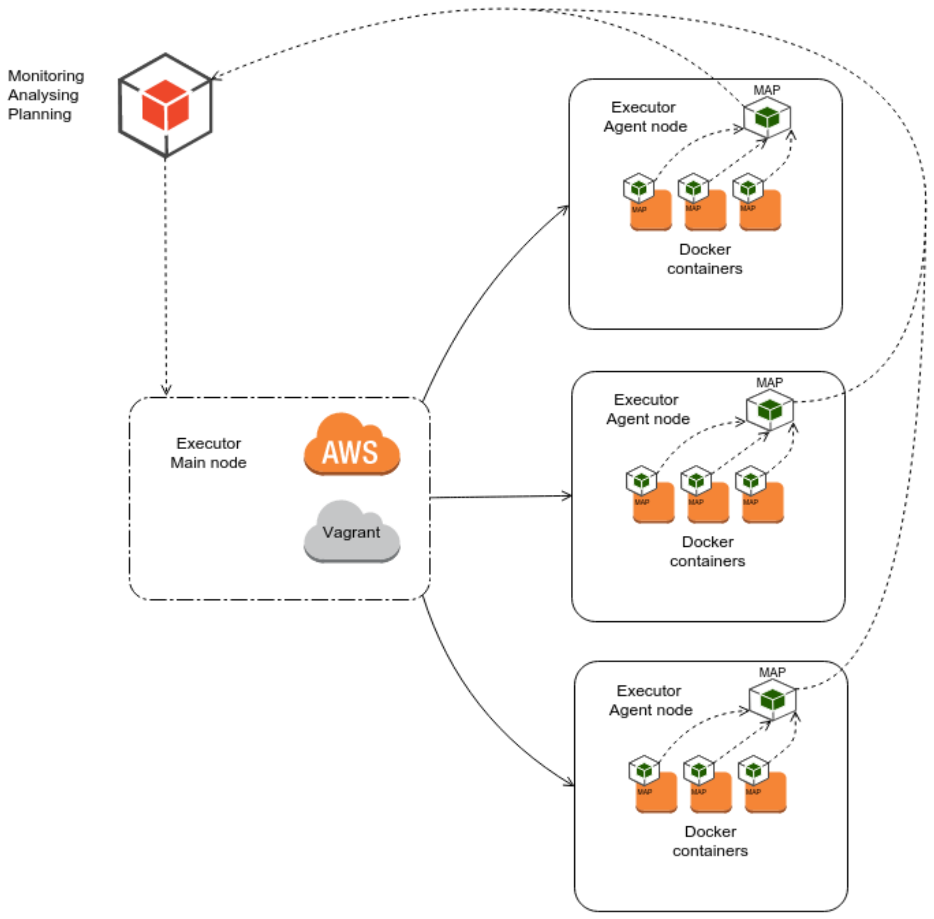


Figure 4.3: Hierarchical Executor

The current approach with solving a plan using ILP formulation we call the "monolithic" approach. Considering that all the implementation is run in the MAPE framework environment for the monolithic approach we have a global planner. Having a global planner has some drawbacks, as the control loop timeout should be at least the time required to run the plan (considering the VM creation take a long time for our test it was approximately 150s).

To go around large control loop timeouts we decided that MAPE components can be hierarchically spread among all the VMs and containers. So instead of having the one global planner, we want to have planner for each container. This we call the hierarchical approach.

In the hierarchical approach for each container there is the planner which produce the plan for its own container. To handle the VM constraints in CPU cores and memory, the VM has another planner that consolidates all containers plan and resolves this constraints. In the future work we are thinking to have the CPU requirement in the plan as the float number, this simplifies the constraint resolving considering each plan.

After plans are built for each VM, it hierarchically goes to the root planner and the root planer provides the plan to the master node executor. The master node executor accepts the plan, but it do not run it until the plans from each VM have arrived. This is because of the tier-hooks. To support tier-hooks we need to execute actions in the particular order (calculated by the dependency-graph), so we can not run some actions, until we know all the actions we need to run.

4.7 Drivers

There were implemented several drivers for managing virtual / cloud infrastructure: Vagrant, AWS, Docker.

As vagrant is an interface (a facade) to large list of different virtualization providers and AWS also supports very large list of vitalization (even Docker containers). This 3 cases covers a lot of different virtualization techniques.

