Evaluatie van resource-allocatieschema's op OpenStack

Jerico Moeyersons

Promotoren: prof. dr. ir. Filip De Turck, prof. dr. Bruno Volckaert
Begeleider: Pieter-Jan Maenhaut

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. Bart Dhoedt
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2016-2017

UNIVERSITEIT
GENT

Dankwoord

Na een intensieve periode van vijf maanden heb ik de laatste hand gelegd aan deze masterproef. Op verschillende vlakken heb ik nieuwe elementen binnen de wondere wereld van de informatica kunnen ontdekken. Daarom wil ik aan een aantal personen een welgemeende dankjewel zeggen om mij steeds te steunen tijdens het maken van deze masterproef.

Allereerst wil ik mijn promotoren, prof. dr. ir. Filip De Turck en prof. dr. Bruno Volckaert, bedanken voor het vertrouwen, de steun, de tips en de feedback. Ook mijn begeleider, de heer Pieter-Jan Maenhaut wil ik hartelijk bedanken voor de steun, de vele tips en uitgebreide feedback. Daarnaast wil ik ook mevrouw Leen Pollefliet bedanken voor de vele tips die ik nuttig heb kunnen gebruiken tijdens het schrijven en presenteren van deze masterproef. Vervolgens wil ik ook mijn vriendin, Lynn Haentjens, hartelijk bedanken om mij steeds te steunen in deze periode alsook voor het leveren van grammaticale feedback. Ten slotte wil ik ook mijn ouders bedanken voor de geleverde steun tijdens deze periode.

Om te eindigen met mijn dankwoord wil ik ook nog een aantal vrienden, namelijk Cédric Reyniers, Maxim Ronsse en Simon Vermeersch, bedanken voor de aangename middagpauzes, de relevante en ook de minder relevante gesprekken.

Bedankt allemaal!
Jerico Moeyersons

Evaluation of resource allocation schemes on OpenStack

Jerico Moeyersons
Supervisors: Prof. dr. ir. Filip De Turck, Prof. dr. Bruno Volckaert
Counsellor: Pieter-Jan Maenhaut

***Abstract*—One of the major challenges of cloud computing is to efficiently divide and manage resources over different customers, also referred to as resource allocation. Over recent years a lot of resource allocation schemes have been designed and tested by means of simulation tools such as the CloudSim framework, but very rarely on a real cloud testbed. Large experiments using real cloud testbeds are both expensive and time-consuming, especially when there are some faults in the new scheme. In this thesis, we try to offer an alternative to test and evaluate new allocation schemes on a small OpenStack cloud environment. OpenStack is a free, open-source software platform for cloud computing, deployed as Infrastructure-as-a-Service. With the aid of a scalable OpenStack application and a monitor web application, new resource allocation schemes can be plugged into an OpenStack cloud and evaluated. A Proof-of-Concept, with a Round Robin allocation scheme, is introduced to verify how easy and fast an evaluation works with this approach. After this evaluation, we can conclude that with OpenStack, two test scenarios and a monitoring application, a new resource allocation scheme can be easily and quickly evaluated.**

Keywords*—Cloud computing, OpenStack, resource allocation, Round Robin, monitoring

I. Introduction

With cloud computing, efficient resource management is of great importance. A cloud application can benefit from efficient resource management so it'll have enough resource to work well, will be more reliable and fault tolerant, etc. and thus good resource allocation algorithms are needed. In recent years, many new cloud resource allocation schemes have been designed and evaluated but only half of them are evaluated using a real cloud testbed. The others are only validated using cloud simulation frameworks such as CloudSim [1]. Though, simulations are needed for the design and development of new allocation strategies, evaluations on physical hardware can bring new insights such as how new resource allocation schemes react on the heterogeneity of physical resources. Running experiments on a public cloud is both expensive and time-consuming, causing that not every new allocation strategy will be evaluated on a public cloud because a simulator can do it faster.

One of the strengths of the cloud is the possibility to scale up and down or in and out. When instances, hosted by the cloud, are scaled up or down, additional resources are added or removed from that instance. This is an easy and fast approach but not every application can benefit from it because these applications need to be designed to deal with more resources. The other approach, scaling out or in is far more interesting but more complex. When a cloud application is scaled out, more instances are added to split the needed calculations over the available instances and when a cloud application is scaled in, some instances are deleted. This approach requires a specific software architecture so it can benefit from more instances but useful to test new resource allocation schemes.

In this thesis, a setup is presented for easy and quick evaluation of new allocation strategies on top of an OpenStack cloud [2] together with an OpenStack application and a monitor setup. To verify this setup, a Proof-of-Concept (PoC) with an existing allocation scheme, Round Robin, is evaluated.

II. Related work

Some work that can be related to this research is the Raspberry Pi cluster explained in Maenhaut et al [3]. In this paper, a cluster of Raspberry Pi's gives the opportunity to validate new allocation strategies instead of using a simulator. Through the provided web interface, new clusters can be easily configured. The way how the clusters are monitored is strongly related to the used monitoring system in this thesis, described in Section V.
Other work is RT-OpenStack [4], a CPU-resource management system plugged into OpenStack. The system consists of three components, namely the integration of a real-time hypervisor, a real-time scheduler and a VM-to-host mapping strategy. The interesting part of this work is the way how this system is used in combination with an OpenStack cloud but this thesis won't use custom management systems and hypervisors to evaluate new resource allocation schemes.
Beside this, there are also different simulators such as OpenStack EMU [5], a simulated OpenStack environment with a SDN controller and a large scale network emulator CORE. The quantity of simulators to evaluate new resource allocation schemes is one of the most import reasons of this thesis because of the need to evaluate these schemes on physical hardware instead of simulated hardware.

III. What is OpenStack?

OpenStack [2] is an open-source software platform for cloud computing, under the Apache License 2.0, that controls large

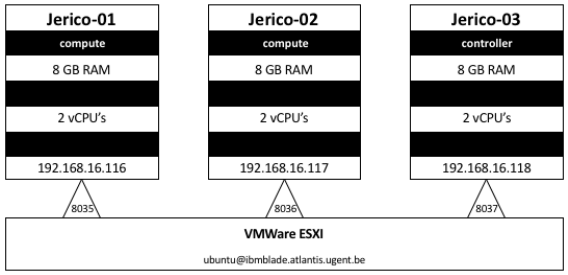


Fig. 1: Structure of the test environment

pools of compute, storage and networking resources throughout a datacenter. OpenStack works with popular open-source technologies so it's ideal for heterogeneous infrastructure. Most of the time, it is deployed as Infrastructure-as-a-Service (IaaS) whereby virtual servers, containers and other resources are made available to the customers. The platform itself consists of multiple, interconnected components managing the different resources (such as CPU, memory, storage, network, etc). All these components are controlled through a RESTful API, a web based dashboard or through command line tools. The most relevant components of OpenStack for this thesis are Nova, responsible for managing the compute resources, Heat, responsible for the orchestration services and Aodh and Ceilometer, both responsible for the telemetry services. Apart from these components, there are many other ones created by the OpenStack community but these are less related to this thesis.

IV. Test environment

For this thesis, OpenStack is deployed on 3 Ubuntu 16.04.02 LTS systems, each with 8 GB of memory, 2 vCPUs and 10 GB of storage. The OpenStack test environment is deployed using DevStack [6], a collection of bash scripts that automatically deploys an OpenStack cloud with less configuration, ideal for development or test environments. The structure of the test environment is visualized in Figure 1.
After the deployment of OpenStack, the Round Robin scheduler is plugged into Nova. By default, Nova works with a filter and weighting scheduler that first filters the available hosts and then selects the best hypervisor to host the new instances based on the amount of free memory, average CPU usage, etc. For implementing the Round Robin scheduler, the same principle is used, namely, Round Robin is written as filter with a row of the different hypervisors (in this case, the 3 hypervisors in Figure 1). When a new instance needs to be created, only one hypervisor will be chosen by the Round Robin filter and thus the default Nova weighter will let that hypervisor host the new instance (because there is but one hypervisor that came through the filter).
Finally, a cloud application called FaaFo (First App Application For Openstack) [7] is deployed with an OpenStack stack. FaaFo is a cloud application that calculates fractals with the possibility to split the work over different worker nodes, all orchestrated by a controller node. A stack, written in the Heat Orchestration Template (HOT) format [8], is used for the creation of security groups, alarms, servers, scaling groups, etc. Such a stack is then monitored by Ceilometer and Aodh, the Telemetry services of OpenStack, and when the CPU-usage is too high or low, Heat will scale out or in the whole cloud application or in this case the number of worker nodes.
The combination of these three steps allows us to create a cloud environment, plug in a new allocation strategy and generate some workload to test the placement of new instances and the distribution of resources over the different hypervisors.

V. Monitoring tool

There are many possibilities to monitor the hosts of an OpenStack cloud. The first possibility is Rally [9], a tool that automates and unifies a multi-node OpenStack deployment, verifies the cloud, benchmarks the deployment and profiles it. In this context, Rally is used for executing benchmarking tests and for the possibility to verify the cloud or in other words, to check of the cloud is working properly under simulated workload.
The second possibility is a custom developed nodejs-agent. This agent, developed using Node.js, uses different Node packages to construct a web service that will save the current resource usage such as CPU-usage, free memory and number of hosted instances in a MongoDB and also returns these results to the caller.
Other monitor tools exists, such as Grafana and DataDog. The OpenStack community is also busy with a project called Monasca that will be able to monitor the whole OpenStack cloud.
For the evaluation of resource allocation schemes on OpenStack, we used the first and second possibility because it is a free option with less and easy configuration in contrast to DataDog, a paying service, or Grafana that needs many and difficult configuration. Also the possibility to adapt this application or expand it with more graphs is interesting for other evaluations.

VI. Evaluation of schemes

With the test environment described in Section IV and the monitoring tools Rally and the nodejs-agent described in Section V, the PoC with the Round Robin scheduler can be evaluated.

First of all, a Rally test creates 10 times a standard instance with two iterations at the time (concurrency), simulating that two users are creating instances at the same time. While the test was being executed, the nodejs-agent was monitoring the three hypervisors and show the results as visible in Figure 2 after the completion of the Rally test. In this case, the instances are equally divided between the three hypervisors but the CPU-usage and the memory usage of jerico-03, the OpenStack controller, is significantly greater than the two OpenStack-compute nodes.
After the Rally test, the FaaFo application is deployed with a stack on the cloud. By default, the deployment consists of

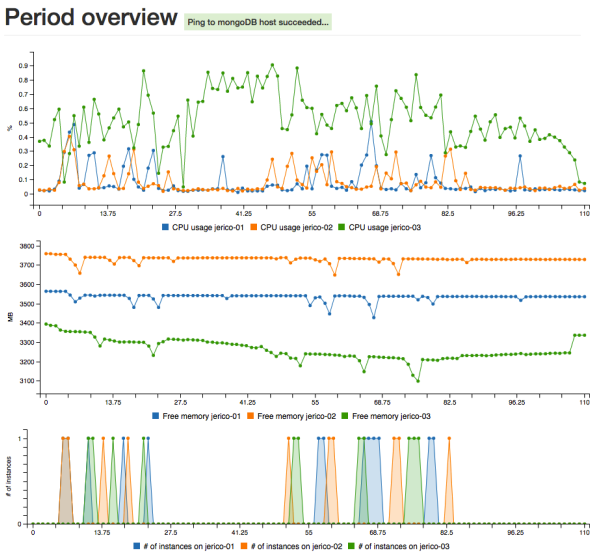


Fig. 2: Results of the Rally test in the nodejs web application

the required security groups, alarms, scaling policies, one controller node and one worker node. When the deployment is complete and FaaFo is running, a task to create three 5555 x 5555 fractals is started on the FaaFo controller. The only worker node will start to calculate these fractals with a high CPU usage as a result. Thanks to the configured alarms in the stack, Ceilometer and Aodh will monitor the worker node and trigger an alarm when the CPU usage is too high. Heat, on his turn, will then start a new worker node that can help the other node to calculate the fractals. This will go on till everything is calculated and when the CPU usage starts decreasing, each unnecessary worker node will be deprovisioned.

From the Rally test, the following can be concluded: the creation and deletion of different instances in this order is no problem for a Round Robin scheduler. The whole process of the FaaFo test, from deploying the stack, calculate the fractals and in- and out-scaling the worker nodes is also monitored with the nodejs agent and visualized in Figure 3. Here, we can conclude that in the beginning, new instances are equally divided between the hypervisors but near the end, after deprovisioning of some instances, this equal distribution of instances faded away. Also the available memory on the OpenStack controller is very low. In a worst case scenario, there is a possibility that when Heat is consequently scaling-up and scaling-down, all instances will be hosted on one hypervisor. The reason for this is that Round Robin acts without any context, meaning that it doesn't know the capabilities of the hypervisors.

In both tests, there is one remarkable similarity, namely the CPU and memory usage of the OpenStack controller is much greater than the other nodes. A possible explanation for this are the many tasks an OpenStack controller has to do,

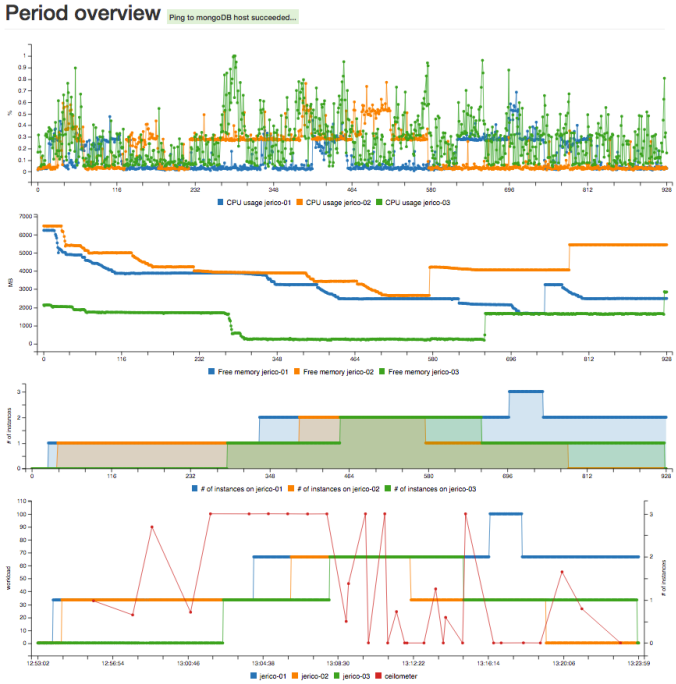


Fig. 3: Results of the FaaFo test in the nodejs web application

such as scheduling, logging, authentication, etc. A possible fix to lower the CPU and memory usage of this OpenStack controller is to disable the hypervisor on it so it only needs to manage OpenStack and not OpenStack in combination with some instances.

VII. Conclusion

The target of this thesis was to provide an easy and quick method to evaluate new resource allocation schemes on a real cloud testbed. With OpenStack, a new allocation strategy can easily be plugged into Nova, whether or not as a filter. A whole new approach of writing schedulers for Nova is possible but requires further research. Rally and the FaaFo application will then test the new strategy while the nodejs-agent monitors everything so it can be easy evaluated afterwards.

References

[1] R. N. Calheiros, R. Rajiv, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit formodeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software - Practice and Experience*, vol. 41, no. 7, pp. 23 – 50, 2011.

[2] OpenStack, "OpenStack," 2017. [Online]. Available: https://www.openstack.org/

[3] P.-j. Maenhaut, B. Volckaert, V. Ongenae, and F. D. Turck, "RPiaaS : A Raspberry Pi Testbed for Validation of Cloud Resource Management Strategies," *proceedings of the 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), Atlanta, GA, USA*, 2017.

[4] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. X. Phan, I. Lee, and O. Sokolsky, "RT-Open Stack: CPU Resource Management for Real-Time Cloud Computing," *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing, CLOUD 2015*, pp. 179–186, 2015.

[5] C. H. Benet, R. Nasim, K. A. Noghani, and A. Kassler, "OpenStackEmu - A Cloud Testbed Combining Network Emulation with OpenStack and SDN," *The 14th Annual IEEE Consumer Communications & Networking Conference*, no. January, 2017.

[6] DevStack, "Multi-Node Lab," 2017. [Online]. Available: https://docs.openstack.org/developer/devstack/guides/multinode-lab.html

[7] OpenStack, "FaaFo - LibCloud," 2017. [Online]. Available: https://developer.openstack.org/firstapp-libcloud/getting{_}started.html

[8] OpenStack, "Heat Orchestration Template (HOT) specification," 2017. [Online]. Available: https://docs.openstack.org/developer/heat/template{_}guide/hot{_}spec.html{\#}hot-spec

[9] OpenStack, "Rally," 2017. [Online]. Available: http://rally.readthedocs.io/en/latest/index.html

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Lijst van listings

"If you think you've seen this movie before, you are right. Cloud computing is based on the time-sharing model we leveraged years ago before we could afford our own computers. The idea is to share computing power among many companies and people, thereby reducing the cost of that computing power to those who leverage it. The value of time share and the core value of cloud computing are pretty much the same, only the resources these days are much better and more cost effective."

~David Linthicum, author of Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide

1

Inleiding

Het onderzoek naar nieuwe cloud resource allocatieschema's staat niet stil. Elk jaar worden er nieuwe schema's ontwikkeld en gevalideerd met behulp van een simulator en/of een fysiek cloud testbed. Een belangrijke opmerking hierbij is dat niet elk nieuw schema wordt uitgetest op een fysiek cloud testbed wegens de complexiteit en de kostprijs. Hierdoor komen verborgen constanten van een fysiek cloud testbed niet aan het licht en dat heeft mogelijks een grote invloed op de performantie van het nieuwe schema. In deze masterproef wordt daarom dieper ingegaan op het bieden van een zo goedkoop mogelijk alternatief om deze nieuwe schema's te testen in een OpenStack [**?**] cloud-omgeving. Met behulp van DevStack [**?**] wordt een OpenStack cloud-omgeving uitgerold waarbij een bestaand resource allocatiechema wordt ingeplugd met de *nova scheduler* in plaats van het standaard meegeleverde schema van OpenStack zelf. Dankzij Rally [**?**] en een Node.js-agent kunnen de gecreëerde instanties gemonitord worden waardoor de hele testopstelling uiteindelijk de mogelijkheid biedt om nieuwe allocatieschema's te evalueren in een reële OpenStack cloud-omgeving.

