Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

TU WIEN FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Improving elastic testing using container-based virtualisation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Preißler
Matrikelnummer 1129560

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Dr. Alessio Gambi, PhD

Wien, 19. August 2016

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Improving elastic testing using container-based virtualisation

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Thomas Preißler
Registration Number 1129560

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Dr. Alessio Gambi, PhD

Vienna, 19. August 2016 ____________________ (Signature of Author) ____________________ (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Thomas Preißler
Strozzigasse 6-8, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Ganz besonderen Dank gilt meinem Betreuer, Alessio Gambi. Durch seine fortwährende, stets kompetente und zuvorkommende Hilfe war es mir erst möglich diese Arbeit zu erstellen. Ohne Alessio würde diese Arbeit nicht in dieser Qualität und und Form möglich gewesen.

Meine bezaubernde Freundin Manu gab mir einen letzten Motivationsschub um diese Arbeit nach langer Zeit endlich zu beenden. Um mit ihr gemeinsam in einen neuen Lebensabschnitt zu starten ist es an der Zeit mein Studium zu beenden. Sie lehrte mir, wie wichtig es ist, sich auf das Leben zu konzentrieren und gesteckte Ziele zu erreichen.

Danken möchte ich auch meiner guten Freundin Simone. Das ganze Studium hindurch hat sie mich begleitet und stand auch in schwierigen Prüfungszeiten an meiner Seite. Besonders durch ihre große mentale Unterstützung war sie mir bei dieser Arbeit eine unabdingbare Hilfe.

Zuletzt möchte ich mich auch bei meinen Eltern und Schwestern bedanken. Alle Vier gaben mir stets Rückhalt und Unterstützung, egal wie stressig oder nervenaufreibend diese Arbeit war.

Abstract

Elastic web applications automatically scale up and down depending on the current workload. Scaling up and down enables to distribute the application workload to more or less servers. Such elastic applications are getting popular because of the cheap and high-available public clouds as provided by Amazon AWS, Microsoft Azure and others.

An elastic application contains an algorithm, the so-called elasticity policy, that decides how much and when to scale up and down the application depending on various inputs. There are inputs such as the current number of requests, CPU load, memory consumption and many more. On public clouds, the computing power of the available resources may fluctuate, therefore the elasticity policy cannot assume constant and continuous performance. In order to ensure that the algorithm behaves as expected it needs to be tested against such brittle behaviour.

Usually, when testing elasticity policies one takes a real cloud instances and simulates some workloads. This approach is quite cumbersome and quite expensive. Using a real cloud costs money, it takes time to get results and the fluctuation of available resources cannot be controlled.

This thesis proposes a system that employs container-based virtualisation to test an elastic algorithm without the need of a real cloud. Container virtualisation allows to run multiple instances of an application strictly separated from each other and with very little overhead compared to real virtualisation.

The proposed system is able to start multiple instances of the system under test in parallel on the local development machine using container-based virtualisation. By offering an API the available CPU time and memory can be controlled in order to simulate resource shortage. An integration to JUnit offers an easy entry point for testers to write their test cases. There is no need to adjust the system under test if it connects to the API of a cloud provider using an abstraction layer such as JClouds. Because of the local execution of the test environment a test run is free of charge.

Several experiments show the feasibility of the developed system. It can be seen that there exist use cases where bugs of the elastic policy can be found easily when tests are performed with resource limitations as offered by the developed system in the context of this Thesis.

Kurzfassung

Abhängig von veränderter Last skalieren elastische Web-Anwendungen automatisiert um die Anzahl der zur Verfügung stehenden Server anzupassen und so alle Anfragen bedienen zu können ohne stets eine große Anzahl an Servern vorhalten zu müssen. Solche Elastic Applications gewinnen aufgrund der günstigen und weit verbreiteten Clouds, wie sie von Amazon AWS, Microsoft Azure und anderen angeboten werden, zunehmend an Bedeutung.

Eine Elastic Application enthält eine Algorithmus, die sogenannte Skalieriungsstrategie. Diese entscheidet über den Zeitpunkt der Skalierung und über notwendig gewordenen Ressourcen. Eingabeparameter wie die aktuelle Anzahl von parallelen Anfragen, Prozessorauslastung, Speicherauslastung und viele andere bilden eine Entscheidungsgrundlage. Cloud-Anbieter garantieren jedoch keine Ressourcen, diese können stark fluktuieren. Um sicherzustellen, dass ein solcher Skalierungsalgorithmus wie erwartet funktioniert, muss dieser getestet werden.

Oftmals werden für Tests eines solchen Skalierungsalgorithmus echte Cloud-Instanzen herangezogen. Dieser Ansatz ist jedoch schwerfällig und sehr teuer. Eine echte Cloud zu nutzen kostet Geld und Zeit bis Testergebnisse zur Verfügung stehen. Darüber hinaus gibt es keine Steuerungsmöglichkeit für die aktuell verfügbaren Ressourcen.

Diese Arbeit schlägt eine System vor, das eine containerbasierte Virtualisierung nutzt um einen solchen Skalierungsalgorithmus zu testen ohne auf eine reale Cloud zurückgreifen zu müssen. Containervirtualiserung erlaubt mehrere Instanzen einer Anwendung streng voneinander getrennt zu starten und hierbei deutlich weniger Overhead als im Vergleich zu einer Vollvirtualisierung zu erzeugen.

Das vorgeschlagene System kann mehrere Anwendungsinstanzen des zu testenden System auf einer lokalen Maschine betreiben. Durch eine zur Verfügung gestellte API kann die verfügbare CPU-Zeit und der Arbeitsspeicher gesteuert werden um die Fluktuation von verfügbaren Ressourcen zu simulieren. Eine Integration in JUnit erlaubt einen einfachen Einstieg beim Schreiben von Testfällen. Wird eine Abstraktionsschicht wie JClouds genutzt, ist beim zu testenden System keine Anpassung notwendig.

Aufgrund der lokalen Ausführung der Testumgebung fallen keine Kosten an wie für die Nutzung einer Cloud. Verschiedene Experimente zeigen die Machbarkeit des entwickelten Systems. Es wird dargelegt, wie Bugs im Skalierungsalgorithmus sehr einfach gefunden werden können, sobald der Algorithmus auf fluktuierende Ressourcen reagieren muss.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 3
1.3 Organization of this Thesis . . . 3

2 Background **5**
2.1 System Virtualisation . . . 5
2.2 Cloud Computing . . . 6
2.3 Elasticity . . . 7

3 State of the Art **9**
3.1 Testing elastic systems . . . 9
3.2 Cloud Uncertainties . . . 11
3.3 Brownout-aware cloud-based application . . . 12
3.4 Critical Analysis . . . 13

4 Methodology **15**
4.1 Requirements . . . 15
4.2 Design . . . 17

5 Solution Design **23**
5.1 System Overview . . . 23
5.2 High level architecture . . . 24
5.3 Detailed Architecture . . . 26
5.4 Test Case Definition . . . 28
5.5 Technical Details . . . 30

6 Evaluation **35**
6.1 Feasibility . . . 35
6.2 Case Study . . . 37
6.3 Financial and organisational improvements . . . 40

7 Conclusions **41**

8 Future Work **43**
8.1 Extend resource limitations 43
8.2 Test framework integration 44
8.3 Support for Continuous Integration 44

Bibliography **45**

CHAPTER 1

Introduction

1.1 Motivation

Importance of cloud-based applications

Cloud computing has become a significant technology trend. In general, it describes to use computing resources such as processing and storage by a cloud provider instead of local resources on own servers or other devices. [29]

The National Institute of Standards and Technologies (NIST) provides a definition of cloud computing: [32]

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

According to Armbrust et al. [1] there are three main points why cloud computing have been getting popular and widely used nowadays:

- Computing resources are available on demand and almost instantly without the need for planning ahead.
- Companies can start with a low amount of resources and can increase available resources as required without an up-front commitment.
- Billing periods are very short so that computer resources that are no longer needed can be removed on a short-term basis.

In the recent years cloud-based applications gained more importance because of the cheap and highly-available public cloud providers. In it's simple form a cloud-based applications doesn't need to be adjusted to be executed on a public cloud. Common applications such as Wordpress can be taken without modification and deployed to a public cloud provider. [38]

Elastic Applications and their need for tests

Pierre et al. [38] show that an application can even be extended to automatically use more resources if the workload increases. Such an application is called 'elastic'. Elastic cloud-based applications are applications which are able to handle a fluctuating workload. They need to monitor data such as the CPU utilisation, number of requests and many more in order to decide if more computing resources are required. If this is the case, the elastic application connects to the cloud provider API to request new resources. An elastic application reorganises itself, depending on the current workload. Usually, changes in the access rates of a cloud-based application will result in up- and down-scaling tasks.

If such an elastic application is able to scale up and down autonomously it has to be tested if it does this correctly. Additionally, there might be some kind of internal state, such as session information which has to be distributed to all instances of this application.

IaaS provider and their resource uncertainties

An Infrastructure as a Service (IaaS) provider usually offers virtual machines that consist of just a part of the available resources of the physical host machine. In order to maximale their income, the resources are only provided on best-effort. IaaS providers can rent more virtual machine than the physical host has available resources for. This leads to shared resources between all virtual machines on the same physical host. If many customers of the same physical host require much of the offered resources it might happen that not enough resources are available anymore and virtual machines start to fight for resources. This resources contention results in fluctuations on the performance of the virtual machines.

Matthews et al. [31] show that one bad-behaving virtual machine, a so-called Noisy Neighbor, can hugely impact other virtual machines on the same physical host. Depending on the virtualisation technology a single noisy neighbor can result in a performance degradation of 20 % for all other virtual machines. Additionally Ou et al. [35] were able to identify that resources of virtual machines of a public cloud provider can vary up to 30 % even when they should be virtually equal. In addition, there might be long unexpected startup times for newly started virtual machines. As analysed by Mao and Humphrey [30] the startup time strongly depends on the image size and the operating system. But they also observed fluctuations from a few seconds up to seven minutes when the same instance type has been created and started.

Because of the best-effort resource allocation for virtual machines, customers cannot expect a given amount of available resources. It might be the case that resources are not available if they're needed. Such resource uncertainties might result in an unexpected behaviour of an elastic application. It might be the case that a cloud-based application needs much longer to respond

to web requests or cannot retrieve database records in an adequate amount of time. Even worse, the application can run into timing issues that throw errors to the end user.

If one takes into account that resources are provided on a best-effort basis and the actual available resources can be much lower than expected, they can design their cloud-based application so that resource uncertainties can be handled. There exist various possibilities to solve this problem: rent much more resources than actually required, reduce the quality of a provided application and many more.