4.7.1 Vagrant driver

Vagrant was used mostly during the implementation stage of the work to test everything locally without paying for Amazon AWS infrastructure.

The image for Vagrant is based on the "ubuntu/trusty64" image with docker and "Ecoware executor agent" installed.

All ports that are required for tiers should be preconfigured in the Vagrantfile. The application "Ecoware executor agent" uses the port 8000, which should be also preconfigured in the Vagrant file. The Vagrant configuration files can be found in the repository.

Vagrant is something like an interface or a facade to list of the virtualization providers. As providers Vagrant supports:

- VirtualBox
- VMware
- Hyper-V
- Docker
- You can develop your own plugin for custom provider

Using Vagrant helps to abstract from different providers and make it easy to change providers without changing our driver.

To manage Vagrant our driver uses the python module *subprocess* to call OS commands like *vagrant up*

4.7.2 AWS driver

For implementing AWS driver it were tested 3 different ways to communicate with AWS:

1. HTTP REST calls
2. AWS management console
3. Python AWS SDK boto3

As all the code for the paper was written in the python, it was decided that using boto3 is the simplest and cleanest way.

In current implementation it was easier to use the AWS Auto Scaling Groups to create VMs. So creating VMs now is based on the "desired capacity" method for Auto Scaling Groups in AWS. So the workflow for creating VMs now is:

1. Get required VM number from ILP solution (n).
2. Set desired capacity for the Auto Scaling Group to n.
3. Poll number of instances in the Auto Scaling Group until it equals to n.
4. Poll the status of all instances of the Auto Scaling Group to be the "in service".
5. Get IP addresses of all VMs.
6. Analyse the previous allocation, the new allocation and separate existed machines with the new.

7. Set the topology to the new VMs, on error sleep 10s and repeat.

The "polling" in this context means to make the request to the AWS and if the result is not satisfactory, sleep 10s, and repeat the request again.

Creating the VM takes approximately 150s.

For the deletion of the VM we can not just set the desired capacity to the smaller number, because we need to delete some particular machine, but not any one from all machines. To remove particular machine we just do 2 actions:

1. Detach the instance from the autoscaling group.
2. Terminate the instance

The AWS infrastructure supports different virtualization techniques:

- HVM
- PV
- Docker containers

HVM stands for Hardware-Assisted Virtualization. PV stands for Paravirtualization. In spite of the fact that these virtualization techniques are different and have its own advantages and disadvantages, AWS helps us to abstract from this and use its unified API.

To configure AWS for using with AWS driver it is required to do some steps:

AMI. The AMI should contain the *Ecoware Executor Agent* and docker installed. The *Ecoware Executor Agent* and docker engine should be in the autostart. All the containers images that will be used for different tiers should be pulled in the docker. Also all the containers should be removed to not have the name collision.

Security Group. Security Group should be configured considering all the ports that containers want to listen to and the *Ecoware Executor Agent* default port 8000.

Launch Configuration. Launch Configuration should be created considering the AMI we want to use, the security group, and the AWS instance type.

Auto Scaling Group. Auto Scaling Group should be created using the Launch Configuration with the initial instance number equals to 0.

4.7.3 Docker driver

Besides the fact that both Vagrant and AWS support Docker containers as under layer virtualization technique (or the provider), we use docker without any facade or interface, but directly. This helps to simplify and separate the separation between fine-grained and coarse-grained virtualization.

Docker is managed as the Vagrant using OS calls with the python module *subprocess*.

The command to run have this structure (example of the run action):

Listing 4.17: Docker run template

```
docker run -itd {PORTS} {HOSTS} --cpuset-cpus={CPUS}
  -m={MEM}m --name={NAME} -v=/ecoware:/ecoware {IMAGE}
  {ENTRY_PARAMS}
```

Listing 4.18: Docker run example

```
docker run -itd -p 8080:5000 --add-host="db:172.31.31.123"
  --cpuset-cpus=0,1 -m=512m --name=rubis_app_server
  -v=/ecoware:/ecoware pwitter-web -w 3 -k eventlet
```

-it option means to allocate a tty for the container process (to run it interactively).
-d option means to run container in the *detached* mode (background mode).
The variable {*PORTS*} is built considering ports specified in the topology.
The variable {*HOSTS*} is built if the tier has dependency n-to-1 or 1-to-1.
Variables {*CPUS*} and {*MEM*} is resources demand from the plan.
The variable {*NAME*} is built from the topology like
"{APP_NAME}_{TIER_NAME}".
-v option means to add a data volume. The format is host_dir:guest_dir. The name of the directory is hardcoded, this is directory where the git repository with hooks is pulled.
The variable {*IMAGE*} is taken from the topology.
The variable {*ENTRY_PARAMS*} is the entry point params also taken from the topology.