1.1 Wat is cloud computing?

Wat is de cloud? Waar is de cloud? Zijn we in de cloud? Dit zijn allemaal relevante vragen want de term *cloud computing* is overal [**?**]. Het cloud computing paradigma beschrijft applicaties waarbij

toegang gebeurt via het internet en die een *pool van resources* delen met andere applicaties en/of toepassingen. De eigenlijke term cloud computing verwijst naar applicaties die beschikbaar zijn via het internet alsook naar de hardware- en softwaresystemen in verschillende datacenters die deze applicaties hosten [**?**].

In een cloud-omgeving zijn er steeds drie belangrijke stakeholders die vervuld worden. Ten eerste zijn er de *cloud providers*, organisaties die fysieke en virtuele servers verhuren aan andere organisaties, bedrijven, etc. Vervolgens zijn er de *cloud-gebruikers* die een omgeving huren van een cloud provider om zo een dienst aan de volgende categorie te bieden. De laatste categorie beschrijft de *eindgebruikers*, ook wel de gebruikers genoemd. Ze gebruiken de aangeboden diensten van de cloud-gebruikers waardoor ze een werklast genereren. Een schematisch overzicht wordt weergegven in Figuur **??**. Een mooi voorbeeld van de drie rollen en hun onderlinge overeenkomsten is Netflix. Netflix is een cloud-gebruiker die een cloud omgeving afhuurt van een grote organisatie zoals bijvoorbeeld Google of Amazon (de cloud providers). De mensen thuis (eindgebruikers) kijken naar verschillende series, films, etc. waardoor ze op hun beurt werklast genereren voor de servers die Netflix huurt.

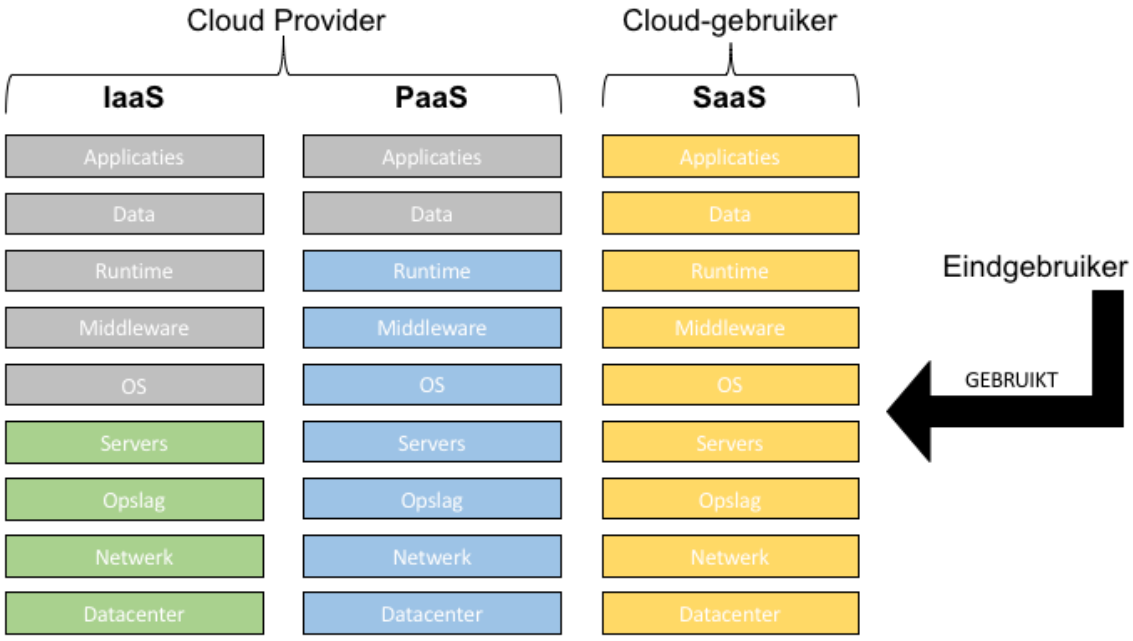
De drie stakeholders van cloud-computing hebben elk hun kosten. Cloud providers betalen de kosten om infrastructuur aan te kopen en deze te onderhouden waarna ze deze infrastructuur meestal aanbieden via een pay-as-you-go schema aan de cloud-gebruikers. Cloud-gebruikers daarentegen factureren vaak vaste maandelijkse bedragen aan hun eindgebruikers waardoor het efficiënt gebruiken van resources zeer belangrijk is zodat hun eigen kosten aan de cloud providers gedekt worden.

1.1.1 Tenants en *multi-tenancy*

Multi-tenancy betekent dat er verschillende aanvragen van verschillende organisaties of bedrijven (tenants) concurrent behandeld worden door 1 of meerdere instanties van de applicatie die de hardware- en software-infrastructuur met elkaar delen [**?**]. Een tenant wordt dus voorgesteld als een eindgebruiker die een deel van de cloud-applicatie wenst te gebruiken, al dan niet gedeeld met andere tenants. Hierbij moet de cloud-applicatie zo ontwikkeld worden dat deze multi-tenancy ondersteunt. Cloud computing zelf is een vorm van multi-tenancy aangezien verschillende gebruikers dezelfde infrastructuur met elkaar delen.

1.1.2 Service Level Agreement

Een Service Level Agreement (SLA) is een belangrijk onderdeel in de wereld van cloud computing. Het is een overeenkomst tussen twee verschillende partijen die dienst doet als blauwdruk en garantie biedt over de aangeboden diensten [**?**]. Zo sluit bijvoorbeeld een cloud provider een



Figuur 1.1: Cloud stakeholders met de bijhorende cloud-vorm

SLA af met een cloud-gebruiker en sluit een cloud-gebruiker een SLA af met de eindgebruikers. Een SLA kan beschouwd worden als een reeks van verwachtingen en criteria waardoor geen van beide partijen voor verrassingen komen te staan bij eventuele wijzigingen, zowel verwachte als onverwachte, in de cloud service. Een aantal voorbeelden van deze criteria zijn beschikbaarheid (99.95% *uptime*), performantie, beveiliging, *data recovery*, locatie van de data, etc. Dankzij zo'n SLA kan een gebruiker de meest geschikte (beste prijs/kwaliteit afhankelijk van de noden) cloud-oplossing kiezen, want niet elke gebruiker heeft een zeer betrouwbaar en duur systeem nodig.

1.2 Probleemstelling en doel van de masterproef

Er zijn al meerdere nieuwe technieken ontwikkeld om cloud resources te alloceren en deze worden samengevat in Sectie **??**. Een belangrijk detail dat hierbij nog niet vermeld is: iets minder dan de helft van deze nieuwe methoden zijn enkel getest in simulators en dus niet in echte cloud omgevingen zoals we zullen zien in Tabel **??** van Hoofdstuk **??**. Deze kolom geeft aan of het nieuwe schema getest is op een echte cloud-omgeving of met behulp van een simulator en de verdeling is respectievelijk 16 schema's en 13 schema's. Een mogelijke belangrijke reden hiervoor is de zeer hoge kostprijs bij eventuele fouten in de methode zoals vermeld in de inleiding van Hoofdstuk **??**. Stel dat een onderzoeker zijn nieuwe methode wil uittesten in een gehuurde Microsoft Azure omgeving dan kunnen de kosten zeer hoog uitvallen indien er bijvoorbeeld te

veel resources worden aangevraagd door een fout in de methode.

Het doel van deze thesis is het mogelijk maken om nieuwe methoden te testen op een kleinschalige OpenStack cloud-omgeving. Met behulp van Nova en de Orchestration-services van OpenStack kan een nieuwe methode eenvoudig en snel worden getest. Via de Nova-scheduler kan het nieuwe schema worden ingeplugd en met behulp van de Orchestration-services kan een schaalbare applicatie horizontaal schalen. Dit zorgt voor een zware werklast op de cloud-omgeving waarbij de verschillende hosts gemonitord zullen worden. Ten slotte worden de verkregen resultaten geëvalueerd met nadien een eventuele vergelijking tussen deze resultaten en de resultaten van een simulatortest.

Om de werking van dit alles uit te leggen zal een *proof-of-concept* worden uitgewerkt met een eenvoudig schema. Allereerst zal het eenvoudig schema, Round Robin, worden ingeplugd in Nova. Vervolgens zal een schaalbare applicatie, namelijk FaaFo (First App Application For OpenStack) worden uitgerold op de ontwikkelde OpenStack-omgeving. Daarna zal, door een stijgende, gegenereerde werklast de applicatie horizontaal geschaald worden dankzij de Orchestration-services van OpenStack. De verschillende hosts worden hierbij gemonitord om te kijken of de werklast verdeeld werd zoals gewenst.

Het vervolg van deze masterproef wordt als volgt ingedeeld. In het volgende hoofdstuk wordt er dieper ingegaan op de termen resource-allocatie en OpenStack. Daarna wordt het relevant werk besproken dat een mogelijke leidraad kan bieden. Vervolgens beschrijft Hoofdstuk 3 de testomgeving en hoe deze tot stand is gebracht waarna Hoofdstuk 4 de verschillende mogelijkheden om te monitoren toelicht samen met de gekozen manier. Hoofdstuk 5 legt de manier waarop de evaluaties gebeuren uit met de bijhorende proof-of-concept. Ten slotte worden in Hoofdstuk 6 de besluiten geformuleerd, gevolgd door de bibliografie en enkele bijlagen.

2

Cloud resourcebeheer, resource-allocatie en OpenStack

In dit hoofdstuk wordt er dieper ingegaan op enkele belangrijke termen binnen cloud computing. Deze termen geven een beter idee over de werking van diverse cloud-systemen en de mogelijkheden die er zijn voor zowel de cloud-provider als de cloud-gebruiker, beiden besproken in Hoofdstuk **??** om hun cloud te beheren.

2.1 Cloud resourcebeheer

Om ervoor te zorgen dat een cloud provider en een cloud-gebruiker steeds voldoen aan hun SLA, moeten beide partijen *resource management* toepassen. Resource management heeft als doel de beschikbare bronnen zo efficiënt mogelijk te benutten, en precies voldoende bronnen te voorzien om *underload* en/of *overload* te vermijden.

2.1.1 Resourcebeheer: de mogelijkheden

Het doel van een cloud provider is het efficiënt beheren van de infrastructuur van het datacenter. Onderdelen hiervan zijn bijvoorbeeld *balanced load* waarbij resources op een welbepaalde manier worden toegekend zodat het gebruik gebalanceerd is over alle resources van dat type, *fault*

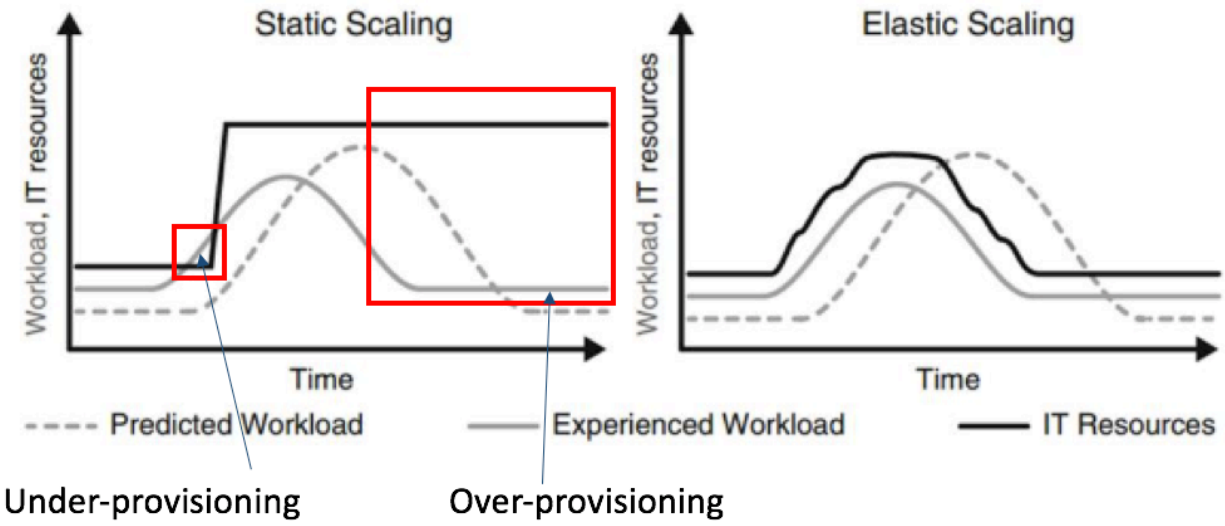
tolerance waarbij de resources worden toegekend zodat de impact van het falen van een onderdeel minimaal blijft, of *energy use minimization* waarbij de plaatsing van taken of jobs over resources zo gebeurt dat de energiekosten minimaal zijn. Een eventuele combinatie van bovenstaande onderdelen aan de hand van prioriteiten is mogelijk en de cloud provider kan ook bepaalde onderdelen koppelen aan verschillende operationele condities. Een voorbeeld van het laatste is het minimaliseren van het energieverbruik tijdens perioden met lage belasting, maar bij hogere belasting overschakelen naar balanced load in plaats van energy use minimization.

Een cloud-gebruiker heeft meestal een SLA met zijn eigen eindgebruikers met andere management doelen dan de cloud provider ten opzichte van zijn cloud-gebruikers. Hij maakt gebruik van de elasticiteit van cloud-omgevingen om zo extra resources te reserveren tijdens perioden met een hogere werklast en resources vrij te geven in perioden met minder werklast.

2.1.2 Schalen

Een cloud-gebruiker kan in twee probleemsituaties terecht komen indien hij verkeerd gebruikmaakt van de elasticiteit van cloud-omgevingen: *over-provisioning* en *under-provisioning*. Over-provisioning betekent dat de cloud-gebruiker meer resources ter beschikking heeft dan nodig met een hogere kost als gevolg. Doordat cloud computing ook een “business” is, zorgt dit voor nadelige effecten bij zowel de cloud-gebruiker als eventuele eindgebruikers. Anderzijds betekent under-provisioning dat de cloud-gebruiker te weinig resources ter beschikking heeft, waardoor hij bijvoorbeeld performantiecriteria in de SLA met zijn eindgebruiker niet kan nakomen. Ook dit is een zeer nadelig effect voor beide partijen aangezien een eindgebruiker dan langer moet wachten op een antwoord (langere antwoordtijd) of geen antwoord krijgt door het falen van componenten.

Schalen van de beschikbare resources (of gebruik maken van de elasticiteit van een cloud-omgeving) voorkomt grotendeels de twee bovenstaande nadelige effecten. Hierin worden twee grote categorieën onderscheiden, statisch of periodiek en horizontaal of verticaal schalen. Bij statisch schalen worden er meer fysieke servers voorzien maar de tijd om deze te ordenen, te configureren en op te starten zal niet reactief genoeg zijn om een snelle stijging van de werklast te kunnen verwerken. Het gevolg hiervan is eerst een under-provisioning en nadien een over-provisioning omdat de opstart te lang duurt en het verlagen van het aantal resources pas na een bepaalde tijd van mindere activiteit zal worden uitgevoerd. Bijgevolg zorgt dit voor een slechtere performantie in het begin, door de under-provisioning, en een te hoge kostprijs achteraf door de over-provisioning. Elastisch schalen daarentegen voorziet een stijging in resources met kleine stappen wat het systeem veel reactiever en kost-efficiënter maakt. Een belangrijke opmerking hierbij is dat het systeem zo ontworpen moet zijn dat het kan schalen, of met andere woorden, nuttig gebruik kan maken van de extra verkregen resources. Een SQL-databank is een voorbeeld

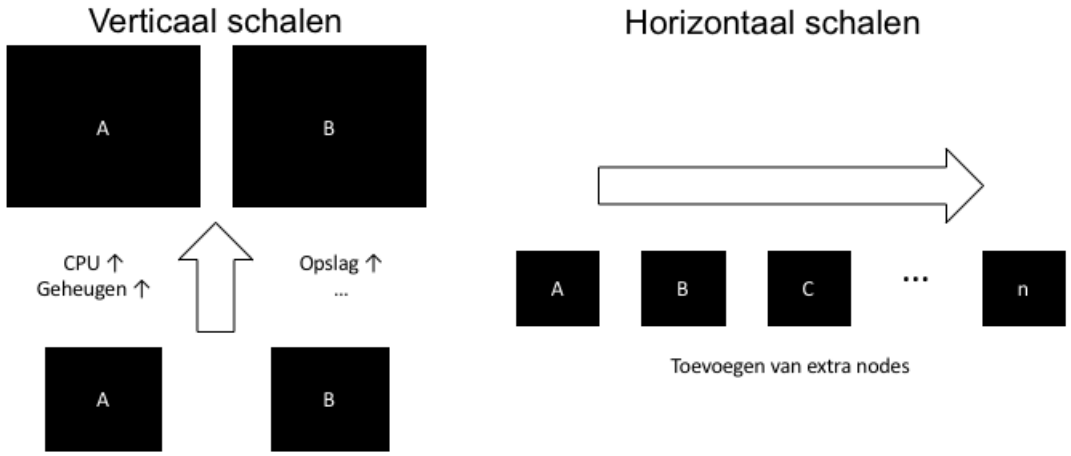


Figuur 2.1: Statische vs. elastische schaling [**?**].

van een minder schaalbare toepassing [**?**] omdat het gebruikmaakt van transacties die het gehele gebeuren blokkeren tot er een *commit* wordt uitgevoerd. Het toekennen van extra resources heeft hier dus weinig nut omdat de toegang tot de gegevens geblokkeerd blijft.

In Figuur **??** worden twee grafieken weergegeven die de toegekende resources weergeven bij statisch of elastisch schalen. Zoals te zien is, hangt statisch schalen sterk af van eventuele voorspellingen van de werklast waardoor het minder performant is bij pieken. Ook de nadelige effecten van under- en over-provisioning worden duidelijk weergegeven in de linkergrafiek. De rechtergrafiek weerspiegelt de kost-efficiëntie en performantie bij elastisch schalen en voor dit scenario is het duidelijk de aangewezen werkwijze om te schalen.