1.2 Problem Statement

Developers of elastic systems make assumptions about the behavior of cloud infrastructures, their available resources or about the environment in which applications will run [12]. Unfortunately, there are many resource uncertainties even when virtual machine instances should be equal [35]. There are differences in available computing power, network speed, disk latency, actuation delay [13] and many more. Noisy neighbors or unexpected events might be responsible for such uncertainties.

Simulating such cloud uncertainties might be used to get realistic test results for elastic applications. Fast local tests are important to improve the software quality of elastic systems. At the moment there exists no easy way to control the cloud uncertainties during test execution [4], meaning that live tests might not even be able to recreate various, yet realistic, working conditions [12]. Since live tests are not able to cover the possible uncertainties of real environments, their relevance is limited.

To test accurately and effectively cloud-based applications and to improve their software quality there is the need of a novel test environment that allows testers to easily recreate the behaviour of real cloud to test elastic applications under various circumstances such as different CPU , disk and network contention, virtual machine startup time and many more.

In this thesis, I address the problem of testing elasticity algorithms of cloud-based applications. When applications automatically scale up and down depending on the current workload this scaling algorithm might consider various inputs and can be quite complex. To make sure such an algorithm behaves as expected it must be tested. In order to get quickly and unexpensively results, tests should be executed locally on the developer's machine instead of using a public cloud. In order to simulate resource uncertainties the local environment needs to be controllable.

1.3 Organization of this Thesis

The thesis is organised as follows:

- Background information about the technologies of this thesis are given in Chapter 2. It introduces to the reader the main concepts so that the they can understand the ideas of thesis.

- Chapter 3 introduces the State of the Art and gives an overview about related techniques. By looking at various approaches and studies that has been evolved in the past, a fundamental basis of this thesis can be set.
- Chapter 4 describes the approach proposed in this thesis. There can be found some information about requirements, the design of the developed system as well as some development decisions.
- Chapter 5 introduces some parts of the developed system. Later on, in-dept technical details and behavioural information will be given so that the reader gets a good understanding of the developed systems.
- Chapter 6 evaluates the gained results. It describes the advantages and improvements by using the developed system.
- To conclude this thesis Chapter 7 looks back at the thesis and summarises the work.
- Lastly, Chapter 8 gives an outlook of future work that might extend this thesis. It shows ideas that can be used to get the project production-ready or to create new features.

CHAPTER 2

Background

This chapter introduces background concepts that are necessary to understand the idea and the work of this thesis. It introduces the concepts of Clouds, Cloud Computing, Cloud Elasticity and different virtualisation techniques.

2.1 System Virtualisation

Virtualisation in general offers to execute multiple systems or applications in parallel on the same physical machine. The two most important techniques for this thesis are the full virtualisation and the container-based virtualisation. There also exist other virtualisation techniques such as an application virtualisation as offered by the Java Virtual Machine.

Full Virtualisation

Full virtualisation, as offered by VMware ESXi, XEN, KVM and others, allows different operation systems to run in parallel on the same physical machine and offers a high isolation level between each operating system instance. Usually, customers who rent a virtual machine of a public cloud have the flexibility to choose their preferred operation system and have full access to their virtual machine in order to install arbitrary software [33]. This usage scenario is useful on public hosting such as a public cloud or when consolidating different application within a company.

In addition to full virtualisation there also exists the concept of para-virtualisation. From a user's point of view these two techniques are quite similar. But the para-virtualisation requires to use special APIs, a so-called hypervisor, from the virtualised operation system in order to be executed. This offers nearly the same flexibility in the choice of the virtualised guest system, but offers better performance due to reduce virtualisation overhead. [28]

Container Virtualisation

Container virtualisation has been firstly introduced by FreeBSD's Jails [24]. Since then many other solutions for various operating systems arised at the horizon. The first container virtualisation for Linux was "Linux vServer" in 2001 [15].

Linux containers, also called "operating system virtualisation", is a technology to execute and isolate multiple operating system instances on the same physical machine. This technology allows to separate multiple processes from each other. Each process has its own filesystem and there its own libraries and dependencies. The host operating system needs to support container virtualisation and process separation.

In contrast to full virtualisation Container Virtualisation does not allow to virtualise an operation system, it just offers virtualisation within a single instance of an operating system. This decreases the flexibility, because it is impossible to choose an arbitrary operation system. But container virtualisation has less overhead and offers better performance than full virtualisation due to less overhead for executing multiple operating system instances [31]. Soltesz et al. show that a container-based virtualisation offers about two times the performance of a system using full virtualisation. According to Soltesz et al. [42] container virtualisation has its own usage scenario when compared to full virtualisation using a hypervisor. It allows to share a single operating system instance between containers which results in less memory overhead.

But the choice of the virtualisation technique strongly depends on the usage scenario. There cannot be a general suggestion for the one or the other virtualisation. Even when choosing the same technique, different products show a very different performance. [23]

2.2 Cloud Computing

Cloud Computing is a widely used term nowadays. Companies such as Google, Amazon, Microsoft and many others offer some kind of cloud computing product [41]. Starting with some research papers by Google employees in 2003 they set the basement and the idea for these products [2,6,7,16,39,41].

In general Cloud Computing is some kind of computing resources that is offered to customers for rent. This can include virtual machines, software hosted in a data centre, a software stack to execute own programs and many more [44]. Usually, the distinct feature of cloud computing is, that such computing resources are offered on very short contract periods and are available on demand as needed by customers [1]. Hence, cloud computing is also known as on-demand computing [19].

There exists different types of products that can be offered to customers. They can get just bare virtualised resources such as virtual machines without any software. This service model is called Infrastructure as a Service (IaaS) [45]. Many well-known cloud providers such as Amazon, Microsoft, Google or Digital Ocean provide virtual machines as part of their IaaS offering.

In contrast to bare virtualised resources, Platform as a Service (PaaS) vendors offer environments for developers that allows to execute their own software without the need to configure the whole

environment including the software stack [45]. Such software stacks can include pre-configured webservers, databases, background job scheduling, email services and many more so that the developer needs to upload only the application itself. The Google App Engine as an example of a well-known PaaS provider offers such software stacks for applications written in Java, Python, PHP and Go [22].

Even though these definitions try to scratch a quite clear distinction between the two types of services, there is a moving and diffuse line between them in reality.

Cloud provider such as Amazon, Google and Digital Ocean heavily employ virtualisation techniques. It allows the provider to offer smaller and cheaper machines to customers and to run multiple virtual machines on a single physical machine. Additionally using virtualisation adds an abstraction layer above the physical resources so that the provider can change hardware components or the whole physical machine without changing the offered virtual machine.

2.3 Elasticity

The term "Elasticity" is defined by Herbst et al. [20] as follows:

> "Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible."

As this definition states, an elastic application is usually able to scale available resources such as computing power. According to Dustdar et al. [8] elasticity can refer to cost and quality as well.

Resource Elasticity

An elastic application needs to implement some auto-scaling behaviour for up- and down-scaling, depending on the fluctuation of the current workload. If the current scaling degree of an application is no longer able to handle the current workload, the application is underprovisioned. In contrast, an application that can meet a higher workload is called overprovisioned.

Cost Elasticity

Cloud providers might change the fees for instances depending on the current demand by their customers. If more customers require instances, prices might increase and the costs for an application will increase as well. For instance, Amazon EC2 offers spot instances which prices fluctuate over time and customers can bid for such spot instances.

If an application employs cost-changing instances it must be taken into account that resources can be more expensive during high-load periods than the bid of a customer. In such a case the application needs to scale down in order to meet the current cost requirements. This behaviour is called utility computing. [8]

Quality Elasticity

According to Dustdar et al. [8] quality elasticity refers to an application that offers higher quality when more resources are consumed. If customers require a better quality an application needs more resources which is usually more expensive. Depending on the offered service, quality can be measured in various ways such as the execution speed or the gained result. For example, offering a service that searches a huge data set can take an unacceptable long time if an exact result is required. In contrast, if a customer needs only a good result to some degree, the accuracy can be decrease and therefore the data set can be searched much faster and less resources are consumed.

CHAPTER 3

State of the Art

This chapter gives an introduction into some background knowledge and show the main areas of the state of the art that are related to this thesis. Testing of elastic systems is a key feature of this thesis where elastic systems should be tested locally. To understand the importance and the scope of this thesis, cloud uncertainties show the problems which might arise when an application should be executed in public cloud. Brownout-aware cloud-based applications is an approach to solve such cloud uncertainties. Lastly, related projects will be introduced.

3.1 Testing elastic systems

Gambi et al. [12] propose a novel approach for testing elastic computer systems. They use the metaphor of elastic materials to describe the behaviour of an elastic computer system. By some assumptions of testing an elastic material and to identify the crossover point where an elastic state of a material gets plastic due to the stress level, they want to identify states for the behaviour of an elastic computer system. Gambi et al. describe three testing approaches, namely Tensline, Impact and Fatigue testing. Based on this testing approaches Gambi et al. propose a conceptual framework that aims to support human testers for testing elastic computing systems. They propose a framework that consists of four test activities: test-case generation, test execution, data analysis and test evolution. Each of these activities has its own purpose and its own task. In general the activities are executed one after another in order to provide outcomes of one task to feed into the successor task. The human tester defines the test by a language that can be used to describe the elasticity. In order to get a deeper understanding of the effects of the elastic system under different configurations, test metrics and criteria will be provided.

In addition to the human support, Gambi et al. [10] suggest a novel approach to create test cases for elastic computing systems automatically. Their approach takes an initial test suite as input and refines the test cases automatically in order to find problematic workloads of an elastic system. In general, a human tester defines workload scenarios, such as a number of requests to an web application. The proposed approach takes this workload as an input model, applies

modifications to this workload model and executes the derived workload on an instance of the system under test. By validating application constraints the proposed approach can check if the system under test runs into a problematic workload and reports its finding to the human tester.

Gao et al. [14] introduce the concept of Testing as a Service (TaaS). They set the basement on the terminology about TaaS, describe various issues that arise when a TaaS should be offered and show some important advantages about TaaS. In order to actually offer a TaaS they argue that there's still some research required before their discussion can result in a production-ready product. By the formalisation of cloud computing and its testability, Chan et al. [5] want to present model-based testing criteria for cloud-based applications.

AUToCLES [11] as developed by Gambi et al. is a prototype that automatize live tests of cloud-based systems. It implements a Testing as a Service (TaaS) service, executes test suites automatically, and allows testers to inspect test results. In principle, AUToCLES offers a service that accepts applications to test together with a testing plan by a human tester and executes test cases in a cloud environment. It automatically scales depending on the current workload. The idea to create a Testing as a Service was that elastic systems share some common behaviour and therefore testing resources can be shared between various instances.

