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

Fine-grained Autonomic Systems 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 20015-20016

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

Abstract

The world is moving to economy of the nature resources and environment-friendly solutions that can be called sustainable solutions. Approach of sustainability can be also applied to large computer infrastructures to adapt allocated resources depending on the load, on the time of the day or on other input.

Modern Web applications exploit Cloud infrastructures to scale their resources and cope with sudden changes in the workload. While the state of practice is to focus on dynamically adding and removing virtual machines, we also implement and look on more fine-grained solution: operation system-level containerization.

In this paper we present an autoscaling technique that allows containerized applications to scale their resources both at the VM level and at the container level. Furthermore, applications can combine this infrastructural adaptation with platform-level adaptation. The autoscaling is made possible by our planner, which consists of a discrete-time feedback controller.

The work investigates coarse-grained virtualization techniques like virtual machine comparing to light-weight fine-grained techniques like operating system-level virtualization (containers virtualization). The scope is of the thesis is implementation of both techniques and comparing the advantages and disadvantages of each of them.

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 supervisors Luciano Baresi and 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 AutoScaling, which manage when and how an application's resources should be dynamically increase or decrease. This paper describes implementation of alternative solution that help to solve autoscaling problems.

1.1 General idea

This paper describes Autonomous Systems management using different types of virtualization: virtual machines and containers. This Autonomous 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.

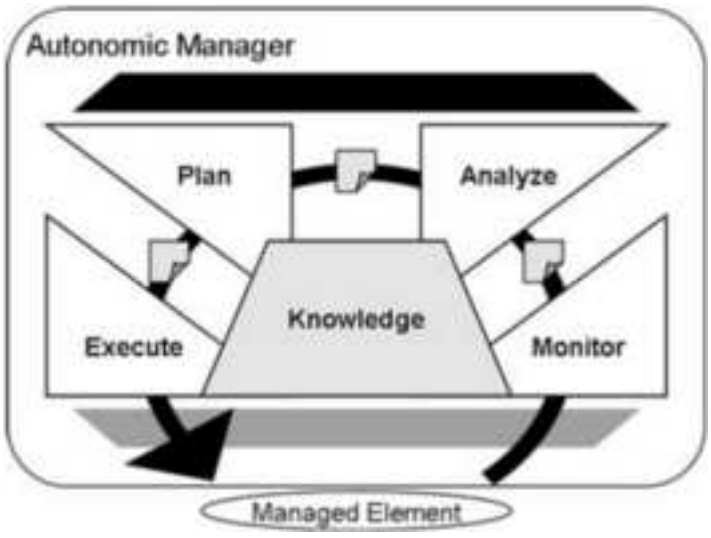

Figure 1.1: Fig. 1 The IBM Autonomic MAPE Reference model

1.4 Cloud and coarse-grained virtualization

Modern web applications more often are run in the cloud. By the cloud it is very often 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. Vagrant and Amazon EC2 drivers do not use vertical scaling features, but only the horizontal one: creating/deling 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.

This is in the opposite to light-weight operation system-level virtualization based on Linux containers. Current implementation uses docker engine for managing containers. With containers vertical scaling (allocation CPU cores and RAM) and horizontal scaling (creating/ deleting new containers) is much faster than the same operation with full virtualization.

1.5 Containers

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

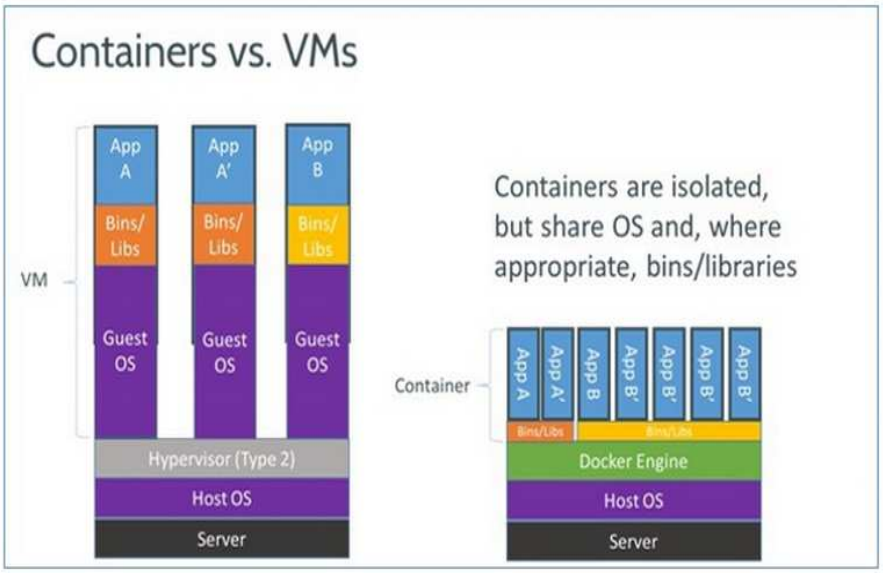

Figure 1.2: Fig. 2 Comparing VM to container

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 "distributed". 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 distributed executor manages only containers on the virtual machines and the plan goes directly to the agent, 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). For coarse-grained virtualization it were implemented drivers for vagrant and amazon elastic compute cloud, that supports a lot of different virtualization types. For containers the driver for docker was implemented.