Naast statisch of elastisch schalen kan er ook verticaal of horizontaal geschaald worden. Verticaal schalen wordt omschreven als het toevoegen van extra hardware zoals meer CPU-kracht, extra geheugen en opslag, etc. De voordelen zijn onder andere dat elke applicatie hier gebruik van kan maken omdat dit geen speciale architectuur vereist. Daarnaast omvat het een weinig risicovolle en minder complexe operatie dan horizontaal schalen en ten slotte is de kost ook redelijk in vergelijking met algoritmische verbeteringen. De nadelen wegen echter iets meer door dan de voordelen. Zo is er mogelijks een *downtime* als er fysieke hardware wordt toegevoegd en is het niet zeker dat de applicatie gebruik kan maken van de nieuwe hardware. Een voorbeeld van dat laatste is een *single-threaded* applicatie waarbij het toevoegen van extra CPU-kernen de applicatie niet zal versnellen en bijgevolg weinig nut heeft. Het is ook veel moeilijker om te *downgraden* bij een dalende werklast en bovendien is er een absolute bovenlimiet doordat de onderliggende hardware van de fysieke node beperkt is.



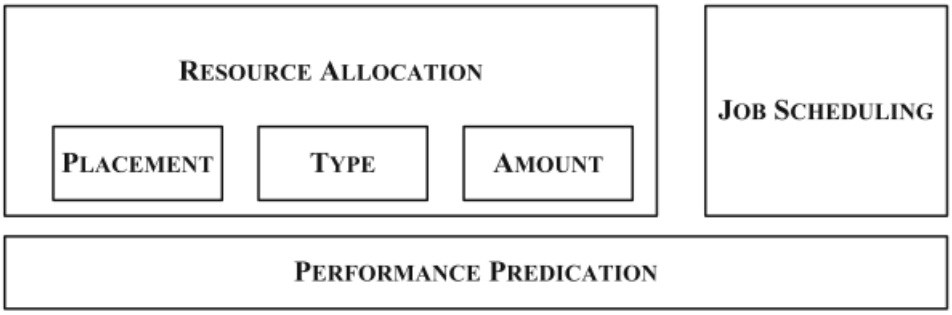
Figuur 2.2: Verticaal vs. horizontaal schalen van cloud resources

Horizontaal schalen voegt extra fysieke- of applicatienodes toe in plaats van deze te upgraden waardoor een veel hogere capaciteit mogelijk is dan de grootste beschikbare enkelvoudige node. Een belangrijk voordeel is de hoge kost-efficiëntie omdat het elastisch schalen toelaat. De enige vereiste is dat de applicatie of toepassing die erop draait een gedistribueerde architectuur moet bevatten zodat verschillende nodes dezelfde functionaliteit van een applicatie kunnen uitvoeren. Meestal zijn zulke nodes volledig identiek geconfigureerd –zelfde hardware resources, zelfde besturingssysteem, zelfde software– waardoor ze homogeen zijn. De homogeniteit van bepaalde nodes is een zeer belangrijke eigenschap bij horizontaal schalen omdat hierdoor *round-robin load balancing* goed werkt en het maakt capaciteitsplanning en auto-schalen veel eenvoudiger dan bij heterogene nodes. Figuur **??** geeft een overzicht van verticaal en horizontaal schalen.

2.2 Resource-allocatie

Resource allocation is een onderdeel van een overlappend concept samen met *resource provisioning* en *resource scheduling* [**?**]. Resource provisioning en resource-allocatie zorgen voor het toekennen van resources van een cloud provider aan een cloud-gebruiker in een concurrerende omgeving van groepen programma's of gebruikers. Resource scheduling geeft een overzicht weer van welke resources er beschikbaar zijn en welke er gedeeld worden op bepaalde tijdstippen om zo zware rekentaken te plannen. Samengevat beschrijven deze drie termen het bepalen van wanneer rekenactiviteit moet gestart of gestopt worden, gebaseerd op voorgaande activiteiten en relaties alsook op de *gealloceerde resources* en de duur van de activiteit.

In het algemeen bepaalt een cloud-gebruiker de hoeveelheid en het type van de resources die hij nodig heeft en zal de cloud provider deze aangevraagde resoures toewijzen in hun datacenters. Bij het bepalen van de hoeveelheid en het type van de resources moet er rekening gehouden worden



Figuur 2.3: Basisoverzicht van het resource-allocatie mechanisme.

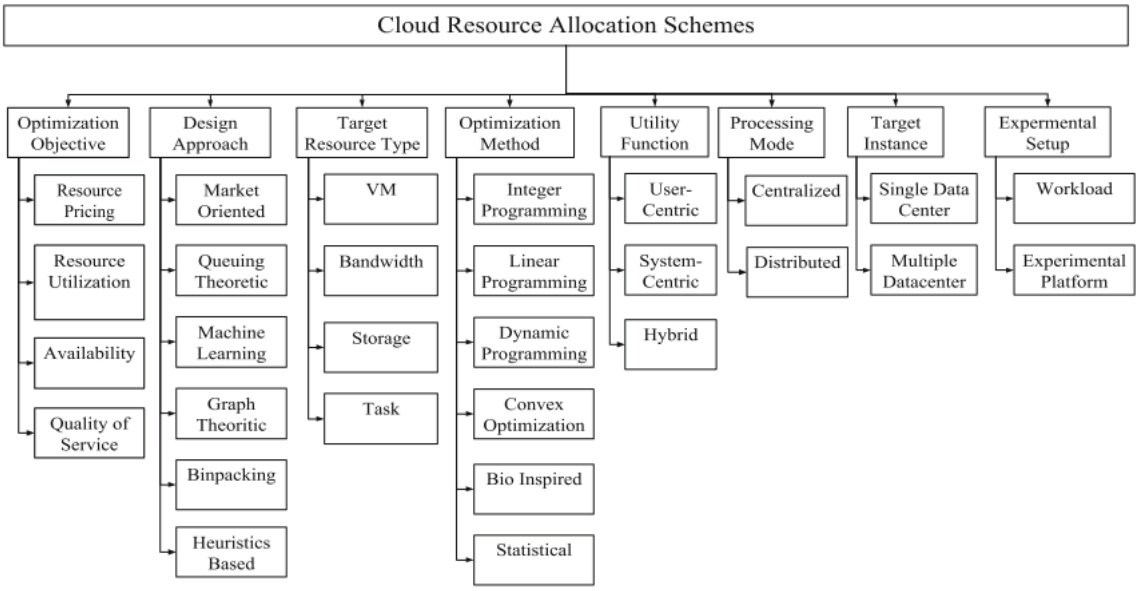
met eventuele vereisten zoals bijvoorbeeld de maximale duur van een taak. Zoals te zien in Figuur **??** beschrijft het cloud resource-allocatie mechanisme de beslissing in hoeveel, welke, waar en wanneer de beschikbare resources moeten toegekend worden aan de cloud-gebruikers. Dankzij de elasticiteit van cloud omgevingen kan een cloud-gebruiker resources dynamisch aanvragen en vrijgeven.

Cloud providers en cloud-gebruikers proberen uiteraard zoveel mogelijk winst te maken. Een cloud provider poogt dit door zoveel mogelijk virtuele machine's (VM's) uit te rollen op elke fysieke machine zodat de inkomsten hoog zijn en de investeringen laag. Een logisch gevolg hiervan is dat te veel VM's uitrollen op een fysieke machine kan leiden tot performantiedegradatie waardoor de cloud-gebruikers minder tevreden zijn. Cloud users daarentegen willen hun kost minimaal houden voor een maximale performantie, wat betekent dat ze efficiënt de nodige resources zullen reserveren voor hun applicatie.

Doordat de datacenters van cloud providers heterogeen toenemen en dus bestaan uit verschillende generaties van hardware, zullen verschillende cloud-gebruikers heterogene resources van de cloud provider gebruiken (en delen met andere cloud-gebruikers) zonder veel inzicht in de infrastructuur met als nadelig gevolg dat het hele systeem onvoorspelbaar wordt. Het doel van resource-allocatie is zorgen voor efficiënte cloud-services, waarbij efficiënt staat voor het toekennen van de juiste resources aan de juiste applicatie op het juiste tijdstip. Hierdoor kunnen applicaties deze resources nuttig gebruiken.

2.2.1 Indeling van cloud resource-allocatie schema's

Zoals weergegeven in Figuur **??** kunnen de cloud resource-allocatie schema's onderverdeeld worden in acht categorieën [**?**]. Deze worden hieronder per categorie kort besproken.



Figuur 2.4: Indeling van resource-allocatie schema's voor cloud computing [**?**].

Optimalisatiedoel (*Optimization objective*)

De huidige resource-allocatie schema's pogen vier optimalisatiedoelen te volbrengen: *Resource pricing* formuleert prijsmodellen om het algemeen verbruik van cloud resources te verbeteren waardoor cloud resources gekocht, verkocht en geruild kunnen worden met andere cloud providers en/of cloud-gebruikers. *Resource utilization* maximaliseert het gebruik van de virtuele en fysieke resources voor een betere performantie en is zeer gerelateerd met energieconsumptie. *Availability* verzekert een bepaalde beschikbaarheid tijdens een contractuele periode door het dupliceren van taakuitvoering op resources of het dupliceren van data op verschillende geografische locaties, om zo om te kunnen gaan met x-voudige *failures*. *Quality of Service* zorgt voor een snelle responsetijd, weinig latency, etc.

Ontwerpbenadering (*Design approach*)

De ontwerpbenadering representeert het onderliggende model van de resource-allocatie en komt voor in vijf verschillende types. Het marktgeoriënteerd model (*market-oriented*) benadert de functies van het vraag en aanbod systeem. Het *queuing theoretic model* voorspelt verschillende performantiemetrieken en bijgevolg de nodige resources om de werklast te kunnen verwerken. *Machine learning models* bekijken de resource-allocatie als een voorspellingsfunctie waarbij resultaten worden verwerkt met statistische technieken om zo verschillende patronen te ontdekken van bijvoorbeeld aanvragen, onzekerheidsfactoren, etc. *Graph theoretic models* abstraheren cloud-systemen en zijn componenten voor het opstellen van modellen alsook voor het optimali-

seren van de resource-allocatie aan de hand van grafen waarin de taken en de *workflow* worden weergeven. Het *bin packing model* plaatst verschillende items in een minimaal aantal *bins* en dat wordt hier gebruikt om VM's te combineren op de fysieke nodes en om vrijgegeven resources consequent samen te nemen met als voordeel een efficiënter energieverbruik. Samengevat bevindt het cloud resource-allocatie probleem zich in de NP, NPC en NP-hard complexe klassen. Oplossingen baseren zich daarom op heuristische methoden, die minder complex en sneller zijn, om dit probleem op te lossen.

Type van doel-resource (*Target resource type*)

Het *target resource type* bepaalt hoe de extra resources worden afgeleverd aan de cloud-gebruiker of eindgebruiker. Afhankelijk van de vraag kan er bijvoorbeeld een extra VM ter beschikking worden gesteld, extra opslagruimte of bandbreedte voorzien worden, etc.

Optimalisatiemethode (*Optimization method*)

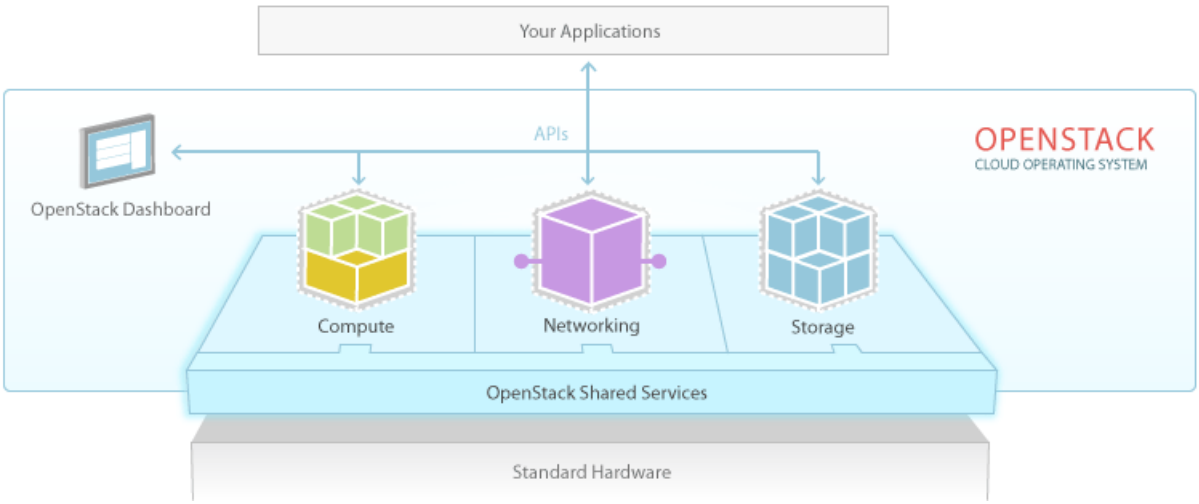
De optimalisatiemethode beschrijft de mathematische formulering voor het resource-allocatie probleem om zo een optimale of bijna-optimale oplossing te bepalen. Doordat de vraag en budgetten van gebruikers steeds variëren zijn polynomiale tijdsalgoritmen ongeschikt voor het oplossen van dit probleem en worden er daarom verschillende optimalisatiemethoden gebruikt om bijna-optimale oplossingen te creëeren zoals *integer programming*, genetische algoritmen, etc.

Nutsfunctie (*Utility function*)

De nutsfunctie bepaalt of het resource-allocatie schema een *system-centric*, *user-centric* of *hybrid* nut volgt. Deze drie types beschrijven hoe de cloud zal reageren indien het geen nieuwe resources meer kan alloceren. Bij system-centric staat de cloud provider zijn performantie centraal, bij user-centric staan de cloud-gebruikers centraal en het hybride-type bevindt zich tussen beiden.

Verwerkingsmodus (*Processing mode*)

Het doel van de nutsfunctie kan behaald worden door een optimalisatiemethode te gebruiken in een oftewel gecentraliseerde processing modus of een gedecentraliseerde modus. De gedecentraliseerde modus heeft als voordeel dat het geen *single-point of failure* is maar het kan, in vergelijking met de gecentraliseerde modus, geen globale informatiestatus bevatten.



Figuur 2.5: Architectuur van OpenStack [**?**].

Doel-instantie (*Target instance*)

De doel-instantie bepaalt of het resource-allocatie schema moet werken in een *single data center (SDC)* of een *multiple data center (MDC)* cloud omgeving.

Experimentele setup (*Experimental setup*)

Elk nieuw resource-allocatie schema wordt eerst getest in een experimentele omgeving waarbij telkens het platform en de werklast bepaald moeten worden.

2.3 OpenStack: een open-source softwareplatform voor cloud computing

OpenStack [**?**] is open-source cloud computing software voor het beheren en onderhouden van clouds. Het OpenStack project is gestart in 2010 door Rackspace en NASA die verantwoordelijk zijn voor de eerste versie van de code. Ondertussen is OpenStack al uitgegroeid tot een ruime community met ondersteuning van meer dan 150 bedrijven [**?**].

Figuur **??** geeft de basisarchitectuur weer van OpenStack, waarbij duidelijk wordt dat het bestaat uit verschillende onafhankelijke componenten met een welgevormde *Application Programming Interface* (API). OpenStack *Compute* –Nova– zorgt voor het beheren van VM's. *Storage* zorgt dan weer voor het beheren van de opslag en *Networking* zorgt voor de netwerkresources. Het OpenStack *Dashboard* is een webinterface waarmee de 3 bovenstaande blokken van componenten

beheerd worden, zoals het starten of stoppen van VM's, virtuele netwerktopologieën maken, etc.

Naast de vier aanwezige basiscomponenten in Figuur **??**, bestaan er nog verschillende andere componenten die deel uitmaken van het OpenStack project. Zo zijn er de *Orchestration*-services Heat [**?**], de Telemetry-Services [**?**] *Ceilometer* en *Aodh*, etc. Bovendien bestaan er naast componenten ook andere toepassingen die ontwikkeld zijn voor en door OpenStack. Zo is er bijvoorbeeld het *Heat Orchestration Template* (HOT) [**?**] formaat, ontwikkeld door community van OpenStack voor het aanmaken en configureren van verschillende onderdelen in Heat, Ceilometer, Nova, Neutron, etc.

Op dit ogenblik bevindt OpenStack zich in een vijftiende release, Ocata genaamd.[1] Deze release is de opvolger van Newton en brengt enkele verbeteringen aan zoals betere stabiliteit en een versterkte *core*-infrastructuur maar ook enkele nieuwe functies zoals de integratie van containers [**?**].

[1] Februari 2017

3

Gerelateerd werk

Aangezien resource allocatieschema's van clouds nog steeds worden onderzocht, bestaan er enkele werken die gerelateerd kunnen worden aan deze masterproef. In Sectie **??** wordt nadien een overzicht gegeven van reeds ontwikkelde cloud resource allocatieschema's.

3.1 Relevant onderzoek

Een zeer gerelateerd werk van Maenhaut et al. [**?**] beschrijft een Raspberry Pi cluster voor het valideren van cloud management strategieën. Dankzij een eenvoudige webinterface kan in deze demo alles eenvoudig geconfigureerd en getest worden. De Node.js-agent beschreven in Sectie **??** heeft dan ook inspiratie gehaald uit deze demo mits enkele aanpassingen en verbeteringen omdat hier gebruik wordt gemaakt van fysieke servers in plaats van Raspberry Pi clusters.

RT-OpenStack [**?**] plugt een CPU-resource management systeem in op een OpenStack omgeving. Het is een systeem bestaande uit 3 onderdelen, namelijk de integratie van een real-time hypervisor, een real-time scheduler en een VM-to-host mapping strategie wat dus vooral de nadruk legt op real-time VM's. Een belangrijke onderdeel van dat werk is de manier waarop RT-OpenStack werkt in combinatie met een normale OpenStack cloud-omgeving maar in deze thesis wordt er geen gebruik gemaakt van een aangepast management systeem noch van aangepaste hypervisors

voor de evaluatie van allocatieschema's.

Een ander voorbeeld van een inplugbaar framework in OpenStack is OpenStack Neat [**?**]. OpenStack Neat is eenvoudig inplugbaar in een bestaande OpenStack cloud-omgeving en gaat zorgen voor dynamische en energie-efficiënte algoritmen om VM's te consolideren. Daarbovenop biedt het ook de mogelijkheid om nieuwe VM consolidering algoritmen te evalueren en te vergelijken. De structuur van het framework en de werking ervan zijn zeer relevant doch ligt in deze thesis de focus meer op de correcte werking van het algoritme in plaats van de energie-efficiëntie.