Their motivation to create such a system was as follows:

- Due to generalised system testing concepts an test framework can be optimised and designed on an abstract level.
- During a test run with different workloads some resources can be shared between the test runs.
- Test results are more comparable between different elastic systems if they use the same cloud and testing environment.

Based on these ideas AUToCLES offers to test elastic applications in a controlled environment. Only a single server is required to startup the system. Elastic applications to test are submitted by a human tester at the front end together with test inputs. During and after a test run various data are collected and provided to the test at the front end. The actual test run is executed by an elastic system that automatically employs multiple app servers and scales up and down as needed.

The approach of AUToCLES contradicts the assumptions of this thesis to test locally but its idea is to solve similar problems.

BonFIRE, as developed by Hume et al. [21], offers a large scale cloud testbed to test applications over multiple clouds. It offers a fine-grained control over available resources in order to simulate resource limitations. In general, it offers an infrastructure that allows testers to perform experiments of their elastic applications and to monitor these experiments. One of BonFIRE's key features is the availability of multiple geographically distributed clouds in order to perform experiments over multiple clouds. Variables such as CPU power, available memory and network

speed can be controlled by the tester. Metrics for CPU and I/O consumption enables a tester a detailed view of the behaviour of the system under test [21]. It's main feature is the ability to control various network variables such latency, speed and quality of service parameters. They can be a crucial culprit for distributed cloud applications. [37]

One of the architectural principles of BonFIRE is to be able to add resources during a test run which is not controlled by BonFIRE. This can be resources such as external storage or Amazon EC2 instances. This feature allows testing of private clouds with a bursting to public clouds. [25]

An experiment built by Gomez et al. [17] on top of the idea of BonFIRE shows the feasibility of the system. They're able to experimentally create a distributed infrastructure that takes applications and executes them within a given quality of service of network variables.

3.2 Cloud Uncertainties

Ou et al. [35] show that public cloud platforms might use various hardware platforms to provide their cloud services. Due to the different hardware behaviour, such as different CPU or memory speed, the virtual machines of a cloud are much different. Even if a customer rents two virtually same instances they might get different computing power due to the hardware heterogeneity. This performance discrepancy comes mainly from hardware diversity as well as different VM scheduling mechanism. Ou et al. employ a trial-and-better approach to keep only better performing machines and drop worse performing. They're are able to increase CPU performance up to 20 % and memory performance up to 268 % on the same instance type. If a customer uses this approach on Amazon EC2 they can achieve up to 30 % cost saving.

According to Pu et al. [40] there might be a significant interference to a virtual machine of the neighbours on the same physical host. Stress of the CPU or network interface can noticeable decrease the expected performance. They investigated into the behaviour of the XEN-driven virtual machine monitor in order to identify the scheduling behaviour if virtual machines consumes high CPU or network resources. During their experiment they conclude five observations, the three most important are introduced as follows:

1. Network-intensive applications force the physical host to a high number of context switches and results therefore in a high overhead.
2. By hosting applications with high CPU workload the available CPU power is under ongoing contention because of the fast I/O processing.
3. By the combination of the CPU-intensive and network-intensive applications on the same physical machine the overall performance can be increased due to the better resource allocation.

In addition to the available CPU power and network performance even the startup time of virtual machines can vary depending on certain criteria. Mao et al. [30] analysed tree different cloud providers and measured the startup time of various instance types, operating systems, image

sizes, regions and the time of the day. There's a difference in the startup time for all the analyzed criteria except the time of the day which seem to be constant.

Jayasinghe et al. [23] did a comprehensive comparison and benchmarking of various public cloud providers and private clouds using a hypervisor. They analysed the IaaS public clouds of Amazon EC2, Emulab and Open Cirrus as well as the open source hypervisors XEN and KVM and an unnamed commercial hypervisor that they call CVM. In order to get realistic and comparable test results they employed the benchmarking tools RUBBoS and CloudSone. During their comprehensive observation they performed more than 10,000 experiments. Therefore their results can be seen as a profound comparison for clouds when executing a controlled benchmarking software.

Their analysis shows significant differences between the public cloud providers. A good performing application might behave very bad on a different cloud. As expected, private clouds offer a better performance when compared to public clouds. Jayasinghe et al. conclude their findings that there is no cloud provider that offer a better performance in general. Each application has to be tested and analysed in the target environment, especially when performance becomes a critical bottleneck.

The CloudSim project describes itself as a "Framework For Modeling And Simulation Of Cloud Computing Infrastructures And Services". Its idea is to simulate clouds for various purposes, not only for testing scenarios. Additionally it offers to control the available resources of a simulated cloud which allows to create reproducible results for test runs. [3, 4]

Developers can employ the CloudSim project to identify bottlenecks in their applications before deploying them to a real cloud. Testing on private hosts, as offered by CloudSim, saves money and gives more control over available resources.

But CloudSim can be used by cloud providers as well. They can test various performance and pricing scenarios before offering a real cloud service.

3.3 Brownout-aware cloud-based application

Klein et al. [26] propose an approach for more fault-tolerant cloud-based applications. They take the idea of quality elasticity into a practical use case. When hardware fails on a public cloud they assume that the remaining machines need to take over the workload and might run into situations where the load for each single machine is more than it can handle. In such circumstances the cloud-based application will result in errors or timeouts which decreases the satisfiability of the users. In order to decrease the workload for each remaining instance Klein et al. propose to reduce the quality of the provided cloud-based application and therefore reduce the computation time needed to answer a request. They mentioned that an e-commerce application can skip to show product comments in order to keep the website online under heavy load.

Based on the same idea, by reducing the quality in order to keep a cloud-based application running, Klein et al. [27] developed a load-balancing algorithms that decides how many requests should be served with reduced content. This algorithm in conjunction with the brownout-aware

cloud-based application allows to decrease available resources that are kept for high-load spikes. If there occurs such a high-load spike the algorithm decides to reduce the quality in order to keep the response time in an acceptable waiting time. Elastic cloud-based applications can employ this novel approach to reduce costs.

Tomas et al. [43] show that resource overbooking in order to meet resource uncertainties in public clouds gets less important when using the novel brownout-aware load-balancing algorithm. It allows to gracefully decrease the served quality during peak times in order to avoid errors and timeouts. They show an increase in the resource utilisation of up to 37 percentage points when employing a brownout aware elastic cloud-based application.

A critical analysis of the state-of-the-art, it can be seen that there already exist approaches that deal with elastic systems and their ability to test such systems. According to Gambi et al. [12] testing elastic systems in a real cloud environment is necessary to provide realistic test results. Unfortunately, none of the already existing approaches offer to simulate cloud uncertainties. But such a cloud-near simulation is required to get realistic test results for best- and worst-case behaviour of cloud instances.

3.4 Critical Analysis

The critical analysis of the state-of-the-art leads to the following considerations for a testing framework:

- Tests should be executed locally. A real cloud does not offer to control the instances in such a way that various uncertainties can be simulated.
- To simulate cloud uncertainties the system has to be able to dynamically control the physical resource assignment to instances such as computing power or startup time. Out et al. [35] show that a cloud might get the following uncertainties: CPU time, memory speed, network and disk delay.
- The system has to offer control commands for cloud uncertainty test suites.

Unfortunately, none of the above named projects offer a feature set that meets these requirements.

CHAPTER 4

Methodology

4.1 Requirements

The task to underperform instances and to control resources as well as performance variability is the main idea of this thesis. Cloud providers don't guarantee resources, therefore the available cpu time and power can vary. An option to limit memory is required as well, because it is also limited in a real cloud environment.

Behavioural Expectations

In order to get a clear picture of the problem statement and the resulting expectations the following list introduces some high-level requirements.

- Verify that a System under Test (SuT) is able to start and stop instances.
- A System under Test increases the number of instances if the load increases. Check if the SuT starts the intended number of new instances.
- A System under Test increases the number of instances if the CPU power of the currently running instances decreases and therefore the workload cannot be handled anymore. Check if the System under Test starts enough new instances to handle the current workload.
- The available CPU power might increase. Check that a System under Test stops instances as expected in order to save money.
- Limit the amount of memory which is available for certain instances.
- The CPU power varies on a very high level. It changes randomly every couple of seconds between very low CPU power and full CPU power.

- Some cloud provider offer a feature called 'dynamic memory' where the available system memory is not fixed. Vary the available memory and verify if the System under Test behaves as expected.

Use Cases

Depending on the decision of section 4.2 to provide an API, there exists two use cases for a human tester to define test cases:

- Human testers can defines static behaviour before the actual tests starts. For example, the CPU time of the second node can be limited to 50 %.
- In contrast a human tester can use the API to interact with the developed system. A callback function will be provided that allows a tester to decide on demand when instances have to be started or stopped.

In addition to the definition of new test cases it might be suitable to adopt existing tests. This can be a huge time saver for the human tester if there already exists a comprehensive test suite that employs a real cloud provider for testing purposes.

Such a test suite needs to call the API of the cloud provider in order to control and verify if instances are started and stopped as expected. If an abstraction layer such as jClouds is used to decouple the test suite and the System under Test from a specific cloud provider, the test suite should be able to be executed on the developed system without changing significant parts of the code. In the case if the test suite uses jClouds it should be required only to change a configuration file.

Usually a test suite contains many tests that don't rely on a cloud and don't test any cloud related code. It might be the case that there are some test marked as integration test. The developed system should be able to execute only those tests on the container-based virtualisation environment.

By adding some behaviour definitions to the existing test suite it can be controlled how the simulated environment should behave. This allows a human tester a fine-grained control over available resources.

Scenarios to apply limitations

To be more concise in the following a few scenarios will be given that must be supported by the developed system. It must provide some selection criteria to identify nodes for applying resource limitations.

- all nodes
- nodes with a specific name (identified regular expression)

- nodes by number (the 5th)
- by time (all nodes that are started after 2 minutes)

Based on the above named simple criteria, one can build some quite complex rules. The following list gives an introduction what a human tester might have in mind. But the list is not complete, arbitrary combinations should be supported.

- Limit the available CPU time of all nodes to 10 % after the third node has been started.
- Limit the available CPU time of all nodes to 10 % after three minutes of test running.
- Limit the available CPU time of all nodes with a given name after two minutes of test run.
- Limit the available memory to 256 MB of all nodes when a node has been stopped.
- Limit the available CPU time for a nodes for two minutes from starting the node.
- Limit the available CPU time for the first two minutes after startup.
- Limit the available memory for the first two nodes with a given name for two minutes when the fifth node has been started.
- Limit the available CPU time of all nodes as long as a node with a given name is running, don't limit the available CPU time when the node has been stopped.