4.8 Unit testing

Some of the algorithms can be easily tested with unit-testing.

For example, the ILP algorithm is quite weight sensitive and tuning the weights can break some logic. Moreover there are many different ways how the system can satisfy the new plan and sometimes it is not clear and obvious which way is more optimal. So for the ILP solver it was written a list of the tests.

The input for the test is 3 json files: "plan.json", "result.json" and "allocation.json". So the iLP solver is tested that given the current allocation and the plan, it produces the new allocation equals to specified in the "result.json".

Any arguable ILP solution can be inspected in details and added to the tests.

Also algorithms like "building the dependency graph", "translation from allocation to actions" are tested too.

4.9 UML

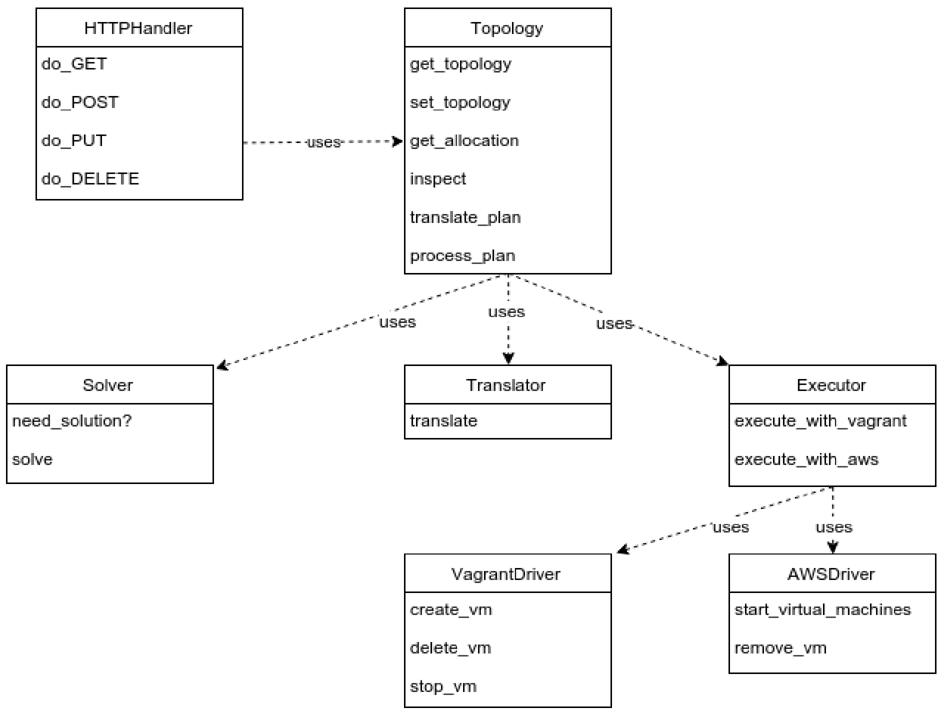


Figure 4.4: Class diagram for the executor on the Main Node