Daarnaast bestaan er ook verschillende simulators om nieuwe schema's te testen zoals bijvoorbeeld OpenStackEmu [**?**]. Hierbij wordt een OpenStack omgeving gesimuleerd samen met een *Software Defined Networking (SDN)* controller en een large-scale network emulator CORE (*Common Open Research Emulator*). Dergelijke oplossing benadert al iets meer een reëele cloud-omgeving en de gebruikte methode is gerelateerd aan de werking van deze thesis. Enkel wordt er hier getracht om een echte cloud-omgeving te gebruiken in plaats van te benaderen of te simuleren.

Een ander voorbeeld van een simulatietool wordt beschreven door Maenhaut et al. [**?**] waarbij verschillende nieuwe schema's worden getest met behulp van verschillende invoerparameters. Deze resultaten geven een goed oog op de mogelijk performantie maar opnieuw gaat het om een simulator waarbij mogelijke constanten van een echte cloud-omgeving verborgen blijven. Deze thesis probeert daarom deze verborgen constanten zichtbaar te maken door de evaluatie uit te voeren op een reëel cloud-testbed.

Ten slotte bestaan er ook enkele interfaces om eenvoudig met OpenStack te communiceren zoals Apache Libcloud [**?**], een python bibliotheek, of pkgcloud [**?**], een node.js-pakket. Via een eenvoudige interface bieden ze een mogelijkheid aan om met verschillende cloud-besturingssystemen, zoals onder andere OpenStack, Google Cloud Platform, CloudFlare, etc te communiceren en hierop commando's uit te voeren. Indien deze interfaces gebruikt kunnen worden bij bijvoorbeeld de evaluatie van zo'n nieuw schema kan dit de overstap naar een andere cloud-omgeving zeer eenvoudig maken aangezien de commando's dezelfde blijven en de interface deze automatisch zal omvormen naar de commando's typisch voor het gebruikte cloud-systeem. Deze bibliotheken bieden dus de mogelijkheid om gebruikte cloud-omgeving te besturen en niet om evaluaties uit te voeren.

3.2 Beknopt overzicht van cloud resource allocatieschema's

Globale planning van gevirtualiseerde resources beschrijft het systeemwijde perspectief van de allocatie van fysieke en virtuele resources in een cloud omgeving. De meeste van deze methoden

zijn gecentraliseerd zodat ze volledige controle hebben over de allocatie van een verkregen set van resources. De hiervoor gebruikte technieken worden onderverdeeld in 4 groepen namelijk de initiële plaatsing van de VM's, dynamische plaatsing van VM's, VM-plaatsing rekening houdend met de netwerkbronnen en technieken om het energieverbuik te minderen.

De initiële plaatsing van VM's op fysieke machine's is gerelateerd aan het *vector bin packing* probleem, wat NP-hard is waardoor enkel heuristische methoden in aanmerking komen [**?**]. Enkele voorbeelden hiervan zijn de *First Fit Decreasing (FFD)* en *Best Fit Decreasing* heuristieken die gebaseerd zijn op gulzige algoritmen [**?**] of het *Reordering Grouping Algorithm (RGGA)* gebaseerd op genetische algoritmen [**?**]. Daarnaast zijn er ook algoritmen om dynamisch een pool van resources te alloceren aan een set van concurrerende VM's en algoritmen die rekening houden met beperkingen opgelegd door de cloud-gebruikers.
Live migration [**?**] of dynamische plaatsing van virtuele machines is een zeer handige techniek waarbij een draaiende VM wordt gepauzeerd, geserialiseerd en verplaatst naar een andere fysieke machine. Gebruikte heuristieken hiervoor zijn bijvoorbeeld de *first-fit* heuristiek [**?**] dat de verschillende VM's dynamisch hermapt op de fysieke machine's met als resultaat een minimaal gebruik van fysieke machines, of *King-fisher* [**?**], een set van technieken voor VM-schaling, replicatie en migratie waarbij de problemen worden geformuleerd als ILP-model. Een ander voorbeeld is het live migration proces voorstellen als een roman zodat het probleem herleid wordt tot een *Stable Marriage* probleem [**?**].

Er zijn ook algoritmen die rekening houden met de netwerktopologie vooraleer een VM wordt geplaatst op een fysieke machine. Een voorbeeld hiervan zijn heuristieken die VM's die data-intensief met elkaar moeten communiceren dicht bij elkaar trachten te plaatsen.

Technieken om het energieverbuik te minderen zijn gebaseerd op *right-sizing* van datacenters.[1] [**?**] Nieuwere methoden maken een combinatie van deze technieken om het energieverbuik te minderen samen met de mogelijkheid tot het aanpassen van de processorsnelheid van fysieke nodes met behulp van *Dynamic Voltage Scaling* (DVS). Dit zorgt ervoor dat de kloksnelheid van een processor kan dalen (minder stroomverbruik) en een processor kan werken in een omgeving met hogere temperaturen (minder koeling) wat in beide gevallen zorgt voor een lagere energieconsumptie.

Naast de globale planning van gevirtualiseerde resources moet er ook lokale planning van de gevirtualiseerde resources gebeuren. Lokaal resource management bepaalt hoe de fysieke resources worden gedeeld tussen de gevirtualiseerde resources die erop draaien. Recente onderzoeken proberen hiervoor een volledige automatische oplossing te vinden die constant de VM's monitort waarbij dynamisch extra resources voorzien worden, rekening houdend met de afgesloten SLA's. Technieken die hiervoor gebruikt worden zijn onder andere *fuzzy-logic* [**?**] en *reinforcement*

[1] Het dynamisch herschalen van actieve fysieke nodes bij een veranderende werklast.

learning [**?**]. Een voorbeeld van een systeem dat het delen van lokale fysieke resources tussen virtuele resources vergemakkelijkt is *DeepDive* [**?**], een systeem ontwikkeld om interferentie tussen verschillende VM's te identificeren en te verzachten.

Tot slot zijn er nog tal van andere technieken om resources te alloceren rekening houdend met bijvoorbeeld het profiel van de vraag, de schaalbaarheid van de applicatie, etc. maar hier wordt niet dieper op ingegaan.

In Tabel **??** wordt een overzicht weergegeven van een aantal ontwikkelde resource allocatieschema's tussen 2012 en 2015. De verschillende kolommen beschrijven respectievelijk de naam van het algoritme of de naam van de ontwikkelaar, het jaartal van de publicatie, het type waarop het schema is gebaseerd, of het een algoritme, framework of protocol is, wat de invoerparameters zijn, wat de uitvoerparameters zijn en op welke manier het getest is.

Tabel 3.1: Overzicht resource-allocatieschema's
A=Algoritme, P=Protocol, F=Framework S=Simulator, C=Cloud, ILP=Integer Linair Programming, GH = Greedy Heuristic, SBP=Stochastic Bin Packing, MINLP=Particle swarm, RR=Round-Robin, SA=Simulated Annealing, SMT=Satisfiabilty Module Theory, FFD=First Fit Decreasing, MI(N)=Mixed Integer (Non-), SPLE=Software Product Line Engineering, L=Lijst, G=Graaf, B=Boom, FN=Fysieke node, Mig=Migraties, (?)=Niet vermeld

Naam	Jaar	Type	A | F | P	Invoer	Uitvoer	Getest
Alicherry et al. [**?**]	2012	k-sneden	A	G	par{G}	S
MCRVMP [**?**]	2012	ILP & GH	A	B{netwerk}	VM-plaatsing	C
Breitgand et al. [**?**]	2012	SBP	A	L{VM}	L{VM}/Bin	S
Snooze [**?**]	2012	Framework	F	-	-	C
Sequence planning [**?**]	2012	Heuristiek	A	Net + S{Mig}	L{VM/Mig}	S
Giurgu et al. [**?**]	2012	Beam seach	A	-	VNI-plaatsing	S
Seagull [**?**]	2012	Framework	F	-	-	C
v-Bundle [**?**]	2012	Novel	A	B{Node}	VM-plaatsing	S + C
VMPR [**?**]	2012	Markov-ketens	A	L{Jobs}	Job-afhandeling	S (?)
TROPIC [**?**]	2012	Framework	F	-	-	C
Konstanteli et al. [**?**]	2012	MINLP	A	-	Gealloceerde resources	C
CloudMap [**?**]	2012	RR	A	Resource-verbruik	Servercluster	C
VirtualKnotter [**?**]	2012	SA	A	Verkeer	VM-plaatsing	S
P* [**?**]	2012	Gossip protocol	P	-	-	S
Scattered [**?**]	2012	Black- en Gray-box	A	L{hosts}	VM-plaatsing	C
VMM-Planner [**?**]	2013	SMT	A	VM-plaatsing + doel	L{Mig}	C
Shi et al. [**?**]	2013	ILP	A	L{FN}	VM-plaatsing	C
Shi et al. [**?**]	2013	FFD	A	L{FN}	VM-plaatsing	C
Omega [**?**]	2013	Large scale	F	-	-	S
PACMan [**?**]	2013	-	A	VM interferentie	VM-plaatsing	C
Max-BRU [**?**]	2014	-	A	Datacenter	VM-allocatie	C
RT-OpenStack [**?**]	2015	-	A	Budget VCPU's	CPU-resources	C
J.Bi et al. [**?**]	2015	SA, MINLP	A	Datacenter	VM-allocatie	S
F. Fakhfakh et al. [**?**]	2015	MILP	A	L{Taken, deadl.}	VM-allocatie	S
L. Li et al. [**?**]	2015	Framework	F	-	-	(C)
A. Ruiz-Alvarez et al. [**?**]	2015	IPL	A	Max. duur	Optimale kost	C
Dohko [**?**]	2015	SPLE	A	Vereisten	Resource selectie	S
A.aral et al. [**?**]	2015	LAD	A	Gewenste topologie	Topologie	S
K. Metwally et al. [**?**]	2015	MILP	A	Datacenter	VRP + resource allocatie	C

4

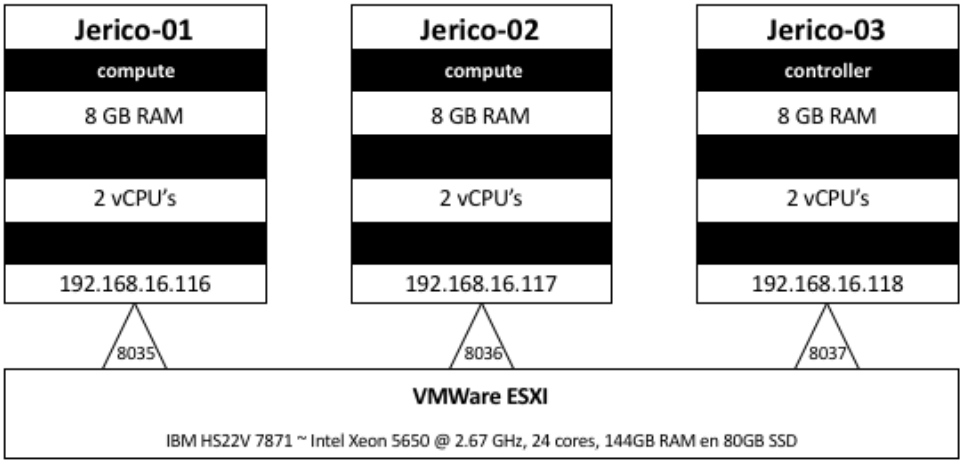
Overzicht van de testomgeving

De testomgeving om OpenStack op uit te rollen bestaat uit 3 Ubuntu Server 16.04.02 LTS systemen met elk 8 GB vRAM, 2 vCPU's met 4 kernen en 10 GB HDD opslag. Deze 3 systemen worden gevirtualiseerd met behulp van VMWare ESXI 5.5 update 2 op 1 fysieke server. Via drie SSH-verbindingen kunnen de systemen aangestuurd worden. Figuur **??** geeft een schematisch overzicht weer van de drie systemen.

De OpenStack-omgeving bestaat uit 1 controller node en 2 compute nodes. De node *jerico-03* doet dienst als controller node waarbij alle nodige services van OpenStack zijn geactiveerd. De nodes jerico-02 en jerico-03 doen dienst als compute-nodes en enkel de hiervoor nodige services zijn geactiveerd. Een overzicht van de actieve services per node bevindt zich in Tabel **??**. Een overzicht van alle services met bijhorende uitleg bevinden zich in Bijlage B.

Tabel 4.1: Overzicht van de actieve services per node

Host	KeyStone	Glance	Neutron	Nova	Cinder	Horizon	Heat	Ceilometer	Aodh
jerico-03	✓	✓	✓	✓	✓	✓	✓	✓	✓
jerico-02			q-agt	n-cpu	c-vol			✓	✓
jerico-01			q-agt	n-cpu	c-vol			✓	✓



Figuur 4.1: Overzicht van de testomgeving

4.1 DevStack

OpenStack kan op verschillende manieren worden uitgerold zoals met Autopilot, conjure-up, alsook via DevStack. DevStack [**?**] is een collectie van *bash scripts* ontwikkeld door de community van OpenStack zelf dat volledig automatisch, mits enkele voorafgaande instellingen, OpenStack met al zijn componenten installeert en configureert. In een *local.conf* bestand kunnen alle instellingen worden gewijzigd zoals de verschillende wachtwoorden, de IP-adressen, welke services actief moeten zijn, etc. Een aangemaakte stack gebruiker met *sudo*-permissies kan het script uitvoeren en zo OpenStack volledig installeren op het desbetreffende systeem. Bijlage A bevat een stap-voor-stap installatiehandleiding van OpenStack met behulp van DevStack.

Aangezien DevStack nog volop in ontwikkeling is (de huidige versie is 0.0.1[1]), zijn er een aantal gekende problemen. Het grootste probleem is dat OpenStack niet meer naar behoren zal werken na een heropstart van het systeem met als mogelijke oorzaak het niet opnieuw automatisch initialiseren van de daarvoor gebruikte *screens* van DevStack in Ubuntu. Hierdoor worden bepaalde services niet meer gestart en is de enige mogelijkheid het opnieuw uitvoeren van het *stack.sh* bestand als stack-gebruiker. Hierbij wordt alles gewist, van draaiende instanties in OpenStack tot de volledige databank die OpenStack gebruikt wat in de meeste gevallen leidt tot ongemakken. Een 'echte' oplossing voor dit probleem bestaat nog niet, maar de ontwikkeling hiervan is bezig. Een eventuele optie om dit op te lossen zijn volgende commando's:

```
$ sudo su stack
$ sudo chown stack:stack `readlink /proc/self/fd/0`
$ screen -c /devstack/stack-screenrc
```

[1] 22 maart 2017

Listing 1: Heropstart van DevStack na systeemherstart

Dit zou de screens terug moeten koppelen aan Ubuntu en de services van OpenStack terug starten aan de hand van de services die voor de heropstart aan het draaien waren. Uitgebreide testen of dit al dan niet werkt moeten in de toekomst nog uitgevoerd worden.

Daarnaast gebeuren er bijna dagelijks *commits* met verschillende oplossingen waardoor er hoogstwaarschijnlijk verschillende bugs aanwezig zijn in de gebruikte testomgeving.

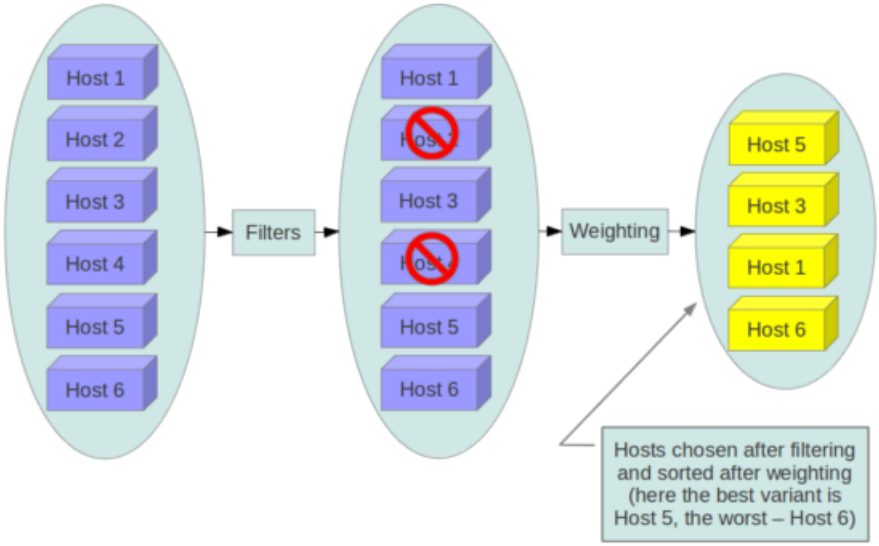
4.2 Nova scheduler

OpenStack levert Nova af met een standaard *filter and weighting*-scheduler die zal bepalen op welke host, ook wel *hypervisor* genoemd, de nieuwe instantie geplaatst wordt. Het plaatsingsproces gebeurt in twee stappen. Eerst gaan alle beschikbare hypervisors door verschillende filters om te bepalen of de hypervisor de instantie kan hosten. Voorbeelden van standaard meegeleverde filters zijn de RamFilter, DiskFilter, AllHostFilter, en nog vele anderen. In het configuratiebestand van Nova (standaard *nova.conf in /etc/nova/*) bevindt er zich een lijst met alle filters die gebruikt worden tijdens het verkiezingsproces. Zo zal de RamFilter alleen de hypervisors toelaten die voldoende RAM-geheugen ter beschikking hebben terwijl de AllHostFilter alle hypervisors zal toelaten, ongeacht deze plaats hebben voor de nieuwe instantie of niet. Vervolgens worden de toegelaten instanties gesorteerd volgens een bepaald gewicht waardoor de beste hypervisor de nieuwe instantie mag hosten. Figuur **??** geeft dit proces schematisch weer.

4.2.1 Round Robin scheduler

Zoals beschreven in **??** zal er een proof-of-concept (PoC) worden uitgevoerd om de evaluatie van een resource allocatieschema aan te tonen. Hier is gekozen voor het relatief eenvoudig Round Robin schema, wat een nieuwe instantie steeds op de volgende hypervisor in de rij zal plaatsen. Een hypervisor waarbij net een nieuwe instantie is geplaatst zal zich achteraan in de rij bevinden. Concreet, de eerste instantie zal gehost worden op jerico-01, de tweede op jerico-02, de derde op jerico-03, de vierde op jerico-01 en zo verder.