By analysing the fragments above it can be seen that a general approach for the test case definition is required. An approach where sections of the test case definition is written independently fits best. In detail, four individual parts are necessary:

Events such as a node has been started or stopped as well as a given time frame has passed.

Conditions that allow to decide that a rule gets fired only if a given node or a node with a given name is present.

Selection criteria to choose an arbitrary subset of all running nodes. A selection can be defined by the node number or a regular expression over the node name.

Action is the the actual resource limitation that should applied to the selected nodes.

By the combination the four segments one can build all scenarios as given above. In order to offer a wider flexibility a human tester should be able to create their very own test behaviour by implementing decisions on callback functions.

4.2 Design

The system consists of a simulation environment which allows to start an arbitrary number of instances of a given application. To control uncertainties it offers to limit resources and to restart the whole simulation process with different resource levels. In detail the system consists of three subsystems:

- At first, a subsystem starts the SUT and to deploy it to an arbitrary number of containers.

- A second subsystem controls uncertainties of the executed containers. It varies computing power and available memory as commanded by the test suite.
- A third subsystem gets the test results and inspects the behaviour of the SUT and compares it with the expected and defined behaviour.

Performance Variability/Unreliability

According to Ou et al. [35] a real cloud provider doesn't offer a fixed and guaranteed performance, but can vary in the available resources. In order to test an application under near-realistic conditions a tester needs to to be able to apply limitations in the available resources as they would happen in the real cloud environment.

A tester might want to apply a limitation only for an amount of time or only after the occurrence of certain events such as the third server has been started. Additionally one could combine these variabilities to comprehensive plans for simulating many different resource unreliabilities. There must exists some configuration options to support such requirements.

In order to focus on the main part of this thesis this requirement will be limited to only a small subset of all possible variations. Resource limitations can be applied/removed only under the following criteria:

- after a given time frame (e.g. limit the CPU after 2 minutes)
- after a given node has been started/stopped (e.g. limit the CPU for all nodes after the second has been started)

There will be an additional event selection criteria, such as the third node with a name pattern. The temporal limitations can only be applied to a time frame and the overall number of started/stopped nodes. There will be no "back to normal" commands, such behaviour can be modelled using a second command.

Managing multiple configuration methods

There are two different approaches which might be incorporated in the definition of the resource limitations: Annotations and a configuration file. Both approaches have their very own use cases:

Annotations can be used to quickly add limitations to single test cases or test classes. But annotations require to modify the source code. They allow a tester to easily add the functional test behaviour of the developed system to a test case.

Configuration files offer to add limitation to a whole bunch of test cases (e.g. to all tests in a package) without modifying source code. This approach can be used to run existing tests with different resource configurations. The limitations of a test run can be defined independently from the source code of the tests.

It might be the case that there exist annotations for a single test case and the test class as well as there are additional limitations written in the configuration file. Therefore it must be clearly defined how to configuration chain will look like and which option precedences of the others.

In order to keep this thesis simple, the highest priority will be set to the annotation approach. This option does not restrict the human tester but allows to keep the actual test case together with the configuration of the test behaviour closely together in a single file. The method level annotations will be precedented over the class level annotations, because they are more specific.

Integration with real testing frameworks

There exist many testing frameworks such as JUnit, TestNG, SureAssert and many more. It is fairly out of scope of this theses to provide an integration in many testing frameworks. To show the feasibility of this thesis it sufficient to focus on only one common testing framework. Due to the wide distribution and it's popularity JUnit will be chosen as the desired testing framework.

Obtaining test cases

Testing is an important task in modern software development techniques. There exists different approaches how reasonable test cases can be obtained. By developing a testing framework during this thesis it might be necessary to support various testing techniques.

New tests: A human tester creates new tests cases and uses an API to control the System under Test. This technique is necessary to test some behaviour on special circumstances, e.g. a very low performing instance. But it's a lot of work to create new tests cases and it might be too time consuming to test all relevant aspects.

Regression tests: Each update of the System under Test will be run on a Test Suite to ensure that no new errors have been introduced. Human testers manually test the System under Test, their actions will be recorded and replayed in order to verify that a System under Test behaves after code changes as expected and recorded. Regression tests can be very useful, because the tester defines tests only once and afterwards the tests are performed automatically. During test case recording the tester has to define assertions such as a maximum response time or a number of instances.

Reuse of existing tests: Existing tests can be used in the newly developed environment instead of a real cloud. Control instructions for the simulated cloud instances can be added as external resources such as a configuration file. Reuse of existing tests can be very helpful and time saving if there already exists a noteworthy tests suite for a real cloud. These tests should be adopted easily to execute them within a simulated cloud environment.

Mutation tests: The code in the cloud-based application will be controlled and modified automatically. Existing tests should fail when the code is modified. If there is a code modification without a failing test, it's an indicator that a human tester should add additional test cases. Mutation tests might be helpful to identify if the test suite covers all different

cases of the System under Test. Usually, there are many cases which are not relevant to be tested in a cloud environment such as small changes in the available resources.

The "new test" approach is the most advanced and most flexible way to employ a cloud testing environment. These kind of tests cases should be offered and implemented during the thesis.

Test interface

A test interface is a very abstract definition of how the human tester defines the actual test cases. There might exist different possibilities:

- API (e. g. JUnit tests)
- Domain Specific Language (DSL)
- Web Interface
- Configuration File
- Java Annotations

It depends on the decision how the tests will be obtained in order to decide which testing interface fits best to be used by a human tester. According to the previous section, "new tests" should be supported. Therefore an API is the most suitable solution.

Additionally it depends on the skills of the tester which interface should be offered. A java programmer should be able to work with ease with a Java API, whereas testers with other background would prefer a simpler solution. A webinterface or a desktop client that is accessible with a mouse fits better for non-java-developers. In contrast developing a webinterface is out of scope of this thesis.

If there are already cloud tests available which should be executed, another interface is required. An external configuration which resides next to the actual without the need to modify the tests fits best.

A Domain Specific Language (DSL) can be used in addition to an API. But it doesn't offer a different interface than the API, it's just an easier and more intuitive way for developers than using the API. There is no need to implement a DSL.

Annotations are a different way to specify behaviour without modifying the tests itself. In contrast to a configuration file they still required to modify the java test file, therefore future changes in the test suite might result in the need to re-add annotations. Annotations might be useful for developers to specify the actual test behaviour with java code and add information about limitations using annotations without polluting the test code. But annotations cannot be that flexible like calling an API, because API calls can be react on events from the test run.

In conclusion can be said, that for the purpose of this thesis, the implementation of an API using JUnit fits best.

Long running tests / Execution time

The actual implementation strongly depends on the execution of a test suite. If it contains tests cases that run for a long time (e. g. more than a few minutes) the test environment should be executable independently from the development environment (e. g. don't use eclipse to start tests). It might be cumbersome if the developer has to start long running test suites and to keep this process running while he wants to continue developing. If there are such long-running test suites they have to be executed on a separate test execution server or a specialised test environment.

But the more a test run is long, the less is obvious why one should opt for the Docker approach other than controlling the state of the cloud. The docker approach is useful to simulate short circumstances such as to control that the cloud increase the number of servers as the load increases within a given time frame. Furthermore the scope of this thesis is to get fast test feedback by employing the container-based virtualisation. But long running tests contradict to the idea of fast test feedback.

To support the developing process and to run comprehensive test suites a continuous integration (CI) system can be used. Usually, such a CI pulls code from a repository regularly, executes the test suite and notifies developers if any tests fail. It might be feasible to create a test suite for cloud tests and to employ a CI to execute the test suites. This approach offers to execute long-running test suites regularly without interrupting a developer's workflow.

CHAPTER 5

Solution Design

The developed system offers to retrieve test cases including resources limitations for a System under Test. In order to simulate cloud uncertainties the test cases can respond to events such as newly started nodes.

5.1 System Overview

The developed system consists of several components. These components are loosely coupled in order to exchange them easily. In detail, the prototype consists of the following components:

- Interceptor to gather commands from the System under Test to the cloud.
- Rules Engine to decide if there has a limitation to be performed.
- Interface to send limitation commands.
- A main routing component to connect all these components.

These components and their interaction can be seen in Figure 5.1. Besides the above named components, there exists the human tester who defines test cases, the System under Test and the Docker Host that executes the System under Test.

Using the system works as follows: A tester defines several test cases. They describe how the System under Test (SuT) should behave under certain circumstances. Such a test case definition can contain rules to limit resources such as CPU time and memory. Rules can be applied from the beginning, after a given time or when a given node has been stared or stopped. Additionally, a tester can directly interact with the System under Test.

Usually, when a System connects to the Docker API it directly sends commands to the REST interface as provided by the Docker host. In order to allow the interceptor to get notified about

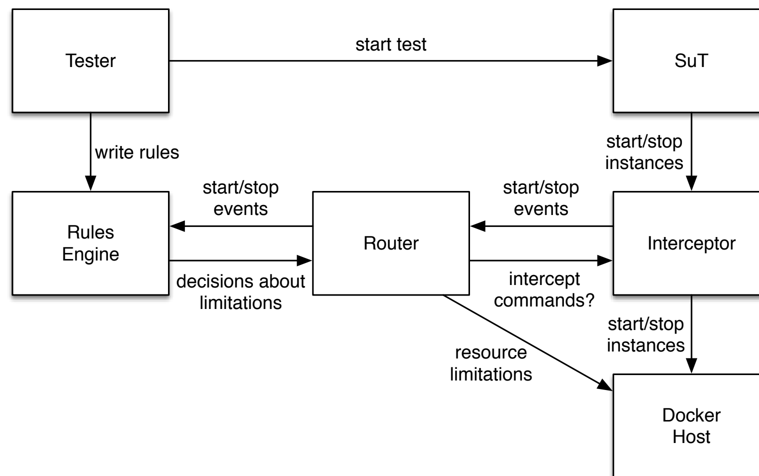

Figure 5.1: System Overview

started or stopped instances a HTTP proxy is provided so that a System under Test does not need to change its behaviour. It's just required to connect to a different host or port and the proxy will forward requests and responses transparently to the actual Docker Host.

But there are Frameworks such as jClouds that allow a System to connect to different public cloud provider without adjusting the application to the specialised API of the Provider. jClouds abstracts the APIs and offers common behaviour in order to connect to different provider. The idea of this thesis is that such an application just needs to change their configuration in order to connect to Docker, respectively the proxy, to use the developed system.