Also for continuously changing environment it should be implemented some configuration adjustment. 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 are 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 are run for the nodes that depends on the changed tier.

1.7 Thesis structure

The second part describes functional and non functional requirements for the executor module of the MAPE framework. Also it describes solution design, considering initial problems, approaches, decisions made and architecture chosen.

The third part describes the technologies, algorithms and implementation details.

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. Under 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 change topology request. The **plan** says how many resources is required for each tier.

The plan can be of two different types: the **monolithic** one and the **distributed**. The **monolithic** plan means that it 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** it is information about currently used VMs and containers and tiers running on them.

The **distributed** 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 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 be needs 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 needs IP address of 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 features nice to have, 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 includes: • Infrastructure description – Driver used (Vagrant, AWS) – VM resources (CPU cores and RAM) – VM image – 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 the 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 the 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 system should show the created VMs with its IP address and containers allocated on it.
Priority	1
Risk	C

Chapter 3

Impostazione del problema di ricerca

"Bud: Apri!
Cattivo: Perché, altrimenti vi arrabbiate?
Bud e Terence: Siamo già arrabbiati!"

Altrimenti ci arrabbiamo

In questa sezione si deve descrivere l'obiettivo della ricerca, le problematiche affrontate ed eventuali definizioni preliminari nel caso la tesi sia di carattere teorico.

Chapter 4

Progetto logico della soluzione del problema

"Bud: No, calma, calma, stiamo calmi, noi siamo su un'isola deserta, e per il momento non t'ammazzo perché mi potresti servire come cibo ..."

Chi trova un amico trova un tesoro

In questa sezione si spiega come è stato affrontato il problema concettualmente, la soluzione logica che ne è seguita senza la documentazione.

Chapter 5

Architettura del sistema

> *"Terence: Ma scusa di che ti preoccupi, i piedipiatti hanno altro a cui pensare, in questo momento stanno cercando due cadaveri scomparsi*
> *Bud: Se non spegni quella sirena uno di quei due cadaveri scomparsi lo trovano di sicuro!"*
>
> Nati con la camicia

Si mostra il progetto dell'architettura del sistema con i vari moduli.

Chapter 6

Realizzazioni sperimentali e valutazione

"Bambino: Questo è l'ultimo avviso per voi e i vostri rubagalline
Il pistolero si alza: Che avete detto?
Bambino: RUBAGALLINE
Il pistolero si risiede: Aaah."

Lo chiamavano Trinità ...

Si mostra il progetto dal punto di vista sperimentale, le cose materialmente realizzate. In questa sezione si mostrano le attività sperimentali svolte, si illustra il funzionamento del sistema (a grandi linee) e si spiegano i risultati ottenuti con la loro valutazione critica. Bisogna introdurre dati sulla complessità degli algoritmi e valutare l'efficienza del sistema.

Chapter 7

Direzioni future di ricerca e conclusioni

"Terence: Mi fai un gelato anche a me? Lo vorrei di pistacchio.
Bud: Non ce l'ho il pistacchio. C'ho la vaniglia, cioccolato, fragola, limone e caffé.
Terence: Ah bene. Allora fammi un cono di vaniglia e di pistacchio.
Bud: No, non ce l'ho il pistacchio. C'ho la vaniglia, cioccolato, fragola, limone e caffé.
Terence: Ah, va bene. Allora vediamo un po', fammelo al cioccolato, tutto coperto di pistacchio.
Bud: Ehi, macché sei sordo? Ti ho detto che il pistacchio non ce l'ho!
Terence: Ok ok, non c'è bisogno che t'arrabbi, no? Insomma, di che ce l'hai?
Bud: Ce l'ho di vaniglia, cioccolato, fragola, limone e caffé!
Terence: Ah, ho capito. Allora fammene uno misto: mettici la fragola, il cioccolato, la vaniglia, il limone e il caffé. Charlie, mi raccomando il pistacchio, eh."

Pari e dispari

Si mostrano le prospettive future di ricerca nell'area dove si è svolto il lavoro. Talvolta questa sezione può essere l'ultima sottosezione della precedente. Nelle conclusioni si deve richiamare l'area, lo scopo della tesi, cosa è stato fatto,come si valuta quello che si è fatto e si enfatizzano le prospettive future per mostrare come andare avanti nell'area di studio.

Bibliography

Appendix A

Documentazione del progetto logico

Documentazione del progetto logico dove si documenta il progetto logico del sistema e se è il caso si mostra la progettazione in grande del SW e dell'HW. Quest'appendice mostra l'architettura logica implementativa (nella Sezione 4 c'era la descrizione, qui ci vanno gli schemi a blocchi e i diagrammi).

Appendix B

Documentazione della programmazione

Documentazione della programmazione in piccolo dove si mostra la struttura ed eventualmente l'albero di Jackson.

Appendix C

Listato

Il listato (o solo parti rilevanti di questo, se risulta particolarmente esteso) con l'autodocumentazione relativa.

Appendix D

Il manuale utente

Manuale utente per l'utilizzo del sistema

Appendix E

Esempio di impiego

Un esempio di impiego del sistema realizzato.

Appendix F

Datasheet

Eventuali Datasheet di riferimento.