Aangezien OpenStack en al zijn componenten open-source zijn, biedt het de optie om eigen nova-filters en -weighters te implementeren. Hieraan zijn wel enkele voorwaarden verbonden. Zo moet een custom-filter steeds overerven van *filters* die zich in het *Nova.scheduler*-pakket bevinden (ook wel de *BaseHostFilter* genaamd). Bovendien moet de methode *host_passes* geïmplementeerd worden dat de waarde *True* als resultaat teruggeeft indien de hypervisor geschikt is om de instantie te hosten, en *False* indien de hypervisor niet geschikt is voor het hosten van de instantie.



Figuur 4.2: Werking van de Nova-scheduler [**?**]

Om het Round Robin algoritme eenvoudig in OpenStack te kunnen implementeren, is hiervoor een *RoundRobinFilter* ontwikkeld. Deze laat enkel de host, die als eerste in de virtuele rij staat, toe om de nieuwe instantie te hosten. De hiervoor gebruikte code bevindt zich hieronder.

```
from oslo_log import log as logging
from nova.scheduler import filters
LOG = logging.getLogger(__name__)
counter = 0
counter_check = 0
hypervisors = ["jerico-01", "jerico-02", "jerico-03"]

class RoundRobinFilter(filters.BaseHostFilter):
    run_filter_once_per_request = True

    def host_passes(self, host_state, spec_obj):
                global counter
                global counter_check
                global hypervisors
                counter_check += 1
                if counter_check == 4:
                        counter += 1
                        counter_check = 1
```

```
LOG.debug("Counter is %d" % counter)
LOG.debug("Host_state name %s" % host_state.host)
if host_state.host == hypervisors[counter % 3]:
        LOG.debug("SELECTED HOST is %s" % host_state.host)
        return True
else: return False
```

Listing 2: RoundRobinFilter

Bij iedere nieuwe aanmaak van een instantie zal elke hypervisor door deze filter passeren. Enkel de hypervisor die als eerste in de virtuele rij staat, bepaald door de *counter*, zal True als resultaat van de methode teruggeven en uiteindelijk de nieuwe instantie hosten.

4.2.2 Inpluggen van de Round Robin Scheduler

Om de Round Robin scheduler te activeren in de gebruikte OpenStack omgeving moeten volgende stappen gebeuren:

1. Bewaar bovenstaande code in /opt/stack/nova/nova/scheduler/filters/round_robin_filter.py (naam vrij te kiezen)

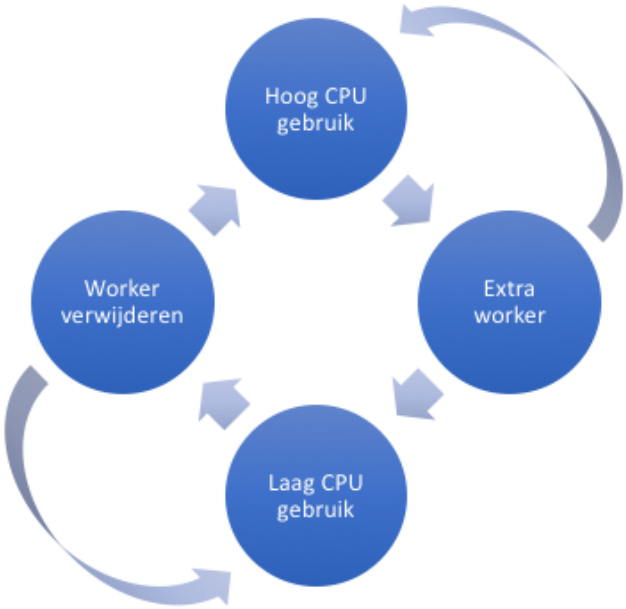
2. Pas het /etc/nova/nova.conf bestand aan zodat in de [DEFAULT]-sectie de lijst van *scheduler_default_filters* vervangen wordt door *RoundRobinFilter* (naam van de klasse)

3. Controleer ook in dit bestand of de *scheduler_driver* staat ingesteld op *filter_scheduler*

4. Als laatste moet de Nova-scheduler worden herstart. Gebruik hiervoor het commando *$ screen -x stack* om de screens te betreden, navigeer vervolgens naar het *n-sch* screen en herstart deze service door deze te stoppen (ctrl + c) en terug te starten (pijl naar boven + enter)

Het gebruik van de screens van OpenStack staat uitgelegd in Bijlage A.

Zoals te zien is in het *nova.conf*-bestand, zijn er verschillende mogelijkheden om de scheduler van Nova te wijzigen. In dit geval is er voor een relatief eenvoudige optie gekozen om een nieuw allocatieschema te gebruiken. Meer complexere allocatieschema's zullen ook aanpassingen van de scheduler_driver vereisen voor een correcte werking maar daar wordt niet dieper op ingegaan.



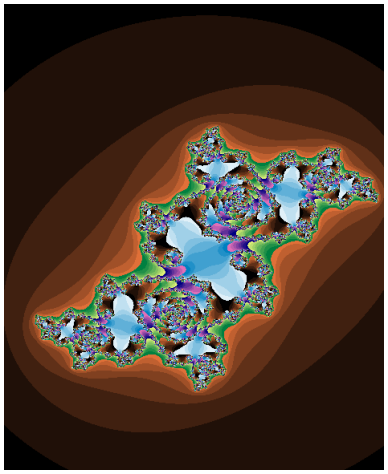
Figuur 4.3: FaaFo: het schaalscenario

4.3 FaaFo - First App Application For OpenStack

Een cloud-applicatie is optimaal indien het de mogelijkheid biedt om te schalen. FaaFo [**?**] is een OpenStack-applicatie die fractalen, voorbeeld in Figuur **??**, van een bepaalde grootte berekent. FaaFo is ontwikkeld door OpenStack om horizontaal uit te schalen. In dit onderzoek is FaaFo uitgerold op minimaal twee nodes. De eerste is de controller-node, die de databank en de queue beheert en de verschillende workers opdrachten kan geven om fractalen te berekenen. De twee node is bijgevolg een worker-node die een fractaal uit de queue van de controller-node zal berekenen en het resultaat zal bewaren in de databank op de controller-node.

De mogelijkheid om te schalen binnen deze applicatie komt door de goede opsplitsing van de verschillende taken in *microservices* en door gebruik te maken van een queue waar de fractalen, die nog berekend moeten worden, in worden bewaard. Een optimaal scenario om te schalen zou als volgt verlopen: indien de enige worker veel CPU-kracht gebruikt binnen een bepaalde periode, dan moet er een tweede worker worden aangemaakt die de eerste worker zal helpen met al het werk. Blijken deze twee samen dan nog veel CPU-kracht te gebruiken, dan wordt een eventuele derde, vierde, ... worker aangemaakt. Indien er meer dan twee workers actief zijn en indien de queue leeg is, dan moeten de overbodige workers worden verwijderd. Dit hele proces wordt eveneens verduidelijkt in Figuur **??**.

Het scenario is sterk gerelateerd aan de werkelijke gang van zaken binnen cloud computing. Zoals beschreven in Sectie **??** en in Sectie **??** moet een cloud-gebruiker enerzijds schalen zodat



Figuur 4.4: Voorbeeld van een fractaal

de SLA tegenover hun eindgebruikers wordt voldaan en anderzijds de kosten voor het gebruik van de resources van de cloud provider worden geminimaliseerd. Doordat de applicatie bij hoge werklast horizontaal zal uitschalen, kan de SLA tegenover de eindgebruiker worden voldaan (de fractaal zal snel berekend worden) en bij te weinig werk zal de applicatie horizontaal krimpen zodat er minder resources gebruikt worden en er minder kosten zijn aan de cloud-provider.

4.3.1 Schalen binnen OpenStack

Zoals vermeld in Sectie **??** bevat OpenStack verschillende componenten, waaronder ook een aantal die het mogelijk maken om applicaties automatisch te schalen. De combinatie van de Orchestration- [**?**] en de Telemetry-services [**?**] zorgt ervoor dat automatisch schalen mogelijk wordt. *Heat* biedt de mogelijkheid om meer of minder instanties aan te maken afhankelijk van de vraag. *Ceilometer* en *Aodh* zorgen voor de monitoring van bepaalde instanties om zo te bepalen of het wel nuttig is om te schalen. Aan de hand van alarmen zullen deze twee componenten Heat op de hoogte brengen van de situatie van de verschillende instanties, waarop Heat op zijn beurt zal reageren door het aanmaken van een nieuwe instantie, een instantie te verwijderen of het alarm te negeren.

Het gehele gebeuren van alarmen, schalen, etc. gebeurt in OpenStack met een *stack* [**?**]. Een stack is een verzameling van OpenStack resources zoals instanties, IP-adressen, volumes, gebruikers, etc die gecreëerd worden met een *template*. Zulke template, geschreven in OpenStack Heat Orchestration Template (HOT) [**?**] formaat of Amazon Web Services (AWS) formaat, is een beschrijving van de gebruikte instanties, gebruikers, images, alarmen, etc.

4.3.2 FaaFo met OpenStack SDK - Libcloud

Een eerste mogelijkheid om FaaFo te configureren en installeren is met behulp van LibCloud [**?**], een Python-bibliotheek voor interactie met verschillende cloud service providers. Deze werd reeds kort aangehaald in Sectie **??**. De code is grotendeels gebaseerd op een handleiding van OpenStack zelf [**?**], mits enkele kleine aanpassingen.

De code bestaat uit verschillende stappen en is beschikbaar via GitHub.[2] Na het initialiseren van de cloud provider en indien er een connectie is gemaakt kan de eerste stap beginnen. Deze print alle images alsook alle flavors uit. Ook zal er een specifieke image en flavor geselecteerd worden voor verder gebruik. De tweede stap gaat een instantie maken, alle instanties printen en dan deze instantie terug verwijderen. In de derde stap gebeurt een configuratie die helpt om de werking van OpenStack zelf te begrijpen. Hierin wordt er eerst een sleutelpaar aangemaakt met de naam *demokey* indien deze nog niet bestaat. Dit sleutelpaar is nodig om een SSH-verbinding met de instantie mogelijk te maken. Enkel de gebruikers die dezelfde sleutel bevatten als diegene waarmee de instantie is geïnitialiseerd, zullen een SSH-verbinding kunnen maken. Naast een sleutelpaar worden er ook enkele *security groups* aangemaakt die dienst doen als een soort *firewall*. Standaard blokkeren de instanties alle inkomende en uitgaande verbindingen[3] waardoor er nood is aan het toelaten van enkele connecties zoals bijvoorbeeld een SSH-connectie (poort 22) vanaf de OpenStack-controller. In de vierde stap wordt met de vorige instellingen en configuratie een instantie gestart die de FaaFo-applicatie zal uitvoeren (all-in-one). De volgende stappen zijn gelijkaardig zodat de FaaFo-applicatie wordt opgesplitst in een FaaFo-API voor de API-services, een FaaFo-controller voor de arbitrage en een FaaFo-worker voor de eigenlijke berekeningen. Hier wordt gebruik gemaakt van een *floating ip*, nodig om een externe connectie, bijvoorbeeld een SSH-verbinding, te maken met de instantie. De standaard toegewezen IP-adressen zijn enkel bruikbaar voor de communicatie tussen de instanties zelf.

De FaaFo-applicatie heeft via LibCloud niet de mogelijkheid om te schalen waardoor het niet bruikbaar is voor de Proof-of-Concept. Toch is dit hele proces geen verloren zaak geweest doordat de werking van OpenStack duidelijker is geworden en de derde stap nog gebruikt wordt voor de configuratie van het sleutelpaar.

4.3.3 FaaFo & OpenStack-Orchestration: een schaalbare cloudapplicatie

Om een schaalbare FaaFo-applicatie uit te rollen moet er gebruik gemaakt worden van een HOT-template of een AWS-template. In deze thesis is gekozen voor het HOT-formaat omdat het

[2] `https://github.ugent.be/jfmoeyer/EvaluationRASOpenStack/blob/master/application/firstapplication.py`

[3] In firewall-termen: de default policy is block/discard

onderdeel is van OpenStack en omdat er voldoende informatie beschikbaar is. In onderstaande paragrafen bevinden zich delen van de code met de nodige informatie. De volledige code is te vinden op Github.[4]

De volledige template bevat twee grote delen, namelijk de parameters en de resources. De parameters, weergegeven in Listing **??**, beschrijven de variabelen die worden meegegeven door de gebruiker. Zo zal de *key_name* bepalen welke SSH-sleutel gebruikt wordt voor de toegang tot de instanties zodat niet iedereen toegang heeft. De *flavor* beschrijft welke hardware de instantie ter beschikking heeft en de *image_id* beschrijft het besturingssysteem voor de instantie. De laatste drie parameters beschrijven de periode voor het monitoren en de gebruikte scripts zodat de nieuwe instanties bepaalde code automatisch kunnen uitvoeren. Deze 3 parameters zijn optioneel aangezien er een *default*-waarde aan is toegekend.

```
heat_template_version: 2014-10-16
description: |
A template that starts auto-scaling workers for the faafo application

parameters:
  key_name:
    type: string
    description: Name of the keypair to enable ssh
    default: id_rsa
    constraints:
      - custom_constraint: nova.keypair
      description: Must already exist on the cloud

  flavor:
    type: string
    description: The flavor that the application uses
    constraints:
      - custom_constraint: nova.flavor
      description: Must be a valid flavor provided by the cloud provider.

  image_id:
    type: string
    description: The ID fo the image for the creation of the instances.
    constraints:
      - custom_constraint: glance.image
      description: Must be a valid image on your cloud

  period:
    type: number
    description: The period to use to calculate the ceilometer statistics (in seconds)
    default: 60
```

[4] https://github.ugent.be/jfmoeyer/EvaluationRASOpenStack/blob/master/application/autoscaling_workers.yaml

```
  worker_source:
    type: string
    description: The location of the installation script for the workers
    default: https://raw.githubusercontent.com/moeyerke/nodejs-agent/master/userdata_workers.sh

  controller_source:
    type: string
    description: The location of the installation script for the controller
    default: https://raw.githubusercontent.com/moeyerke/nodejs-agent/master/userdata_controller.sh
```

Listing 3: FaaFo-template: de parameters

De resources-sectie bevat de eigenlijke essentie van de template. Deze bestaat uit de *security groups*, welke vergelijkbaar zijn met een firewall, servers, *scaling groups* en alarmen. Als eerste worden twee *security groups* aangemaakt, één voor de worker-instanties en één voor de controller-instantie, beiden weergegeven in Listing **??**. De *security group* van de worker laat enkel een SSH-connectie (poort 22) toe vanuit jerico-03, de controller-node van OpenStack. De security group van de FaaFo controller-instantie is uitgebreider. Hier wordt naast het toelaten van een SSH-verbinding vanuit jerico-03 ook een HTTP-verbinding toegestaan en toegang voor de worker-instanties tot poort 5672 (gebruikt door de queue).

```
resources:
  worker:
    type: OS::Neutron::SecurityGroup
    properties:
      description: "Enables ssh to worker node"
      rules: [
        {remote_ip_prefix: 0.0.0.0/0,
        protocol: tcp,
        port_range_min: 22,
        port_range_max: 22},]

  controller:
    type: OS::Neutron::SecurityGroup
    properties:
      description: "For services that run on a control node"
      rules: [
        {remote_ip_prefix: 0.0.0.0/0,
        protocol: tcp,
        port_range_min: 5672,
        port_range_max: 5672,
```

```
      remote_mode: remote_group_id,
      remote_group_id: { get_resource: worker } },
      {remote_ip_prefix: 0.0.0.0/0,
      protocol: tcp,
      port_range_min: 80,
      port_range_max: 80},
      {remote_ip_prefix: 0.0.0.0/0,
      protocol: tcp,
      port_range_min: 22,
      port_range_max: 22},
    ]
```

Listing 4: FaaFo-template: de security groups

Vervolgens wordt er een Nova-server, de FaaFo controller-instantie aangemaakt met de meegegeven *key_name*, *flavor* en *image*. Het aanmaken van de controller-instantie bevindt zich in Listing **??**. De *user_data* beschrijft het commando dat wordt uitgevoerd na het opstarten van de instantie. In Bijlage C worden de scripts toegelicht.

```
controller_instance:
  type: OS::Nova::Server
  properties:
    key_name: { get_param: key_name }
    image: { get_param: image_id }
    flavor: { get_param: flavor }
    name: app-controller
    security_groups:
      - {get_resource: controller}
    user_data_format: RAW
    user_data:
      str_replace:
        template: |
          #!/usr/bin/env bash
          curl -L -s controller_source | bash
        params:
          controller_source: { get_param: controller_source }
```

Listing 5: FaaFo-template: de controller-instantie

Naast de FaaFo controller-instantie moeten er ook worker-instanties worden aangemaakt, weergegeven in Listing **??**. Omdat deze het meeste werk zullen verrichten (en dus veel CPU-resources

zullen verbuiken), moeten deze kunnen horizontaal schalen. De *AutoScalingGroup* beschrijft de worker-instantie met dezelfde parameters als de controller-instantie (met uitzondering van de *security group* en de *user_data*-parameter). Belangrijke parameters bij de *AutoScalingGroup* zijn de onderste drie, namelijk *min_size, desired_capacity en max_size*, die bepalen hoeveel worker-instanties actief kunnen zijn, en aan hoeveel actieve instanties de voorkeur gegeven wordt.

```
worker_auto_scaling_group:
  #The worker instances are managed by this auto_scaling group
  type: OS::Heat::AutoScalingGroup
  properties:
    resource:
      type: OS::Nova::Server
      properties:
        key_name: { get_param: key_name }
        image: { get_param: image_id }
        flavor: { get_param: flavor }
        name: worker
        security_groups:
          - {get_resource: worker}
        user_data_format: RAW
        user_data:
          str_replace:
            template: |
              #!/usr/bin/env bash
              curl -L -s worker_source | bash -s -- controller_ip1
              ↪ controller_ip2
            params:
              controller_ip1: { get_attr: [controller_instance, networks,
              ↪ private, 0] }
              controller_ip2: { get_attr: [controller_instance, networks,
              ↪ private, 1] }
              worker_source: { get_param: worker_source }
    min_size: 1
    desired_capacity: 1
    max_size: 7
```

Listing 6: FaaFo-template: de AutoScalinGroup

Nadien worden er in Listing **??** twee *ScalingPolicies* gedefinieerd, de ene om omhoog te schalen, de andere om omlaag te schalen. Hierbij wordt telkens vermeld welke instanties er moeten

bijgemaakt of verwijderd worden en met hoeveel tegelijkertijd. In deze template gaat het dus over de worker-instanties die telkens met 1 instantie worden verhoogd of verlaagd.

```
scale_up_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    adjustment_type: change_in_capacity
    auto_scaling_group_id: {get_resource: worker_auto_scaling_group}
    cooldown: { get_param: period }
    scaling_adjustment: 1

scale_down_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    adjustment_type: change_in_capacity
    auto_scaling_group_id: {get_resource: worker_auto_scaling_group}
    cooldown: { get_param: period }
    scaling_adjustment: '-1'
```

Listing 7: FaaFo-template: de ScalingPolicies

Als laatste worden er twee alarmen geconfigureerd in Listing **??**, één indien er zeer veel activiteit op de worker-instanties is en één indien er weinig activiteit is. De alarmen gaan in dit geval het gebruik van de CPU monitoren gedurende een bepaalde periode (standaard 60s) en indien de *treshold* overschreden wordt, zal het alarm een *ScalePolicy* triggeren die op zijn beurt een extra worker-instantie zal toevoegen of verwijderen.

```
cpu_alarm_high:
  type: OS::Aodh::Alarm
  properties:
    description: Scale-up if the average CPU > 90 % for period of seconds
    meter_name: cpu_util
    statistic: avg
    period: { get_param: period }
    evaluation_periods: 1
    threshold: 90
    alarm_actions:
      - {get_attr: [scale_up_policy, alarm_url]}
    comparison_operator: gt
```

```
  cpu_alarm_low:
    type: OS::Aodh::Alarm
    properties:
      description: Scale-down if the average CPU < 15 % for period of seconds
      meter_name: cpu_util
      statistic: avg
      period: { get_param: period }
      evaluation_periods: 1
      threshold: 15
      alarm_actions:
        - {get_attr: [scale_down_policy, alarm_url]}
      comparison_operator: lt
```

Listing 8: FaaFo-template: de alarmen

4.3.4 Problemen met de applicatie

Desondanks dat de FaaFo-applicatie ontwikkeld is door OpenStack waren er nog een hele reeks problemen die opgelost moesten worden vooraleer het hele gebeuren correct werkte. Deze problemen worden hier kort toegelicht samen met de eventuele oplossing.