5.2 High level architecture

To get a better understanding the internal behaviour of the overall system Figure 5.2 depicts in the internal message exchange and invocation order. The numbering in the following list corresponds to the numbers in Figure 5.2.

1. The testing process starts by the insertion of rules as defined by the Tester in a Test Case. A rule gets fired by events such as a newly started or stopped node as well as a given time frame. When a Rule gets fired it can create a command to adjust system resources such as CPU power or available memory.
2. Secondly the test case itself gets executed. Usually it starts the System under Test (SuT).
3. It depends on the System under Test what needs to be done during startup, but starting some nodes might be a common behaviour.
4. Under the assumption that a rule has been set to capture node start events the Interceptor recognises the command to start a new node.

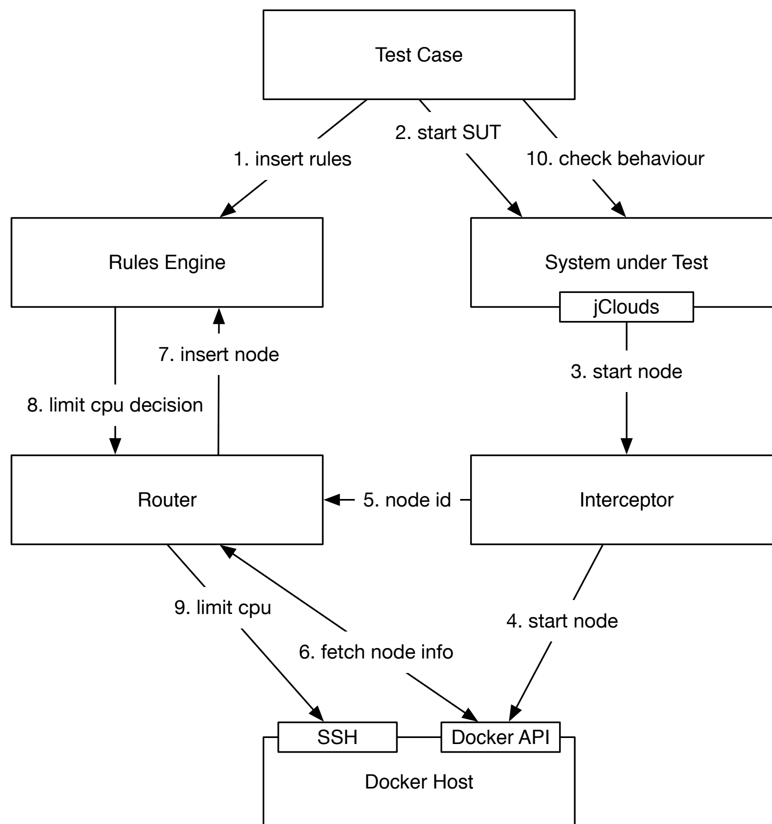

Figure 5.2: Invocation overview

5. In order to create the internal node started event the Interceptor forwards the Node id to the Router.

6. The Router uses the node id to retrieve additional information from the exuting host about the newly started node. This information contain the actual node name as set by the System under Test and the automatically assigned IP address. The node name can be used by a rule definition to decide if a rule needs to be fired.

7. After the Router gathered information to build the full node information it inserts it into the Rule engine in order to decide if a rule matches the newly started node.

8. Under the assumption that there exists a rule to limit the CPU power of the newly started node, the rule engine sends its decision to the router.

9. The Router connects to the execution host in order to actually limit the available CPU time of the started node.

10. Lastly, the Test Case checks if the System under Test behaves as expected with the reduced computing time.

The above description covers only a subset of all available information flows. But it introduces some common behaviour in order give an overview of the overall system.

In order to get a better understanding of the interaction between the various component Figure 5.3 provides an abstract sequence diagram that shows the behaviour of the system in general.

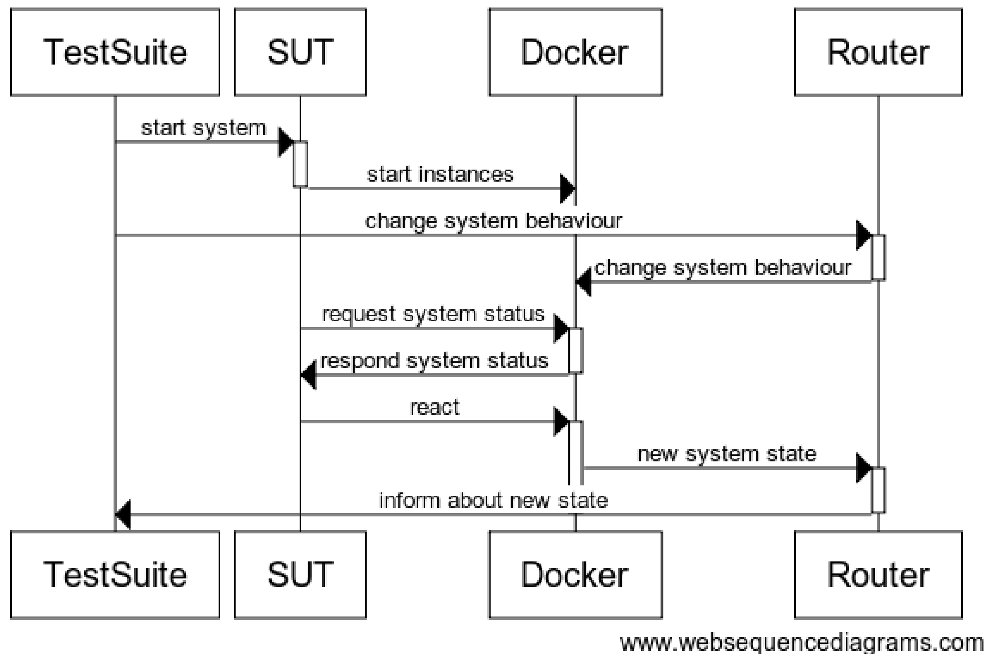

Figure 5.3: General sequence diagram

It can be seen that most of the calls are sent from the System under Test to the Docker Host. Usually a System under Test would not use the Docker API directly but using an abstraction framework such as jClouds. There exists never a communication between the System under Test and the Prototype. The use of the Prototype for test purposes is a transparent operation to the System under Test.

5.3 Detailed Architecture

Interceptor

The Interceptor (Figure 5.4) acts as a proxy that retrieves commands that are sent to the Docker API by the System under Test. This information is required to monitor the System under Test such as respond to started or stopped nodes. The interceptor listens on a given port on the local machine and forwards the whole HTTP traffic to a configurable host and port, e.g. the actual Docker host. The System under Test must be configured to connect to the interceptor port instead of the actual Docker Host.

In order to recognise operations such as starting new Docker instances a pattern matching is applied to each request. In case that a matching pattern has been found, the information about the started node will be forwarded to the router.

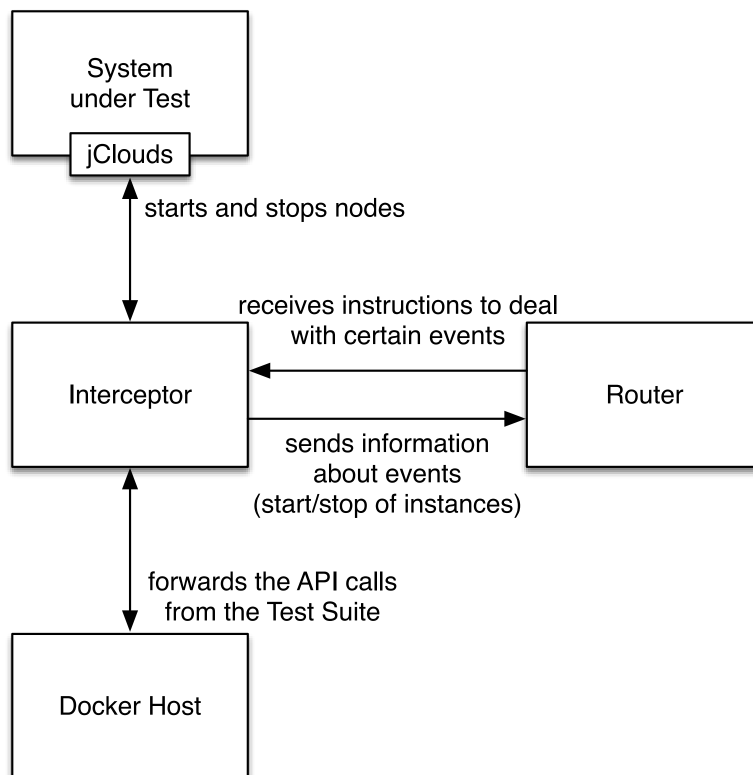

Figure 5.4: Detail view of the Interceptor

Docker API Connector

In order to retrieve additional information about a started node the Docker API has to be queried. A RESTful interface is provided by Docker that allows to get various information. The developed system requests the name and the automatically assigned IP address of the newly started node. Together with the node id internal data structures will be created. Various operations such as decisions about resource limitations or node selections of the RuleEngine rely on the information of the DockerNode java instance.

Rule Engine

A Rule Engine is a computation model that allows to check if certain criteria a fulfilled for some input variables, and if so, to execute some code. This behaviour definition is called a Rule. There is no control flows defined by rule [18]. Usually, a rule engine allows to insert multiple rules and to fire them, e.g. to execute them if they are fulfilled. The advantage to use a rule engine is to

clean up code and to encapsulate the business logic in rules, whereas the actual execution code can be held separated. [9]

The Rule Engine as used in this system stores selection criteria and resulting resource limitations. These criteria and resource limitations decisions are inserted by the test case definition. The Rule Engine will be triggered by events such as when a new node gets started or timing events that may also be happen on fixed intervals. Every time the Rule Engine gets triggered it selects a number of nodes and decides about resource limitations. These decisions get forwarded to the router.

Router

The router is the key component of the developed system. It connects all other components and controls the information flow of the whole system. It does not interact with any external interface.

To get the router running a configuration must be provided that specifies information about the Docker host such as the IP address, the port of the Docker RESTful API and ssh credentials.

5.4 Test Case Definition

As already mentioned a test can define the availability of resources such as the cpu time. An interface is provided that offers to define a behaviour of resource limitations. To be more concise this feature allows to limit the cpu time or available memory when a node has been stopped or started or after a given time frame.

DividedRule annotation

In detail, this behaviour is defined as annotations at the test case header. Listing 5.1 shows a simple definition how such annotation might look like. The concrete example limits the cpu time to 20 % of all nodes which name starts with 'web' after 120 seconds of test run.

```
@Test
@DividedRule(trigger   = NodeStartedTrigger.class,
             condition = TimePassed.class,    conditionParams = "120",
             selection = NodesWithName.class, selectionParams = "web.*",
             action    = LimitCpu.class,      actionParams = "20")
public void test() {
  ...
}
```

Listing 5.1: Simple example of a test case definition.