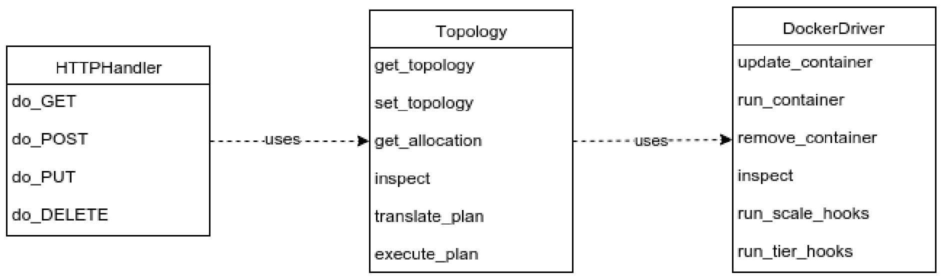


Figure 4.5: Class diagram for the executor on the Agent Node

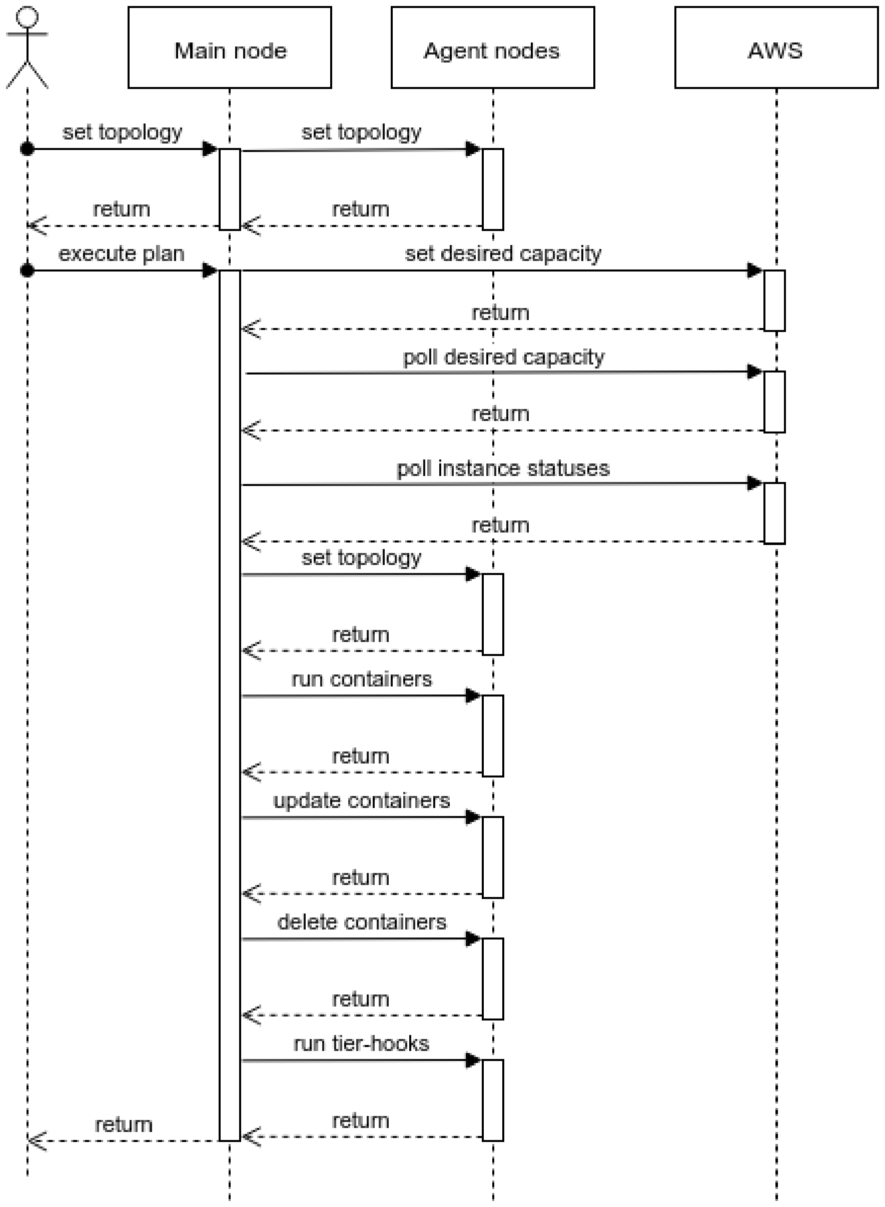


Figure 4.6: Sequence diagram

4.10 Examples

4.10.1 The monolithic executor

For example we have initially empty system without running VMs and containers. The topology of our applications is the same as in the listing 4.4. The executor is running locally on the port 8000: http://localhost:8000/. The autoscaling group ”monolithic_ex_8cpu ” is configured and it uses VMs with 8 CPU cores and 32gb of RAM (64 RAM units).

As the HTTP client we can consider the unix tool ”curl”.

First we set the topology

`curl -X PUT -d @topology.json $endpoint`

where ”topology.json” file contains the same json as in the listing 4.4 and $endpoint is `http://localhost:8000/api/topology`

We can check the new topology opening in any browser `http://localhost:8000/api/topology`.