Het grootste probleem met de FaaFo-applicatie had te maken met de manier waarop deze wordt geïmplementeerd in bovenstaande Listings en de werking van Ceilometer/Aodh. Door in de configuratie van de alarmen geen *query* of metadata mee te geven baseert Ceilometer zich op metingen van alle actieve instanties. Dit heeft als gevolg dat de applicatie op sommige ogenblikken een worker zal verwijderen omdat het *cpu_alarm_low* wordt getriggerd door de lezing van een instantie die totaal niets met de applicatie heeft te maken. OpenStack heeft hiervoor een oplossing beschreven in [**?**]. Bij de alarmen wordt een *matching_metadata* geconfigureerd en bij de *AutoScalingGroup* wordt er metadata meegegeven. Deze configuratie moet ervoor zorgen dat Ceilometer enkel de instanties monitort met dezelfde metadata, maar na het uittesten hiervan blijkt dit niet te werken door verschillende updates van OpenStack, Ceilometer en Aodh. Ceilometer houdt in dit geval nu over niets meer toezicht en bijgevolg wordt er ook niet geschaald. Een werkende, maar niet per se een betere, oplossing is gebaseerd op voorgaande configuratie maar configureert in de alarmen nu een *query*, die controleert of de naam van de instantie gelijk is aan 'worker'. Bij de *AutoScalingGroup* wordt als metadata dan de naam worker meegegeven zodat enkel deze instanties invloed zullen hebben op het schalen. In onderstaande Listing bevindt zich de aanvulling op bovenstaande Listings met de werkende metadata- en query-instellingen:

```
...
resources:
```

```
worker_auto_scaling_group:
  ...
  properties:
    resource:
      ...
      properties:
        ...
        metadata: { "display_name": "worker" }
...
cpu_alarm_high:
  ...
  properties:
    ...
    query:
      - field: metdata.display_name
        op: eq
        value: 'worker'

cpu_alarm_low:
  ...
  properties:
    ...
    query:
      - field: metadata.display_name
        op: eq
        value: 'worker'
...
```

Listing 9: FaaFo template met metadata

Een tweede probleem had te maken met de beveiliging van RabbitMQ, de *queue* die gebruikt wordt voor de communicatie tussen de FaaFo-controller en de FaaFo-workers. Standaard gebruikt FaaFo de login-gegevens *guest:guest* om te communiceren met de *queue* op de FaaFo-controller. Door een eerdere update van RabbitMQ (sinds versie 3.3.0)[5] zijn deze gegevens enkel bruikbaar vanaf de lokale host, en niet meer vanaf een externe host. Om dit probleem op te lossen wordt er bij de FaaFo-controller in de *user_data* een nieuwe RabbitMQ-gebruiker aangemaakt die alle permissies krijgt vanop eender welke host. Hierdoor is communicatie wel mogelijk tussen de FaaFo-controller en de FaaFo-workers. De gebruikte code bevindt zich in Bijlage C.

[5] `http://www.rabbitmq.com/release-notes/README-3.3.0.txt`

Na het oplossen van bovenstaande twee problemen werkt de applicatie grotendeels. Er is nog één probleem dat meer tijd en kennis vereiste om op te lossen en waarmee rekening moet gehouden worden tijdens de evaluaties van de allocatieschema's. Het berekenen van een fractaal kan een bepaalde tijd in beslag nemen met een hoog CPU-verbruik. Indien er meerdere worker-instanties actief zijn waarvan er één nog geen fractaal aan het bereken is en Ceilometer net die instantie monitort, kan het zijn dat er een worker-instantie wordt verwijderd omdat het alarm van laag CPU-gebruik afgaat. Is de verwijderde worker bezig met het berekenen van een fractaal, dan verdwijnt dit fractaal mee met de instantie waardoor het nooit volledig berekend zal worden. In deze context is dit probleem niet erg omdat er vooral wordt gekeken naar waar de instanties geplaatst worden, maar indien een applicatie cruciale berekeningen moet uitvoeren, mogen deze berekeningen natuurlijk niet verdwijnen.

Een mogelijke oplossing voor dit laatste probleem is de FaaFo-applicatie zo ontwikkelen dat een instantie pas verwijderd kan worden indien alle berekeningen op die specifieke instantie voltooid en doorgestuurd zijn naar de FaaFo-controller.

5 Monitoring

Nu de gehele testomgeving geconfigureerd en werkende is, dient het hele gebeuren gemonitord worden. Dit is nodig om verschillende gegevens te verzamelen zodat er nadien kan worden overgegaan tot het evalueren van het allocatieschema in de proof-of-concept. In dit hoofdstuk wordt eerst dieper ingegaan op één specifieke tool, namelijk Rally [**?**], dat de deployment van OpenStack kan verifiëren. Vervolgens zal er een overzicht gegeven worden van de verschillende monitortechnieken waarna er ten slotte dieper wordt ingegaan op de gebruikte monitortechniek.

5.1 Rally: evalueren schaalbaarheid OpenStack

Rally [**?**] is een benchmarking tool dat een OpenStack deployment kan automatiseren, de uitgerolde cloud kan verifiëren, benchmark testen kan uitvoeren en de cloud profileren. Het biedt daardoor een antwoord op de vraag hoe OpenStack werkt op grote schaal. Rally wordt gebruikt in drie high-level use cases die zeer interessant zijn om de deployment van OpenStack te evalueren. Een eerste use case is de mogelijkheid om een OpenStack omgeving te deployen, te testen en afhankelijk van de testresultaten OpenStack opnieuw te deployen met een andere configuratie tot de testen succesvol zijn. De tweede use case is bedoeld voor DevOps waarbij een bestaande cloud getest wordt met een simulatie van echte gebruikerswerklast waarna de resultaten worden weergegeven in allerhande grafieken. Een interessante eigenschap hierbij is dat SLA's gedefinieerd

kunnen worden zodat hiermee rekening wordt gehouden tijdens het uitvoeren van de testen. De laatste use case is Rally CI/CD[1], waarbij OpenStack wordt uitgerold op specifieke hardware met de laatste versie van een eigen geschreven tool. Hierop worden dan testen uitgevoerd die nadien gedeeld worden met de OpenStack community om zo OpenStack te verbeteren in de toekomst.

Rally maakt gebruik van 2 types invoerbestanden, namelijk een standaard JSON-bestand, of een YAML-bestand, een template-syntax gebaseerd op Jinja2[2]. Elk invoerbestand bevat dan één of meerdere scenario's, al dan niet met meerdere configuraties per scenario. De configuratie van een scenario bevat steeds een lijst met argumenten (args) zoals de te gebruiken flavor, image, parameters, etc., de context die de gesimuleerde gebruikers representeert en een runner die bepaalt hoe de test wordt uitgevoerd. Optioneel kan er een nog een SLA, zoals bijvoorbeeld maximum aantal seconden per iteratie en de *failure rate*, toegevoegd worden dat de eigenlijke succescriteria omschrijft.

Een voorbeeld om Rally beter te leren begrijpen maakt gebruik van een *sample* invoerbestand en toont het verschil tussen een *all-in-one* OpenStack deployment en een multi-node OpenStack deployment. Het invoerbestand ziet eruit als volgt:

```
{ % set flavor_name = flavor_name or "m1.tiny" %}
{  "NovaServers.boot_and_delete_server": [{
    "args": {
    "flavor": {
      "name": "{{flavor_name}}"
    },
    "image": {
      "name": "^cirros.*-disk$"
    },
    "force_delete": false
  },
  "runner": {
    "type": "constant",
    "times": 10,
    "concurrency": 2
  },
  "context": {
    "users": {
      "tenants": 3,
      "users_per_tenant": 2
      }}}, {
    "args": {
      "flavor": {
        "name": "{{flavor_name}}"
        },
```

[1] *Continuous Integration* en *Continuous Deployment*
[2] Een template engine geschreven in pure Python

```
      "image": {
        "name": "^cirros.*-disk$"
        },
      "auto_assign_nic": true
    },
    "runner": {
      "type": "constant",
      "times": 10,
      "concurrency": 2
    },
    "context": {
      "users": {
        "tenants": 3,
        "users_per_tenant": 2
      },
      "network": {
        "start_cidr": "10.2.0.0/24",
        "networks_per_tenant": 2
      }}}]}
```

Listing 10: Rally test

Dit invoerbestand stelt een scenario, NovaServers.boot_and_delete_server, voor met twee configuraties. De eerste configuratie maakt gebruik van de flavor m1.tiny tenzij er een andere wordt meegegeven op de commandolijn, de standaard meegeleverde cirros image en de force_delete option ingesteld op false. De runner-instelling bepaalt dat het proces 10 keer herhaald wordt waarbij er twee iteraties tegelijkertijd kunnen gebeuren (concurrency) en de context stelt de 6 gebruikers voor, verdeeld over 3 tenants. De tweede configuratie is grotendeels dezelfde als de eerste en bovendien wordt er een extra network-optie meegegeven.

Met bovenstaand invoerbestand kan de test worden uitgevoerd, eerst op een all-in-one omgeving met 1 node, vervolgens op de testomgeving zoals beschreven in Hoofdstuk **??**. De test uitvoeren gebeurt via volgend commando:

```
$ rally task start `/opt/stack/rally/samples/tasks/scenarios/nova/
boot-and-delete.json`
```

Na het uitvoeren van de test zijn de resultaten beschikbaar voor exporteren. Via volgend commando worden de resultaten geëxporteerd in een HTML-bestand:

```
$ rally task report --out=report1.html
```

Na het uitvoeren van de testen op beide omgevingen zijn er twee rapporten beschikbaar, één voor elke omgeving. De belangrijkste gegevens worden weergegeven in Figuur **??** en Figuur **??**

en hierna besproken. De bovenste grafieken stellen de resultaten van de eerste configuratie voor, de onderste grafieken stellen de resultaten van de twee configuratie voor.

Zoals te zien bovenaan in Figuur **??** en Figuur **??** is er een duidelijk verschil in de *Load Profile* grafieken tussen beide opstellingen. Deze stellen het aantal parallelle iteraties voor in functie van de tijd en hieruit is duidelijk dat de testomgeving parallellisme beter ondersteund. Ook de tijd om een instantie te starten en te verwijderen is 12% tot 20% beter in de testomgeving dan in de all-in-one omgeving. In de onderste grafieken is het meteen duidelijk dat de all-in-one opstelling faalt bij de laatste twee iteraties in tegenstelling tot de testomgeving. De tijd om een instantie te starten en te verwijderen is 16% tot 27% beter in de testomgeving dan in de all-in-one omgeving en het Load Profile is constanter in de testomgeving wat zorgt voor een betere betrouwbaarheid.

De testen uitgevoerd door Rally zijn nuttig om een nieuwe Nova-scheduler te evalueren op high-level niveau. Het ontwikkelen van een eigen test op een aangepaste Nova-scheduler biedt de mogelijkheid om deze nieuwe scheduler te testen met een vari"erende werklast waarbij de resultaten nadien eenvoudig worden omgezet in grafieken. Een belangrijk aspect is dat hiermee bekeken wordt of de nieuwe scheduler niet faalt om nieuwe instanties te creëren bij een stijgende werklast.

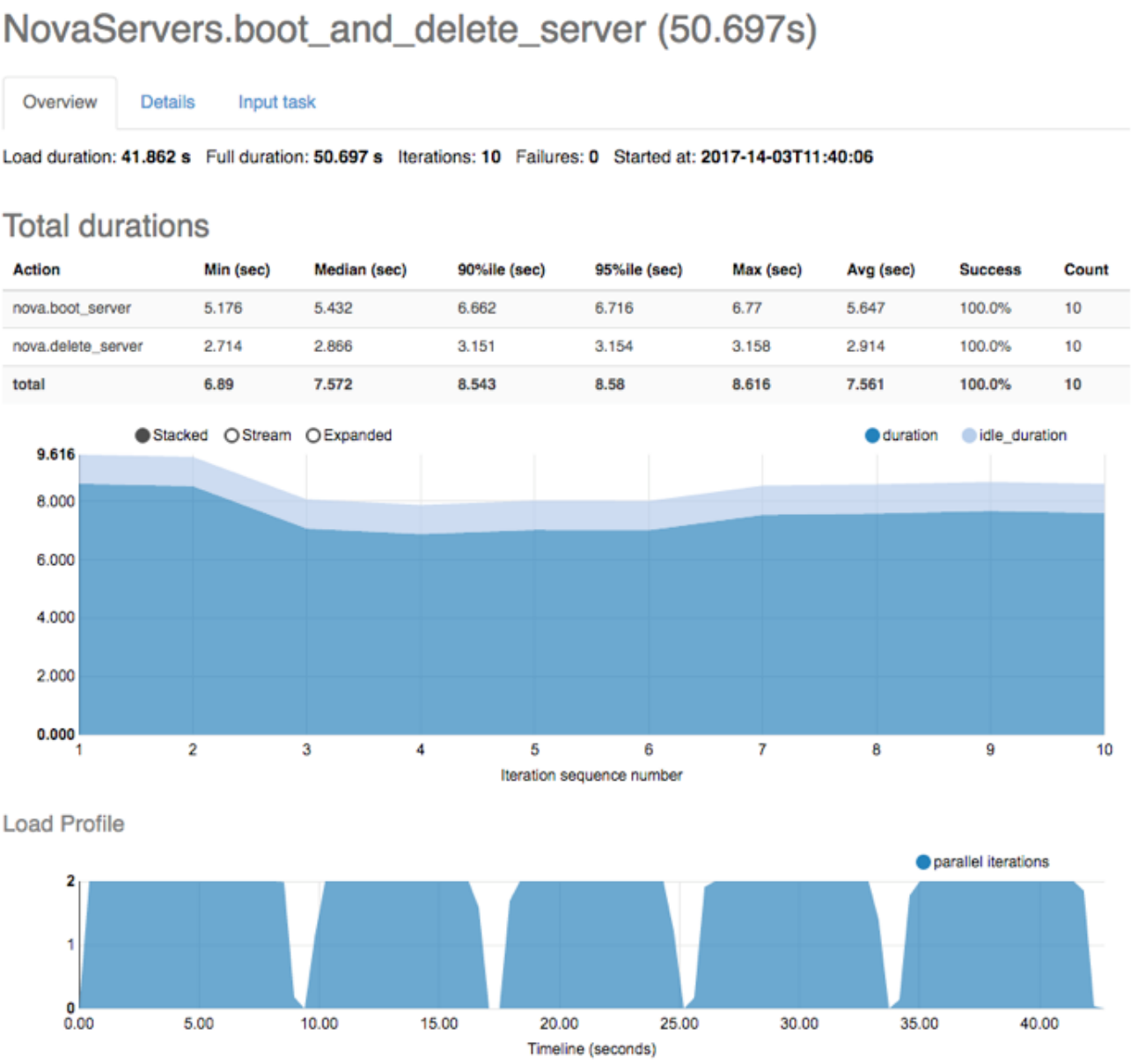
5.2 Mogelijkheden om te monitoren

Deze sectie geeft een overzicht weer van de verschillende mogelijkheden om een OpenStack-cloud te monitoren en data te verzamelen. Elke mogelijkheid zal tevens ook kort worden toegelicht met een eventueel voorbeeld.