The DividedRule annotation is provided by the developed system and allows to define a behaviour rule in four distinguished parts:

Trigger: Specifies when a rule will be triggered by the RuleEngine. Currently implemented are NodeStartedEvent, NodeStoppedEvent and a TimerEvent.

Condition: An additional condition that checks if the rule will be applied.

Selection: The selection criteria defines a subset of all running nodes that receive a ressource limitation.

Action: The actual resource limitation that will be applied to the selected nodes.

Additionally to the above list, Listing 5.1 shows conditionParams, selectionParams and actionParams. These params will be forwarded to the corresponding parts of the DividedRule. Due to limitations of the Annotation semantic in Java no instances can be provided but only classes. Therefore the RuleEngine will create instances and forwards the params as values to the class constructor.

Timer annotation

It might be the case that a tester wants to limit resources or remove resource limitations after a fixed time. The timer annotation offers to trigger a TimerEvent in order to apply resource limitations after a fixed delay, or apply changes repeatingly.

```
@Test
@Timer(name="2min", initialDelay=120, timeUnit=TimeUnit.SECONDS)
@DividedRule(trigger   = TimerEventTrigger.class, triggerParams = "2min",
             condition = NoneCondition.class,
             selection = NodesWithName.class,     selectionParams = "web.*",
             action    = LimitCpu.class,          actionParams = "20")
public void test() {
  ...
}
```

Listing 5.2: Example of a Timer annotation.

Listing 5.2 shows an example how to use the Timer annotation. It sets a timer that gets fired after two minutes. To catch the fired event a DividedRule annotation is used that responds the event and limits the cpu time to 20 % for all nodes which name starts with 'web'.

Additionally the Timer annotation allows to set a recurring time so that a timer gets fired repeatingly. If required a recurring timer can be ended by setting a duration time.

RuleClass annotation

If the above DividedRule and Timer rules don't fit the tester's needs, they can define their very own rule behaviour by using the RuleClass annotation. The use of this annotation is shown in Listing 5.3 whereas the implementation can be seen in Listing 5.4.

The onEvent method is called every time an event happens. Section 5.5 introduces the different event types that can occur.

There can be used as many rules as required for a single test case. Additionally, the provided DividedRule, Timer and RuleClass annotations can be mixed arbitrarily.

```
@Test
@RuleClass(ExampleRuleClass.class)
public void test() {
  ...
}
```

Listing 5.3: Example of a custom RuleClass annotation.

```
public class ExampleRuleClass implements Rule {

  @Override
  public void onEvent(Event event, List<DockerNode> nodes) {
    ...
  }
}
```

Listing 5.4: Example of a custom RuleClass implementation.

```
@Test
@RuleClass(ExampleRuleClass.class)
@Timer(name="2min", initialDelay=120)
@DividedRule(trigger   = NodeStartedTrigger.class,
             condition = NoneCondition.class,
             selection = LatestNode.class,
             action    = LimitCpu.class, actionParams = "20")
public void test() { }
```

Listing 5.5: Example of mixing annotations.

5.5 Technical Details

Interception of the Docker API

The Docker API exposes a HTTP REST interface that offers to perform various actions such as to start or stop nodes. In order to respond to such events the developed system needs to know when such an event happens. This feature is provided by the Interceptor component. The idea is to listen for incoming HTTP connections and forward all traffic to the actual Docker Host and forward the responses back to the caller. The code snippet of Listing 5.6 shows how the interceptor component opens a socket and copies the traffic. The actual caller of Docker API just needs to connect to a different port or host so that the developed system retrieves the traffic to the Docker API. There is no change in the behaviour of the Docker API, all requests and responses are proxied without modification. The interception happens in line 4 where all listeners get notified about the new connection.

There can be defined various listeners to analyse the traffic to the Docker API. Listing 5.7 shows how the traffic of a command to start a new node gets identified. A regular expression matches all HTTP POST commands for the corresponding endpoint. If such an event has been identified, the Rule Engine will be notified about this event.

```
Socket clientConnection = new Socket(dockerHost, dockerPort);

List<DockerApiListener> listener = new ArrayList<>();
listenerFactories.forEach(lf -> listener.add(lf.onNewConnection()));

Consumer<String> onRequest  =
    (String request)  -> listener.forEach(l -> l.onRequest(request));
Consumer<String> onResponse =
    (String response) -> listener.forEach(l -> l.onResponse(response));

new CopyThread(serverConnection, clientConnection, onRequest).start();
new CopyThread(clientConnection, serverConnection, onResponse).start();
```

Listing 5.6: The Interceptor component copies traffic from and to the Docker API

```
public class ContainerStartListener implements DockerApiListener {

  private Pattern pattern;
  private RulesController rules;

  public ContainerStartListener(RulesController rules) {
    this.rules = rules;
    this.pattern = Pattern.compile(".*POST /containers/(\\w+)/start.*",
        CASE_INSENSITIVE+DOTALL);
  }

  @Override
  public void onRequest(String request) {
    Matcher matcher = pattern.matcher(request);
    if (matcher.matches()) {
      String id = matcher.group(1);
      rules.nodeStarted(id);
    }
  }
}
```

Listing 5.7: ContainerStartListener.class intercepts calls to the Docker API

Events

In order to respond to certain external occurrences and to distinguish between different types, a class hierarchy has been created to work more easily with such events. In general, there exists two types of events: events of a started or stopped nodes and events that are fired by a timer. In the first case the Event gets fired by the Interceptor component. A timer event is needed to notify the rule engine about the expiration of a timer as defined by a human tester in the test case definition. There exists no significant code that could be of interest, the Event classes are mostly only class definitions.

Rules

A rule is simple interface (Listing 5.8) that resembles the behaviour of certain event responses. It requires to implement a method to respond to events and to be able to set a ResourceController that offers access to the resource limitation.

```
public interface Rule {

  public void onEvent(Event event, List<DockerNode> nodes);
  public void setResourceController(ResourceController resourceController);
}
```

Listing 5.8: The Rule interface

Section 5.4 introduces how tester can define Rules. In order to take away pressure from a human tester a default implementation is provided. This DividedRule class offers a definition of a Rule in four steps:

- A Trigger that allows to define the Events that should fire the rule.
- The Condition that checks a criteria such as that at least 10 nodes must be running.
- In order to select only a subset of all running nodes a Selection criteria can be used.
- For the definition of the resource limitation an Action is provided.

```
public class DividedRuleImpl implements Rule {

  private Trigger trigger;
  private Condition condition;
  private Selection selection;
  private Action action;

  public DividedRuleImpl(Trigger trigger, Condition condition, Selection selection,
      Action action) {
    this.trigger = trigger;
    this.condition = condition;
    this.selection = selection;
    this.action = action;
  }

  public void onEvent(Event event, List<DockerNode> nodes) {
    if(trigger.test(event) && condition.test(nodes))
      selection.apply(nodes).forEach(action);
  }

  @Override
  public void setResourceController(ResourceController resourceController) {
    action.setResourceController(resourceController);
  }
}
```

Listing 5.9: The implementation of the DividedRule class.

Listing 5.9 shows the implementation of the DividedRule class. It can be seen that most of the class is responsible just for storing attributes. The actual execution of the Rule happens in the onEvent method where modern Lambda expressions are used. The Trigger and Condition parameters are Predicates, the Selection is implemented as a Function and the Action extends a Consumer.

Rule Engine

All Rules are stored in the Rule Engine and called if an event occurs. The Rule Engine is responsible to keep an instance of all Rules, to fire Timer events and to notify all rules about events. Additionally it keeps track of all running nodes so that rules an can select only a subset of running nodes to apply their resource limitation. Listing 5.10 shows the crucial implementation of the Rule Engine. This fire method is called by the Interceptor if a node event has been identified or a timing event occurs.

```
public void fire(Event event) {
  if(event instanceof NodeStartedEvent)
    nodes.add(((NodeStartedEvent) event).getNode());
  else if(event instanceof NodeStoppedEvent)
    nodes.remove(((NodeStoppedEvent) event).getNode());

  rules.forEach(r -> r.onEvent(event, new ArrayList<>(nodes)));
}
```

Listing 5.10: The main implementation of the Rule Engine

Resource Limitation

The actual resource limitation of the Docker Host is provided by the cgroup feature of the Linux Kernel [36]. Each Docker node is encapsulated in its own cgroup environment. To set a resource limitation one needs to write to a file on the Docker host. Whenever a resource limitation should be applied the developed system opens a SSH connection to the Docker host and writes the value to a desired file. In case of a cpu time limitation to 20 % for the Docker node with the id 1234 the value 204 will be written to `/sys/fs/cgroup/cpu/docker/1234/cpu.shares`. The calculated value 204 is computed as the 20 % share of an unlimited cpu time of 1024. Memory limitation values are expected in bytes, there is no calculation necessary [34]. In order to connect to the Docker API a SSH connection is used to get access and write the values to the files. Listing 5.11 shows the method to actually set a new CPU limit.

```
public void limitCpu(DockerNode node, int percentage) {
  int cpuShare = 1024 * percentage / 100;
  sshConnection.writeContentToFile(
    Integer.toString(cpuShare), buildDockerCgroupFilePath("cpu", node.getId(), "cpu.
        shares"));
}
```

Listing 5.11: Setting a new CPU limit at the Docker host

CHAPTER 6

Evaluation

The main idea of this thesis is to provide a way to test the robustness of a elastic policy by providing more control over available resources than a real cloud can offer to testers. In order to achieve this task a system has been developed that employs container-based virtualisation to execute the elastic application and offers fine-grade control over computing resources.

The prototype can simulate various background noise to allow a tester to test how the system behaves if resources are limited and fluctuating. It allows to define the behaviour of the background in a declarative way by means of rules subjected to fluctuations so testers can simulate different behaviours for each test case.

This chapter shows that the developed prototype does not add a significant performance overhead to the system under test which might be able to alter test results. Additionally two use cases will be provided so that the reader will understand the gained improvements.

6.1 Feasibility

Functionality

The developed framework employs Docker to run elastic applications locally. Docker relies on the cgroup feature provided by the Linux Kernel under the hood. That encapsulates processes from each other and allows to concise resources of this processes.

Because of the mode of operation of the cgroup feature the developed framework can simulate various cloud behaviours. The system under test will be executed directly on the cgroup process encapsulation, there exists no additional layer introduced by the developed framework.