For example now we want to start only the rubis application, giving 4 CPU cores and 4gb of RAM to ”app_server” and 1 CPU core and 4gb of RAM to the ”db” tier.

For this example the payload.json” should be:

Listing 4.19: payload.json

```
{
  "rubis": {
    "app_server": {
      "cpu_cores": 4,
      "mem_units": 8
    },
    "db": {
      "cpu_cores": 1,
      "mem_units": 8
    }
  }
}
```

First, we can check which actions would be run for this plan by calling endpoint for translation. This call will not run these actions.

`curl -X PUT -d @payload.json $endpoint`

the $endpoint is `http://localhost:8000/api/translate`

After that we can execute the plan by calling the execute endpoint:

`curl -X PUT -d @payload.json $endpoint`

where $endpoint is `http://localhost:8000/api/execute`

We can check new allocation by calling these endpoints from the browser: `http://localhost:8000/api/allocation` and `http://localhost:8000/api/inspect`.

After the planner decides to change the allocation of the containers it should just send to /execute/ endpoint the new payload with the new resources demand.

Chapter 5

Conclusion and future work

5.1 Conclusion

In this work it was implemented the Executor component for the autonomic system based with the adaption based on the MAPE framework. It were considered different virtualization types: coarse-grained the classic one (virtual machines) and the fine-grained container virtualization. During the implementing and experiments we can see advantages of container virtualization: using the fine-grained adaptation capabilities can greatly improve performance when autoscaling cloud-based web-application.

Modern cloud micro services and multitier architectures can benefit from using the autoscaling techniques, providing the decrease in the computational resource consuming, while showing high performance at the same time. The decreasing in the resource consumption it is not only sustainable approach to the environment, but also the reduction of the expenses considering the pay-as-you-go services like AWS.

5.2 Future work

As it is said in the conclusion part the adaptation using containers can greatly improve the performance of the autoscaling. As the evaluation part was not the part of this paper, the evaluation and the proof should be done as the future work.

Also as the future work it is considered the support of other infrastructure or cloud engines: Google App Engine or Microsoft Azure Cloud

Also the future work comprises the integration of feature adaptation by extending the adaptation hook mechanism, an extension of the planner to make it work hierarchically with respect to the controlled resources, and even finer-grained solution to control the CPUs cores allocated to a container, and further evaluation on more case studies of different kinds.

Bibliography

[1] Mieso Denko, Laurence Tianruo Yang, Yan Zhang. *Autonomic Computing and Networking.* Springer Science & Business Media, Jun 12, 2009.

[2] L. Baresi and S. Guinea. *Event-Based Multi-level Service Monitoring.* In Proceedings of the 20th International Conference on Web Services, ICWS, pages 8390, 2013.

[3] F. Seracini, M. Menarini, I. Krueger, L. Baresi, S. Guinea, and G. Quattrocchi. *A Comprehensive Resource Management Solution for Web-based Systems.* In Proceedings of the 11th International Conference on Autonomic Computing, ICAC - to appear, 2014.

[4] L. Baresi, S. Guinea, and G. Quattrocchi. *Distributed Coordinated Adaptation of Cloud-based Applications.* Politecnico di Milano Dipartimento di Elettronica, Informazione e Bioingegneria.

[5] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi *A Discrete-Time Feedback Controller for Containerized Cloud Applications* Politecnico di Milano Dipartimento di Elettronica, Informazione e Bioingegneria.

[6] Amazon EC2 Autoscaling.
https://aws.amazon.com/autoscaling/.

[7] Amazon EC2 Container Service (ECS).
https://aws.amazon.com/ecs/.

[8] Docker.
https://docker.com.

[9] Vagrant.
https://www.vagrantup.com.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *A view of cloud computing.* Commun. ACM, 53(4):50-58, Apr. 2010.

[11] H. Kienle, M. Litoiu, H. MÃ¼ller, M. PezzÃ¨, and M. Shaw. Engineering self-adaptive systems through feedback loops. *In Software engineering for self-adaptive systems.* Pages 48-70. Springer, 2009.

[12] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. *Adaptive, model-driven autoscaling for cloud applications.* In 11th International Conference on Autonomic Computing (ICAC 14), pages 57-64, Philadelphia, PA, June 2014. USENIX Association.

[13] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu. *Resource provisioning for cloud computing. In Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research.* Pages 101-111. IBM Corp., 2009.