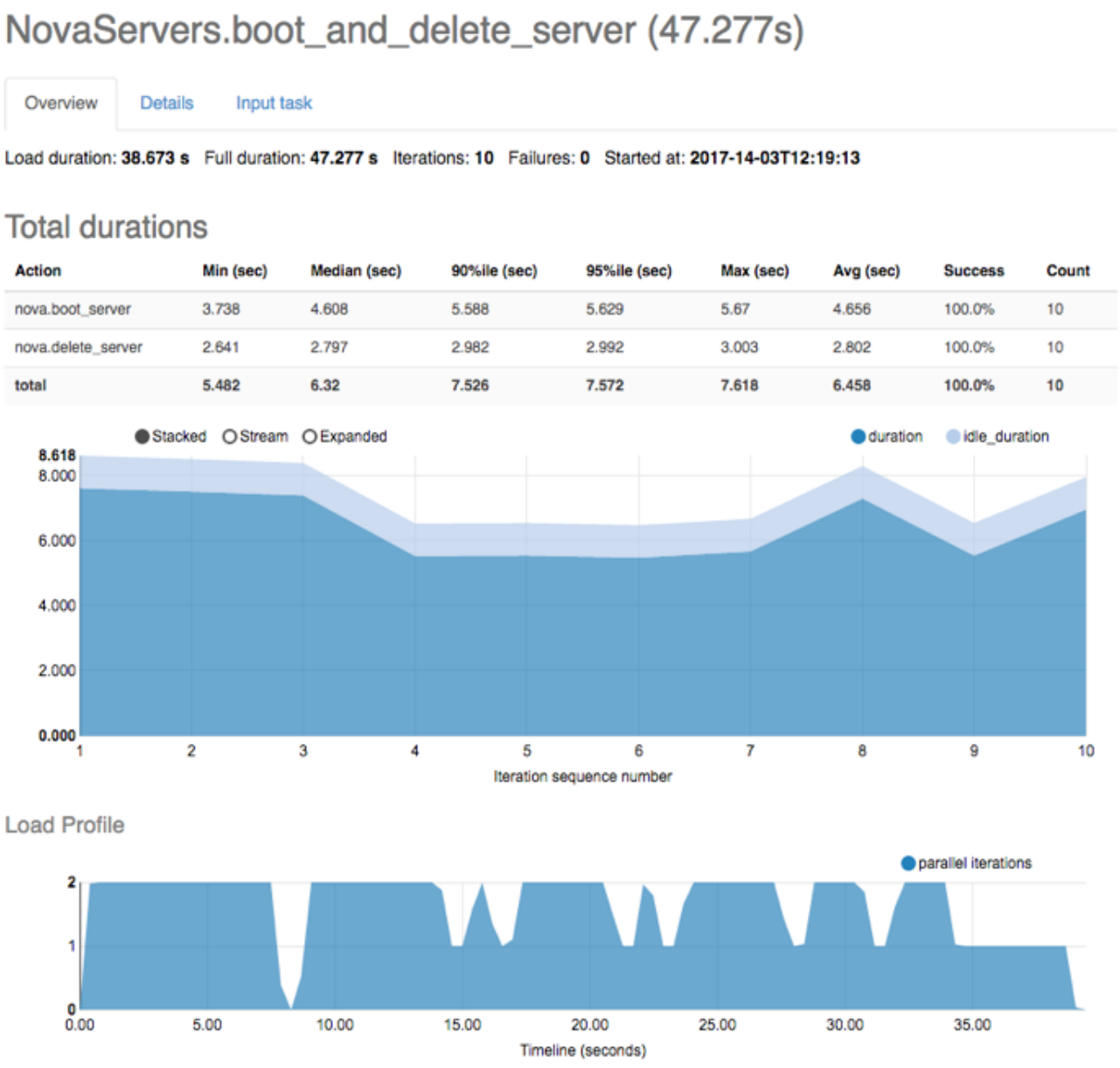
5.2.1 Nodejs-agent

Het monitoren van de drie hypervisors zal gebeuren met behulp van een zelf ontwikkelde Node.js-agent. Deze applicatie is een Node.js-applicatie die zal draaien op elk van de hypervisors. Met behulp van *Express*, een *npm package*, wordt er een node-server geïnitialseerd en gestart die vervolgens zal luisteren naar aanvragen op een vooraf ingestelde poort. Op dit ogenblik zijn volgende aanvragen geïmplementeerd:

- */ping*: verwacht geen query, en geeft een 1 terug als resultaat (*text/plain*)
- */mongodb*: verwacht geen query, en geeft een JSON-string terug met het huidig resource-verbruik (*application/json*)



Figuur 5.1: Resultaat Rally-test: All-in-One Node



Figuur 5.2: Resultaat Rally-test: Testomgeving

- */periodstats*: verwacht een query met 2 parameters, start en stop, waarbij beiden een tijdstip omschrijven met als vorm yyyy-dd-mmThh:mm:ss.xxxZ en geeft een JSON-string terug met het resource-verbruik tussen de twee meegegeven tijdstippen (*application/json*)

- */ceilometerstats*: verwacht een query met 2 parameters, start en stop, waarbij beiden een tijdstip omschrijven met als vorm yyyy-dd-mmThh:mm:ss.xxxZ en geeft een JSON-string terug met de metingen van Ceilometer tussen de twee meegegeven tijdstippen (*application/json*)

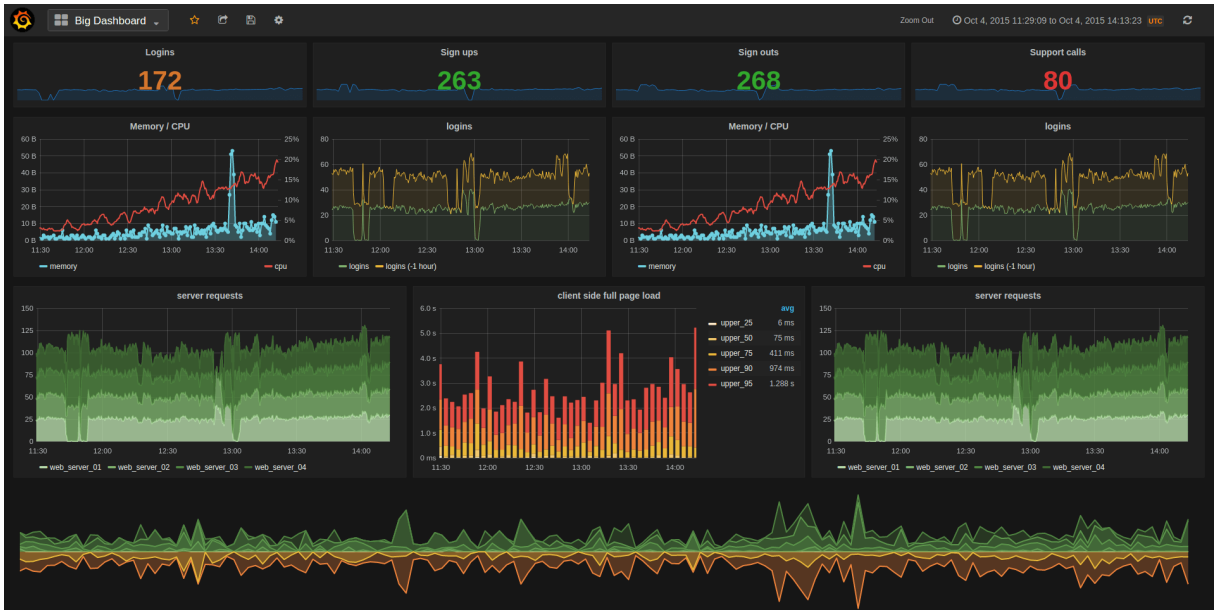
- */timestats*: verwacht een query met 2 parameters, start en stop, waarbij beiden een tijdstip omschrijven met als vorm yyyy-dd-mmThh:mm:ss.xxxZ en geeft een JSON-string terug met het resource-verbruik tussen de twee meegegeven tijdstippen alsook de metingen van ceilometer, beiden geordend volgens tijdstip (*application/json*)

De ping-aanvraag geeft eenvoudig weer of de agent op de hypervisor bereikbaar en geactiveerd is door een 1 terug te sturen naar de aanvrager. De mongodb aanvraag is belangrijk voor de correcte werking van de andere aanvragen. Indien de agent zo een aanvraag krijgt zal hij met behulp van het node-package *os-utils* het CPU-verbruik en de hoeveelheid vrij geheugen bepalen. Vervolgens zal de agent met behulp van het node-package *mysql* een connectie maken met de MySQL-databank op de OpenStack-controller, namelijk jerico-03, om de hoeveelheid actieve instanties op te vragen die op de hypervisor gehost zijn. Al deze resultaten samen met het huidige tijdstip worden dan bewaard in een mongoDB-databank welke actief is op jerico-01. Deze resultaten worden ook teruggezonden naar de aanvrager in JSON-formaat en kunnen nadien ook opgevraagd worden via de andere aanvragen (met uitzondering van de ping aanvraag).

Indien een aanvraag wordt gestuurd naar periodstats samen met twee parameters, een start- en eindtijdstip, zal de agent alle data van alle hypervisors tussen deze twee tijdstippen opvragen van de mongoDB op jerico-01. Een aanvraag naar ceilometerstats is gelijkaardig, enkel zal hier een aanvraag worden gestuurd naar de MongoDB op jerico-03 om zo alle statistieken die Ceilometer gemeten heeft binnen de meegegeven periode terug te geven aan de aanvrager. De timestats-aanvraag is een combinatie van de laatste twee, mits enkele aanpassingen. Zo wordt de data van ceilometer opnieuw opgevraagd van de MongoDB op jerico-03 en worden ook de instanties van de MySQL-databank op jerico-03 opgevraagd, beiden met het bijhorende tijdstip. Een combinatie van deze resultaten wordt nadien bezorgd aan de aanvrager.

5.2.2 Grafana

Grafana [**?**] is een van de meest bekende open-source software voor tijdsreeksanalyse. Het biedt een krachtige en elegante manier aan voor het creëren, ontdekken en delen van dashboards en



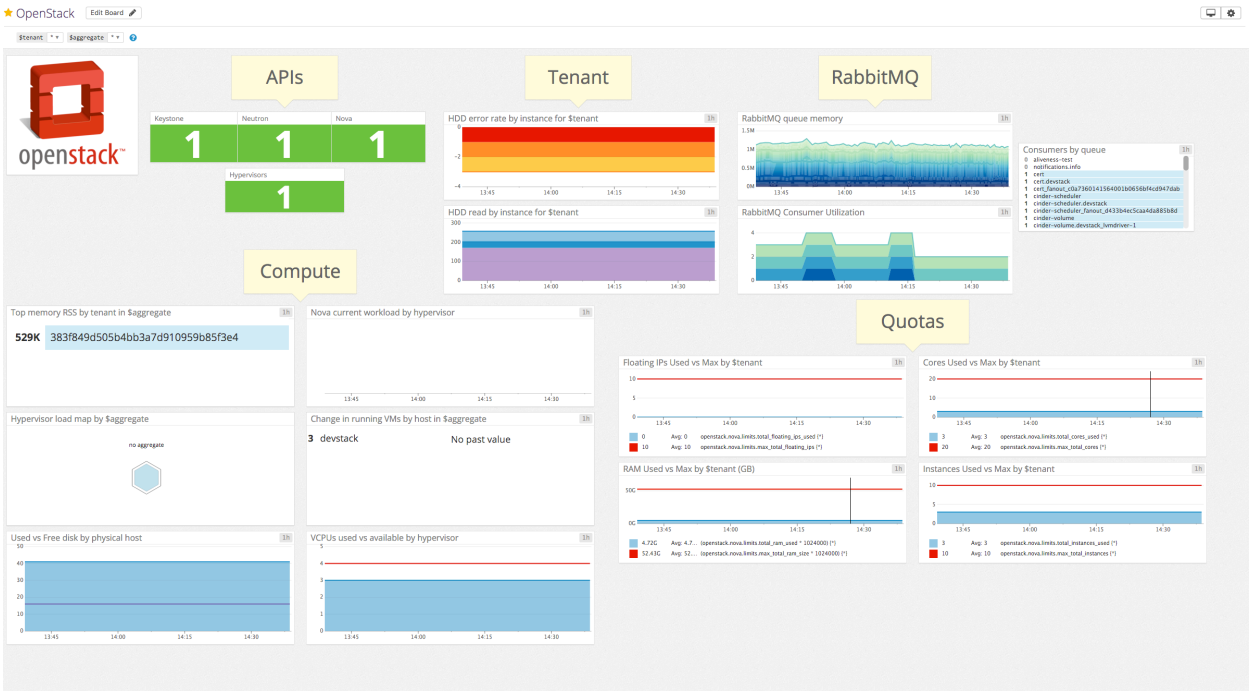
Figuur 5.3: Mock-up van een Grafana-dashboard

data met een team of de hele wereld. Dankzij verschillende plugins is het mogelijk om variërende panelen, die onder andere grafieken bevatten, te configureren en te analyseren.

Ook een OpenStack-cloud kan met Grafana gemonitord en geanalyseerd worden indien aan bepaalde voorwaarden voldaan is. Zo moet OpenStack gebruikmaken van de Gnocchi-database waar Ceilometer zijn statistieken zal bewaren. Gnocchi [**?**] is een projectnaam voor een TDBaaS (Time Series Database as a Service) project dat moet samenwerken met OpenStack Ceilometer. Het voordeel van deze benadering is dat vele metrieken, van de temperatuur tot het CPU-gebruik, worden bewaard per tijdstip en zo eenvoudig kunnen worden opgevraagd. Grafana kan dan met behulp van de Gnocchi-plugin[3] en de nodige configuratie een volledige OpenStack-cloud monitoren met een mogelijk resultaat zoals in Figuur **??**.

Het grootste voordeel van Grafana is dat het eenvoudig uitbreidbaar is naar andere cloud-structuren zonder al te veel wijzigingen. Daarnaast is het snel, performant en is er absolute vrijheid waar de Grafana-host zal draaien. De nadelen zijn echter de complexiteit om alles te configureren en het verplicht gebruik van Gnocchi in een OpenStack-cloud. Aangezien Gnocchi niet wordt gebruikt in de testomgeving (waar gebruik wordt gemaakt van MongoDB), biedt Grafana geen mogelijke manier om de evaluatie uit te voeren.

[3] `https://grafana.com/plugins/sileht-gnocchi-datasource`



Figuur 5.4: Mock-up van een Datadog-dashboard

5.2.3 Datadog

Datadog [**?**] is een softwarepakket in combinatie met een webapplicatie voor het monitoren van clouds. Dankzij meer dan 150 integraties van verschillende cloud-systemen zoals Amazon EC2, Kubernetes en ook OpenStack is het uitermate geschikt voor de monitoring van een cloud. Om Datadog te integreren met OpenStack moet de Datadog Agent geïnstalleerd worden op de hosts die dienst doen als hypervisor. Vervolgens kunnen er via een beveiligde omgeving op de website van Datadog online dashboards worden gecreëerd zodat er verschillende metrieken gemonitord worden. Een voorbeeld van zo'n dashboard is te zien in Figuur **??**.

Net zoals bij Grafana is het grote voordeel bij Datadog de overdraagbaarheid naar andere cloud-systemen. Daarnaast is het ook veel eenvoudiger in gebruik dankzij een eenvoudige installatie en configuratie. Datadog is beschikbaar als een gratis versie met een beperking van maximum 5 hosts waarbij enkel de standaard metrieken slechts 1 dag worden bewaard. Het grote nadeel is dat de prijs voor een beter pakket dat complexere metrieken en een langere bewaartijd biedt. Bijgevolg werd Datadog niet gekozen als optie om resource-allocatieschema's te evalueren.

5.2.4 Overige mogelijkheden

Verschillende overige mogelijkheden staan opgelijst op de monitoringpagina van OpenStack.[4] In deze lijst staan naast Rally en Datadog ook nog enkele interessante opties zoals Monasca [**?**], een lopend OpenStack-project voor een open-source, schaalbaar, performant en fouttollerant *monitoring-as-a-service* oplossing binnenin OpenStack. Andere mogelijkheden zijn bijvoorbeeld Graphite, Ganglia, Nagios, Stacktach, etc.

5.3 Monitoring met Node.js en C3.js

De uiteindelijke techniek om te monitoren maakt gebruik van de nodejs-agent in combinatie met een zelf ontwikkelde webapplicatie met c3.js [**?**] voor het weergeven van de grafieken. In deze sectie wordt dieper ingegaan op de nodejs-agent en de webapplicatie.

5.3.1 Implementatie van de nodejs-agent

Zoals beschreven in Sectie **??** is de nodejs-agent een Node.js-applicatie die gebruikmaakt van verschillende node-packages. Deze applicatie is zo geschreven dat er eenvoudig nieuwe hosts kunnen worden toegevoegd of verwijderd. De code is beschikbaar via GitHub[5] en hier worden enkele stukken code kort toegelicht voorafgegaan door de vereisten.

Om de applicatie te kunnen starten zijn er enkele vereisten. Zo moet elke host waar de applicatie zal draaien voorzien zijn van Node.js en npm. Vervolgens moet er een MongoDB zijn die toegankelijk is voor alle hosts (eventueel met gebruikersnaam en wachtwoord). Deze databank bezit 1 tabel met de naam *monitoring* en n collecties waarbij n staat voor het aantal hosts die gemonitord zullen worden. De databank waar Ceilometer de gegevens bewaard (in de testomgeving is dat de MongoDB op jerico-03) moet ook toegankelijk zijn voor externe verbindingen. Let wel op dat er geen authenticatie wordt ingesteld op deze databank want dit kan de werking van Ceilometer ernstig verstoren.

Een bestand dat steeds aanwezig is in een Node.js-applicatie is het package.json-bestand. Hier wordt meer informatie over de applicatie gegeven, welke commando's er uitgevoerd kunnen worden maar vooral welke afhankelijkheden (*dependencies*) er nodig zijn om de geschreven code te kunnen uitvoeren. De belangrijkste afhankelijkheden hier zijn onder andere *Express*, *MongoDB*, *Q*, etc.

[4] `https://wiki.openstack.org/wiki/Operations/Monitoring`
[5] `https://github.ugent.be/jfmoeyer/EvaluationRASOpenStack/tree/master/nodejs-agent`

Het bestand config.json bevat de nodige configuratie voor het uitvoeren van de applicatie. Hierin staan onder andere de verschillende hosts die gemonitord moeten worden, in welke tabellen de data wordt bewaard, op welke host de code wordt uitgevoerd, welke poort Express zal gebruiken, etc. In het geval van de testomgeving moest dankzij dit configuratiebestand enkel de variabele *current_host* worden aangepast per verschillende host zodat de applicatie weet op welke host hij actief is.

Systeminfo.js bevat de drie verschillende methodes die metrieken zal meten, bewaren en teruggeven. De eerste methode *updateMongoDB(callback)* zal de drie metrieken, CPU-gebruik, vrij geheugen en aantal actieve instanties van de huidige host opvragen, bewaren in de MongoDB en deze ook teruggeven. De code van deze functie ziet er als volgt uit:

```
method.updateMongoDB = (callback) => {
  os.cpuUsage((cpu_usage) => {
    var free_mem = os.freemem();
    connection.query("SELECT host, count(*) from instances where power_state =
    ↪  1 and host like '" + HOST_NAME + "' group by host", (error, result,
    ↪  fields) => {
      if (result[0]) {
        var instances = result[0]['count(*)'];
      } else {
        var instances = 0;
      }
      MongoClient.connect(url_monitoring, (err, db) => {
        assert.equal(null, err);
        insertDocument(db, cpu_usage, free_mem, instances, () => {
          db.close();
          callback({
            "cpu_usage": cpu_usage,
            "free_mem": free_mem,
            "instances": instances,
            "timestamp": new Date()
  });});});});});});
}
```

Listing 11: nodejs-agent: updateMongoDB-functie

Als eerste wordt het CPU-gebruik berekend met behulp van de node-package *os-utils*. Vervolgens wordt met hetzelfde pakket de hoeveelheid vrij geheugen opgevraagd. Daarna maakt de applicatie connectie met de MySQL-host om de hoeveelheid actieve instanties op de host te bepalen. Dit alles wordt tenslotte in de MongoDB bewaard en teruggegeven als JSON-object.