It can therefore be said that the developed framework cannot be responsible for introducing new and unexpcected errors in the software stack. It only controls the cgroup feature but cannot directly interact with the System under Test.

Performance

The overall performance of the system is a key metric in order to show the feasibility of such a cloud simulator. On the one hand, the prototype catches events such as a node has been started or stopped or timer events as specified by the tester. Based on these events, decisions are made about limitations to apply. The crucial performance limitation of the prototype itself is the number of events that can be caught in a given time frame and the number of limitation decision that have to be applied.

On the other hand, each running node takes some resources of the host machine, i. e. cpu time and memory. But the intended idea of this prototype is not to simulate many running instances of an application in parallel. Additionally, it strongly depends on the application under test itself how many instances can be executed in parallel on a single host machine. Therefore a general overview cannot be provided and testers have to take care about the available resources of their host machine and the required resources of an application under test.

In the following, some tests have been performed in order to give an overview about the number of events that the prototype can process. It must be said, that all these test results strongly depend on the computing power of the hardware where the tests have been executed.

In order to distinguish between the external performance impacts and the performance of the frameworks itself, different tests have been executed. The first test measures the executing time of key components of the developed framework. Secondly and thirdly, the external Docker API and cgroup calls to limit resources have been tested and measured. Only by measuring the executing time of the framework itself combined with the external dependencies it can be said if there exists a performance impact of the system under test when simulating and changing the cloud behaviour using the framework.

Internal tests on the Rule Engine
This test gives an overview how long it took for the framework from the event of a new node until the decision about resource limitations. The time measured in this tests shows the duration from a new node event that has been captured until the rule engine performs its decision based on pre-defined rules and currently running nodes.

Before a test has been performed the given number of rules have been inserted into the rule engine. Additionally, the rule engine contains a list of nodes that would have been started already, whereas this tests just contains a list of objects that is used by the rules.

The tests have been performed 500 times, table 6.1 shows the average (standard deviation) for instances of rule engines that contain 10, 20, 50 and 100 rules as well as nodes.

When increasing the number of rules but keeping the number of nodes constant, the system gets a linear increase in executing time. It can be seen that when increasing the number of nodes or the number of rules the time to execute the rule engine increase linearly.

Calls to the Docker API
The framework needs to call the Docker API to fetch additional information such as the node

		nodes			
		10	20	50	100
rules	10	0.05 (0.03)	0.09 (0.05)	0.29 (0.32)	0.54 (0.29)
	20	0.10 (0.06)	0.21 (0.09)	0.54 (0.29)	1.02 (0.44)
	50	0.25 (0.11)	0.52 (0.17)	1.24 (0.28)	2.49 (0.48)
	100	0.54 (0.18)	1.05 (0.26)	2.55 (0.60)	4.99 (0.72)

Table 6.1: Average time in ms to process an event (standard deviation)

name or the IP address for each started node. When performing 1000 calls to the Docker API it took in average 6.27 ms (3.96 standard deviation) on the reference hardware.

Commands to limit resources
Depending on the available rules after performing events by the rule engine, there might be a decision to limit resources such as CPU or memory for a given node. By performing a test with 1000 commands to limit the CPU for a Docker node it took 31.82 ms in average (standard deviation of 7.06) on the reference hardware.

Conclusion
As can be seen in table 6.2 by summing up all the above given numbers it took in average 38.14 ms to receive an event, make a decision and command a resource limitation. In other words, the framework is able to catch up to 26 events per second, retrieve additional information from the host and send commands to limit resources. This huge number of new nodes can hardly be achieved during a cloud simulation task. It can be said, that using this framework there will be just a very little performance overhead and there are no performance limitations that can be reached during a real test run where the amount of rules nodes are much smaller. The performance overhead by using this framework is negligible.

	Average	Standard Deviation
Rule Engine	0.05	0.03
Docker API	31.82	7.06
Cgroup command	6.27	3.96
Summary	38.14	

Table 6.2: Time in ms for the framework to process a single event.

6.2 Case Study

In this chapter some experiments have been performed in order to show the feasibility and the improvements by using the developed framework.

For these experiments a very simple elastic application has been developed. This application offers a web interface to upload an image, it rotates the image by 180° and shows the resulting

image to the user. Depending on an elastic policy the applications scales out and in, e.g. it expands itself to more servers or shrinks by leaving servers.

Testing system behaviour

This experiment uses a fixed number of four nodes without any elastic policy. Requests are distributed randomly to the available nodes. During the whole experiment there have been ten requests in parallel. As soon as the application finishes to serve a response, the test creates a new request.

The experiments starts with four nodes without any limitation for the nodes. Chart 6.1 shows the number of requests that are currently executed on each node. As can be seen in the beginning the incoming requests are roughly distributed equally to the available nodes.

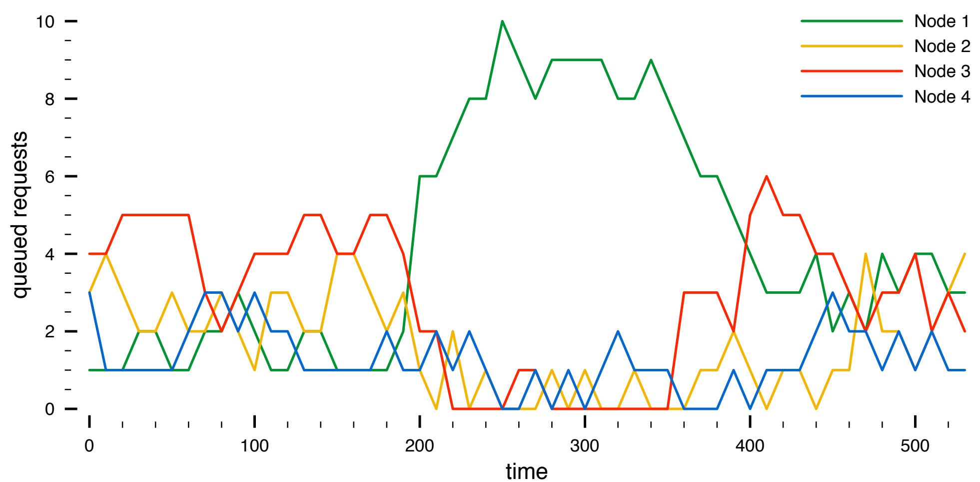

Figure 6.1: Unbalanced load distribution by applying a CPU limitation

After 3 minutes the first node gets a CPU limitation of 50 %. This means the running node was not be able anymore to get more than 50 % of the CPU time as it would get without any limitation. Due to this resource limitation it takes much longer for the first node to respond to a request than it takes for the remaining three nodes. Because of the random request distribution the first node gets almost the same number of request to serve as nodes 1 to 3. This imbalance results in a very long queue for the first node. In the end almost all requests are queued on the first node.

The CPU limitation of the first node has been removed after 3 minutes. This resulted again in a roughly balanced load distribution over all nodes. As can be seen in this experiment, there can be a huge impact of the overall application performance when the load balancing doesn't work as expected.

Test an elastic policy

The Framework allows to test if an elastic policy works as expected by the human tester. There is no need anymore to get instances of a real cloud for such simple test cases.

In the following an experiment shows the influence of a CPU limitation on an elastic policy. The same application as described in section 6.2 will be used. The elastic policy is very simple: A new server has been started as soon as there is at least one server with more than three waiting requests. If there are no more than two waiting requests for each server, a server has been stopped.

To simulate an increasing load over time the experiment started 10 requests per 10 seconds. Each minute the number of requests per 10 seconds has been increased by one until there are 15 requests every 10 seconds. After reaching the maximum number of 15 requests, the load has been decreases until it returned to 10 requests per 10 seconds.

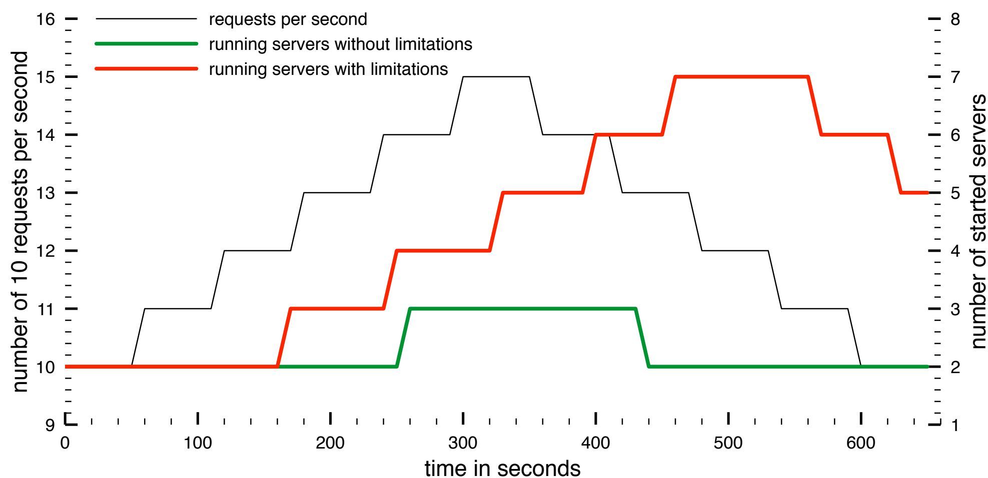

Figure 6.2: Testing an elastic policy locally.

In order to show the difference of the behaviour of the elastic policy without limitation and by applying a limitation two runs have been performed.

Figure 6.2 gives an overview about the results of the experiments. The black line shows the number of requests that have been performed. As already mentioned the the first experiment doesn't introduce any limitation and the system under test behaved as expected. It started a new node when the load reached a limit and stopped a node when it was no longer required. The results in the number of running nodes can be seen in Figure 6.2 as the green line. The number of servers increases roughly equally to the number of parallel request. If the load decreased, the elastic policy stopped unused server reliable.

In contrast, by applying the simple CPU limitation on a single node the elastic policy doesn't work as expected anymore. In Figure 6.2 the red line shows the number of nodes that have been

started with the same load fluctuation and the same elastic policy as described in the previous experiment. But the CPU time of the first node is limited to 50 %. This results in a longer response time of the first node, but due to the round robin load distribution it gets the same number of requests to serve. As can be seen in the chart, the elastic policy behaved as expected while the load increased. But even when the load reached the maximum number of 15 requests per 10 seconds, the elastic policy still started new nodes. Even worse, while the load decreased, there have been new nodes started. This resulted in a much higher number of nodes than actually required to serve the requests. In the end, the elastic policy started up to 7 nodes, whereas just 4 nodes would be enough to serve the incoming requests.

By comparing the green and the red line Figure 6.2 the difference is significant and might be unexpected to the developer of the elastic policy. Such unexpected behaviour of the elastic policy must be tested if resources cannot be guaranteed by cloud provider.

6.3 Financial and organisational improvements

Cheaper test runs

The need to test an elastic policy should be considered to the same degree as testing any other code. Usually, testing an elastic policy requires a test environment that is very close to the actual production environment. This means only for testing the elastic policy, one need to run the policy in a real cloud environment. In many cases such a real cloud environment is not available for free, but costs money. Therefore each test run of the elastic policy can be be really expense.

By using the developed framework const-intensive tests of an elastic policy in a real cloud environment can be reduced. Tests can be performed on the local machine of the developer or tester. Usually, consuming computing power on a local machine is free of charge. In the end, testing an elastic policy locally is much cheaper than by employing a real cloud environment.

Improvements for the Tester

By implementing the developed framework into a test lifecycle one might get faster test feedback. There is no need anymore to start the whole operating system for each virtual machine. Docker allows to use the same operating system instance and just start the system under test itself in a dedicated process. The delay of starting new operating system instances in contrast to new processes is much shorter. Therefore when starting the system under test using process virtualisation the tester gets much faster feedback for the executed test.

Additionally, cloud providers offer resources on best effort. There is no way to limit the resources to a lower level than are actually available. But such a man-made drop in the resources might be helpful to test a system under certain environments. It must be tested if a system under test behaves as expected even if resource are not available as promised by the cloud provider.

CHAPTER 7

Conclusions

Testing elastic policies is a crucial task for the development of elastic applications. The main idea of this thesis is to provide a way to test the robustness of an elastic policy with more control over available resources than a real cloud can offer.

As shown during the thesis there exist many scenarios of an elastic policy that should be tested in order to minimise unexpected behaviour of the elastic application. This thesis proposed a system that allows to test elastic policies locally by employing container-based virtualisation. Testing locally gets faster results and is cheaper than using a real cloud environment. Additionally the developed system allows to limit available CPU and memory in order to simulate resource uncertainties that might occur when a public cloud should be used.

In order to simulate cloud uncertainties the developed system offers to limit the CPU time and available memory individually for each running application instance. The available resources can be changed based on certain events such as a newly started application instance or after a given time frame. An API offers full access to the resource control actions.

As shown in the thesis the developed system can be used to identify issues in the elastic policy. There exist use cases that occur only under strong resource stress, therefore such issues might be overseen when resources are sufficient.

The developed system improves the testing process of elastic policies by faster test results, cheaper test runs and more control over available resources.

CHAPTER 8

Future Work

8.1 Extend resource limitations

As described in the thesis there are limitations only the most common resource bottlenecks, CPU time and memory, are provided. There exists various studies [30], [23], [35] that show performance variations in public clouds. The authors try to identify the equality of virtual instances within and between various Infrastructure as a Service providers. According to them there might exist many other performance bottlenecks that one might run into:

- memory speed
- network speed and latency
- disk speed and latency
- number of I/O requests per second
- process switching rate
- startup time
- startup reliability

But even if the developed framework has been extended with additional limitations, there might still be many other performance bottlenecks. The performance of a single virtual machine strongly depends on the bounds as set by the cloud provider as well as the the current workload of all other virtual machines on the same physical host.

8.2 Test framework integration

The developed prototype during this thesis provides an integration to JUnit only. But one might prefer other testing framework such as TestNG. Even non-java based testing frameworks such as RSpec can be of interest if the system under test has been developed with Ruby. The developed prototype aims to be loosely coupled, therefore creating further integrations should be feasible.

8.3 Support for Continuous Integration

Even though the developed system should be able to be integrated into a continuous integration (CI) basically, there might be some special requirements for the use within a continuous integration environment.

A CI system might be executed on more than one machine and it can be useful that the tests within a single test run are distributed within available CI servers. This can shorten the time to finish a test run as well as increase the degree of capacity utilisation of the available CI servers.

Additionally test might require a lot of resources, therefore a tester should be able to define that certain tests should be executed exclusively on a CI server to make sure it can take as much resources as required.

Bibliography

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. 2009.

[2] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI*, 2006.

[3] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo Calheiros. Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities. *Proceedings of the 7th High Performance Computing and Simulation Conference*, 2009.

[4] Rodrigo Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar De Rose, and Rajkumar Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 2011.

[5] W. K. Chan, Lijun Mei, and Zhenyu Zhang. Modeling and Testing of Cloud Applications. *IEEE APSCC*, 2009.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. *OSDI*, 2006.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[8] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of Elastic Processes. *IEEE Internet Computing*, 2011.

[9] Martin Fowler. *Domain Specific Languages*. Addison Wesley Signature Series, 2010.

[10] Alessio Gambi, Antonio Filieri, and Schahram Dustdar. Iterative Test Suites Refinement for Elastic Computing Systems. *European Software Engineering Conference*, 2013.

[11] Alessio Gambi, Waldemar Hummer, and Schahram Dustdar. Automated Testing of Cloud-Based Elastic Systems with AUToCLES. *IEEE/ACM International Conference on Automated Software Engineering*, 2013.

[12] Alessio Gambi, Waldemar Hummer, Hong-Linh Truong, and Schahram Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 2013.

[13] Alessio Gambi, Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. On Estimating Actuation Delays in Elastic Computing Systems. *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2013.

[14] Jerry Gao, Xiaoying Bai, and Wei-Tek Tsai. Cloud Testing- Issues, Challenges, Needs and Practice. *Software Engineering: An International Journal*, 2011.

[15] Jacques Gelinas. Howto/FAQ project vserver. http://www.solucorp.qc.ca/changes.hc?projet=vserver, 2001.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SOSP*, 2003.

[17] A. Gómez, L.M. Carril, R. Valin, J.C. Mouriño, and C. Cotelo. Experimenting Virtual Clusters on distributed Cloud environments using BonFIRE. *FIRE Engineering Workshop*, 2012.

[18] Ellen Gottesdiener. Business RULES. *Application Development Trends*, 1997.

[19] Qusay Hassan. Demystifying Cloud Computing. *The Journal of Defense Software Engineering*, 2011.

[20] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. *Proceedings of the 10th International Conference on Autonomic Computing*, 2012.

[21] Alastair C. Hume, Yahya Al-Hazmi, Bartosz Belter, Konrad Campowsky, Luis Carril, Gino Carrozzo, Vegard Engen, David Garcia-Perez, Jordi Jofre Ponsat, Roland Kubert, Yongzheng Liang, Cyril Rohr, and Gregory Van Seghbroeck. BonFIRE: A Multi-cloud Test Facility for Internet of Services Experimentation. *Testbeds and Research Infrastructure. Development of Networks and Communities*, 2012.

[22] Google Inc. Google App Engine: Platform as a Service. https://cloud.google.com/appengine/docs, 2015.

[23] Deepal Jayasinghe, Simon Malkowski, Jack Li, Qingyang Wang, Zhikui Wand, and Calton Pu. Variations in Performance and Scalability: An Experimental Sudy in IaaS Clouds using Multi-Tier Workloads. *IEEE Transactions on Service Computing*, 2013.

[24] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. *2nd International System Administration and Networking Conference*, 2000.

[25] Konstantinos Kavoussanakis, Alastair Hume, Josep Martrat, Carlo Ragusa, Michael Gienger, Konrad Campowsky, Gregory Van Seghbroeck, Carlos Vazquez, Celia Velayos, Frédéric Gittler, et al. Bonfire: The clouds and services testbed. *IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2013.

[26] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout: Building More Robust Cloud Applications. *ICSE*, 2014.

[27] Cristian Klein, Alessandro Vittorio Papadopoulos, Manfred Dellkrantz, Jonas Durango, Martina Maggio, Karl-Erik Arzén, Francisco Hernández-Rodriguez, and Erik Elmroth. Improving Cloud Service Resilience using Brownout-aware Load-Balancing. pages 31–40, 2014.

[28] A. Kovari and P. Dukan. KVM & OpenVZ virtualization based IaaS Open Source Cloud Virtualization Platforms: OpenNode, Proxmox VE. *Proceeding of the 10th IEEE Jubilee International Symposium on Intelligent Systems and Informatics*, 2012.

[29] Neal Leavitt. Is Cloud Computing Really Ready for Prime Time? *IEEE Computer Society*, 2009.

[30] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. *Fifth International Conference on Cloud Computing*, 2012.

[31] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the Performance Isolation Properties of Virtualization Systems. *Workshop on Experimental Computer Science*, 2007.

[32] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. *NIST Special Publication 800-145*, 2011.

[33] Mohammadreza Mohammadrezaei. Proposing a new model for infrastructure cloud computing. *Applied mathematics in Engineering, Management and Technology*, 2015.

[34] Peter Ondrejka, Martin Prpič, Rüdiger Landmann, and Douglas Silas. *Red Hat Enterprise Linux 6 Resource Management Guide*. Red Hat, Inc, 2015.

[35] Zhonghong Ou, Hao Zhuang, Andrey Lukyanenko, Jukka Nurminen, Pan Hui, Vladimir Mazalov, and Antii Ylä-Jääski. Is the Same Instance Type Created Equal? Exploting Heterogeneity of Public Clouds. *IEEE Transactions on Cloud Computing*, 2013.

[36] Claus Pahl. Containerisation and the PaaS Cloud. *IEEE CLOUD COMPUTING MAGAZINE*, 2015.

[37] David Garccia Perez, Juan Angel Lorenzo del Castillo, Yahya Al-Hazmi, Josep Martrat, Konstantinos Kavoussanakis, Alastair C. Hume, Celia Velayos Lopez, Giada Landi, Tim Wauters, Michael Gienger, and David Margery. Cloud and Network facilities federation in BonFIRE. *Euro-Par 2013: Parallel Processing Workshops*, 2013.

[38] Guillaume Pierre and Corina Stratan. ConPaaS: a Platform for Hosting Elastic Cloud Applications. 2012.

[39] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 2005.

[40] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds. *IEEE Transactions on Service Computing*, 2013.

[41] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud Computing: An Overview. *CloudCom 2009*, 2009.

[42] Stephen Soltesz, Herbert Pötzl, Marc Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. *EuroSys*, 2007.

[43] Luis Tomas, Cristian Klein, Johan Tordsson, and Francisco Hernandez-Rodriguez. The straw that broke the camel's back: safe cloud overbooking with application brownout. *Cloud and Autonomic Computing (ICCAC)*, 2014.

[44] Luis Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Computer Communications Review*, 2009.

[45] William Voorsluys, James Broberg, and Rajkumar Buyya. Introduction to Cloud Computing. *Cloud Computing: Principles and Paradigms*, 2001.