De tweede methode *getPeriodStatsScale(start, end, callback)* zal de statistieken tussen een bepaalde periode (tussen start en stop) downloaden vanaf de MongoDB. De gebruikte code ziet er als volgt uit:

```
method.getPeriodStatsScale = (start, end, callback) => {
  var promises = [];
  MongoClient.connect(url_monitoring, (err, db) => {
    assert.equal(null, err);
    var complete_result = {};
    for (var node of appConfig.nodes) {
      complete_result[node.server_name + " CPU"] = [];
      complete_result[node.server_name + " Memory"] = [];
      complete_result[node.server_name + " Instances"] = [];
      var cursor = db.collection(node.mongo_table_name).find({ "timestamp": {
      ↪  $gt: new Date(start), $lt: new Date(end) } });
      var promise = new Promise((resolve, reject) => {
        var node_name = node.server_name;
        cursor.each((err, doc) => {
          if (err) {
            reject("Something went wrong with node " + node_name);
          }
          if (doc != null) {
            complete_result[node_name + " CPU"].push(doc.cpu_usage);
            complete_result[node_name + " Memory"].push(doc.free_mem);
            complete_result[node_name + " Instances"].push(doc.instances);
          } else {
            resolve(node_name + " is inserted!");
      }});});
      promises.push(promise);
    }
    Q.all(promises).then(() => {
      db.close();
      callback(complete_result);
    }).fail(console.error);
  });
}
```

Listing 12: nodejs-agent: getPeriodStatsScale-functie

De belangrijkste functionaliteit van deze functie is het gebruik van Promises binnen JavaScript.

Dit laat namelijk toe dat indien er een connectie is met de MongoDB, elke node uit het conf.json-bestand overlopen wordt en dit bepaalde methodes zal uitvoeren (zoals alle metrieken opvragen). Vervolgens zal dankzij het node-package *Q* gewacht worden totdat alle Promises voltooid zijn waarna het verkregen resultaat kan worden teruggegeven.

De laatste methode om metrieken op te vragen is *getCeilometerAndInstancesTimeScale(start, stop, callback)*. Deze dient om de werklast, gemeten door Ceilometer, op te vragen samen met het aantal instanties per host. De gebruikte code ziet er uit als volgt:

```
method.getCeilometerAndInstancesTimeScale = (start, stop, callback) => {
  MongoClient.connect(url_ceilometer, (err, db) => {
    assert.equal(null, err);
    var cursor = db.collection("meter").find({ timestamp: { $gt: new
    ↪  Date(start), $lt: new Date(stop) }, counter_name:
    ↪  appConfig.scale_app.counter_name, "resource_metadata.display_name":
    ↪  appConfig.scale_app.scale_group_name, "resource_metadata.status":
    ↪  "active" });
    var result = {};
    var complete_result = {};
    var promises = [];
    cursor.each((err, doc) => {
      if (doc != null) {
        result[doc.timestamp] = doc.counter_volume;
      } else {
        db.close();
          MongoClient.connect(url_monitoring, (err, db) => {
            assert.equal(null, err);
            for (var node of appConfig.nodes) {
              complete_result[node.server_name] = {};
              var cursor = db.collection(node.mongo_table_name).find({
              ↪  "timestamp": { $gt: new Date(start), $lt: new Date(stop) }
              ↪  });
              var promise = new Promise((resolve, reject) => {
                var node_name = node.server_name;
                cursor.each((err, doc) => {
                  if (err) {
                    reject("Something went wrong with node " + node_name)
                  }
                  if (doc != null) {
                    complete_result[node_name][doc.timestamp] = doc.instances;
```

```
                } else {
                  resolve(node_name + " is inserted!");
              }});});
            promises.push(promise);
            }
          Q.all(promises).then(() => {
            db.close();
            callback([result, complete_result]);
          }).fail(console.error);
  });}});});
}
```

Listing 13: nodejs-agent: getCeilometerAndInstancesTimeScale-functie

Hier wordt eveneens gebruikgemaakt van Promises om alle hosts in het configuratiebestand te overlopen. Eerst worden de gegevens van de ceilometer-databank opgevraagd en nadien de metrieken van elke host. Op het einde van de code wordt er gewacht tot alle Promises zijn voltooid alvorens het resultaat wordt teruggegeven aan de callback-functie.

Het laatste bestand in deze applicatie heet *rpicluster.agent.js*. Dit is het kloppend hart van de applicatie doordat deze de aanvragen vanuit de buitenwereld zal verwerken en uitvoeren. Met behulp van het Express-package van Node wordt er een router geïnitialiseerd, geconfigureerd en per aanvraag, zoals vermeld in Sectie **??**, een functie geïmplementeerd. Een voorbeeld van zulke functie ziet er uit als volgt:

```
router.get("/timestats", function(req, res) {
  res.writeHead(200, { 'Content-Type': 'application/json'});
  var url_parts = url.parse(req.url, true);
  var query = url_parts.query;
  new Systeminfo().getCeilometerAndInstancesTimeScale(query.start, query.stop,
  ↪  (result) => {
    res.end(JSON.stringify(result));
  });
});
```

Listing 14: nodejs-agent: nodejs-agent: implementatie van een aanvraag

Vooraleer deze code tot stand is gekomen, werd er eerst gebruik gemaakt van asynchrone en hard-coded aanvragen die moeilijk uit te breiden waren naar meer of minder hosts. Dankzij de overschakeling naar het gebruik van Promises en het conf.json-bestand, zijn er twee voordelen.

Er kunnen eenvoudig meerdere hosts worden toegevoegd of verwijderd (de databank die alles bewaard kan zelfs extern zijn) en dankzij het gebruik van de Promises is de code performanter omdat deze stukken code "gelijktijdig" uitgevoerd worden.

Om de nodejs-agent te starten zijn er slechts twee commando's nodig, één om de nodige afhankelijkheden te installeren en één om de applicatie te starten. De twee commando's, waarbij de & in het laatste commando ervoor zorgt dat de applicatie in de achtergrond wordt uitgevoerd, zijn de volgende:

Listing 15: Starten van de nodjs-agent

```
$ sudo npm install
$ sudo npm start &
```

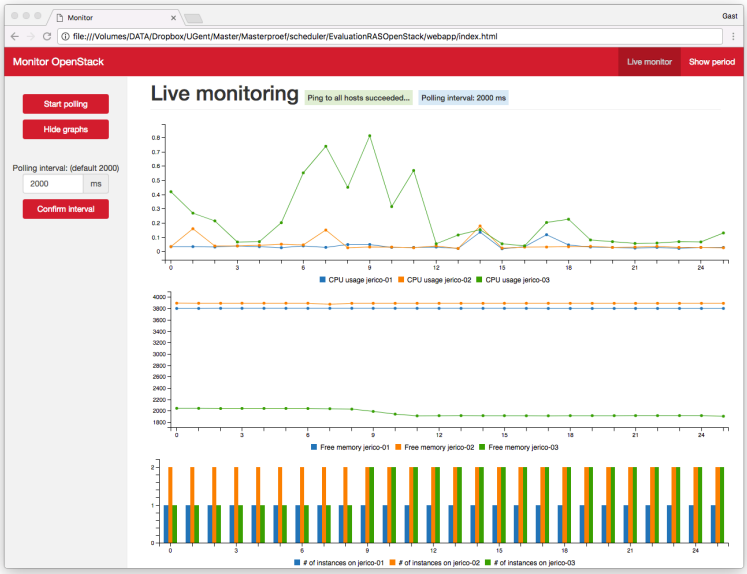
5.3.2 Implementatie van de webapplicatie

De webapplicatie bestaat uit twee pagina's, één voor live-monitoring en één voor het opvragen van gegevens die gemeten zijn tijdens een bepaalde periode (op voorwaarde dat er tijdens die periode live-monitoring is gebeurd). Ze werkt nauw samen met de nodejs-agent voor het opvragen van gegevens en het aansturen van de databank. Ook in de webapplicatie werd gebruikgemaakt van een configuratiebestand zodat er eenvoudig meerdere hosts kunnen worden toegevoegd of verwijderd. De code is beschikbaar via GitHub[6] en maakt gebruik van c3.js [**?**] voor het visualiseren van de grafieken. Ook hier worden Promises gebruikt voor zowel betere performantie als schaalbaarheid. Aangezien de code zeer gelijkaardig is aan die van de nodejs-agent wordt deze hier ook niet verder besproken.

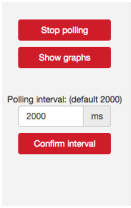
De live-monitoring pagina wordt weergegeven in Figuur **??** en Figuur **??** en de functionaliteiten worden hier kort besproken. De drie grafieken in deelfiguur (a) geven respectievelijk het CPU-gebruik, de hoeveelheid vrij geheugen en het aantal actieve instanties weer per host. Indien de gebruiker de grafieken wenst te verbergen tijdens een sessie, kan hij deze uitschakelen en zal een spinner aangeven of het monitoren bezig is of niet wat te zien is in deelfiguur (b). Bij een sessie zal er standaard elke 2 seconden (2000 ms) een aanvraag worden gestuurd naar de nodejs-agent voor het bewaren en teruggeven van de metrieken op dat moment (de /mongodb aanvraag van de nodejs-agent). Het interval kan aangepast worden naar belangen van de gebruiker. Vooraleer het monitoren kan gestart worden, is er een controle of de te monitoren hosts wel beschikbaar zijn. Dit gebeurt via een ping-aanvraag naar alle hosts en pas wanneer alle hosts antwoord geven, kan er gemonitord worden.

De periode-pagina wordt weergegeven in Figuur **??** en is zeer gelijkaardig aan de live-monitor pagina. Hier wordt wel een extra grafiek weergegeven, die het aantal instanties per host combineert

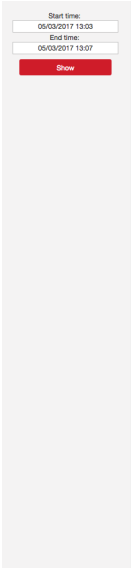
[6] `https://github.ugent.be/jfmoeyer/EvaluationRASOpenStack/tree/master/webapp`



(a) Live monitoring - grafieken zichtbaar



(b) Live monitoring - grafieken verborgen



(c) Overzicht van een periode

Figuur 5.5: Webapplicatie

met de gemeten statistieken van Ceilometer. Onmiddellijk na het laden van de pagina wordt er gecontroleerd of de host die de databank bevat bereikbaar is. Enkel indien deze bereikbaar is kan er een periode worden ingevoerd en zal deze gevisualiseerd worden in de 4 grafieken.

6

Evaluatie van resource allocatieschema's

In dit hoofdstuk wordt de eigenlijke Proof-of-Concept besproken en geëvalueerd. Met behulp van de testomgeving, beschreven in Hoofdstuk **??** en de monitor tools, uitgelegd in Hoofdstuk **??** is de hele evaluatie uitgevoerd.

De opbouw van dit hoofdstuk is als volgt. Eerst wordt er uitgelegd hoe de FaaFo-applicatie werklast kan genereren op één of meerdere worker-instanties. Vervolgens wordt een korte hypothese beschreven met het verwachte resultaat van de werking van Round Robin. Nadien wordt een test uitgevoerd met Rally om te bekijken of het schema naar behoren werkt op high-level niveau om af te sluiten met een stress-test, die wordt gemonitord en besproken.

6.1 De schaalbare FaaFo-applicatie

Zoals besproken in Sectie **??** is de FaaFo-applicatie uitermate geschikt om een stress-test uit te voeren op de drie gebruikte hypervisors met de RoundRobinScheduler ingeplugd in Nova. Na het activeren van de OpenStack stack kan er via de FaaFo-controller een commando worden ingegeven die voor voldoende werklast zal zorgen. De OpenStack-Orchestration en -Telemetry services laten dan alarmen afgaan bij een hoge werklast waardoor er extra worker-instanties aangemaakt worden. Zoals beschreven in de template in Sectie **??** kunnen er maximum zeven

worker-instanties tegelijkertijd actief zijn.

Op de OpenStack controller-node, jerico-03, kan dankzij de security group instellingen en de gebruikte sleutel (de key_name parameter) een SSH-verbinding gelegd worden met de FaaFo-controller. Van hieruit kunnen er dan verschillende taken gestart worden als ook het weergeven van de huidige taken. De FaaFo-applicatie is zo gemaakt dat deze met volgende simpele commando's kan werken:

```
$ faafo create --width 5555 --height 5555 --tasks 3
$ faafo list
```

Listing 16: Starten van drie FaaFo-taken

Het eerste commando zal 3 taken creëren die elk een fractaal moeten berekenen van de meegegeven hoogte en breedte. Deze drie taken komen dan terecht in de queue op FaaFo-controller waarna de FaaFo-workers een taak van de queue halen en deze uitvoeren. Het berekenen van een fractaal vergt veel CPU-verbruik en hierdoor zal het *cpu_alarm_high*-alarm getriggerd worden. De OpenStack Orchestration-services, Heat, zullen hierop reageren door extra FaaFo-workers aan te maken.

Het tweede commando geeft een overzicht weer van de berekende en nog te berekenen fractalen, telkens met hun afmetingen en hun grootte op de harde schijf. Is de grootte nog nul, dan betekent dit dat het fractaal nog berekend moet worden, of dat er een FaaFo-worker mee bezig is

6.2 Hypothese

Round Robin is een contextloos algoritme, het weet niets af van de hosts hun capaciteiten. Bijgevolg wordt er verwacht dat de Rally-test een gunstig resultaat zal afleveren omdat deze test ook zonder context wordt uitgevoerd. Het gewoon starten en stoppen van instanties zonder generatie van werklast is ideaal voor een Round Robin schema en bijgevolg zullen alle instanties gelijkmatig verdeeld worden over de verschillende hosts.

In het geval van de schaalbare FaaFo-applicatie kunnen de nadelen van een Round Robin schema wel eens zichtbaar worden. De Orchestration-services van OpenStack verwijderen immers een willekeurige host bij een laag-cpu-alarm. Indien er nadien terug instanties bijkomen door een hoog-cpu-alarm, kan het zijn dat een host meer instanties bevat dan andere waardoor de werklast niet gelijk verdeeld zal worden. In welke mate dit nadelig is, zal de test zelf moeten aantonen.

6.3 Evaluatie met Rally

De evaluatie met Rally gebeurt met dezelfde template zoals beschreven in Sectie **??**. Het enige verschil met de test die daar is uitgevoerd, is de gebruikte Nova-scheduler. De resultaten van de test zijn te zien in Figuur **??**. De bovenste grafiek is de uitvoer van Rally zelf, gelijkaardig aan het resultaat uit Sectie **??**, de onderste grafiek is de uitvoer van de zelf ontwikkelde monitorapplicatie beschreven in Sectie **??**.

Zoals verwacht in de hypothese wordt de Rally-test beëindigd zonder fouten. Dit komt omdat de instanties steeds gelijkmatig worden verdeeld over de drie hosts. Dit is ook te zien in de Figuur **??** waarbij geen enkele host meer dan 2 instanties tegelijkertijd bezit. Verrassend aan het resultaat is dat de RoundRobin-scheduler er duidelijk langer over doet om alles te starten en te stoppen vergeleken met Figuur **??** en Figuur **??**. Doordat tegelijkertijd de nodejs-agent op de hosts data aan het verzamelen was, kan dit een mogelijke verklaring zijn voor de iets langere duurtijd.

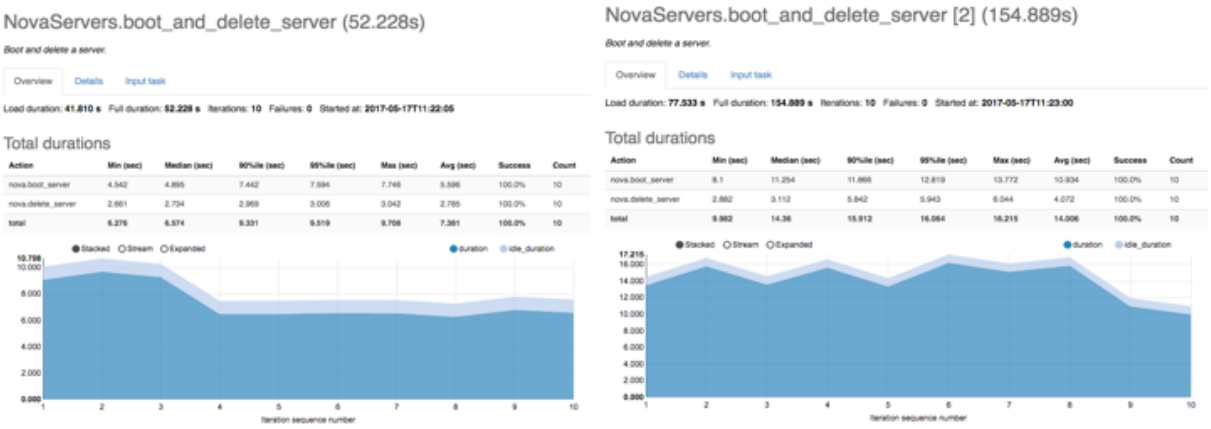
Uit deze test kunnen volgende besluiten genomen worden. Omdat deze test contextloos is (er werd nergens werklast gegenereerd), werkt een RoundRobin-scheduler naar behoren. De werklast op de host jerico-03 is wel een pak groter dan die op de andere hosts. Een mogelijke verklaring hiervoor is het feit dat jerico-03 zowel de OpenStack-controller is en dus de andere hosts moet aansturen alsook zelf instanties moet hosten. Daarom is het aan te raden om een OpenStack-controller geen dienst te laten doen als hypervisor, zodat de werklast beter verdeeld kan worden.

6.4 Evaluatie met de FaaFo-applicatie

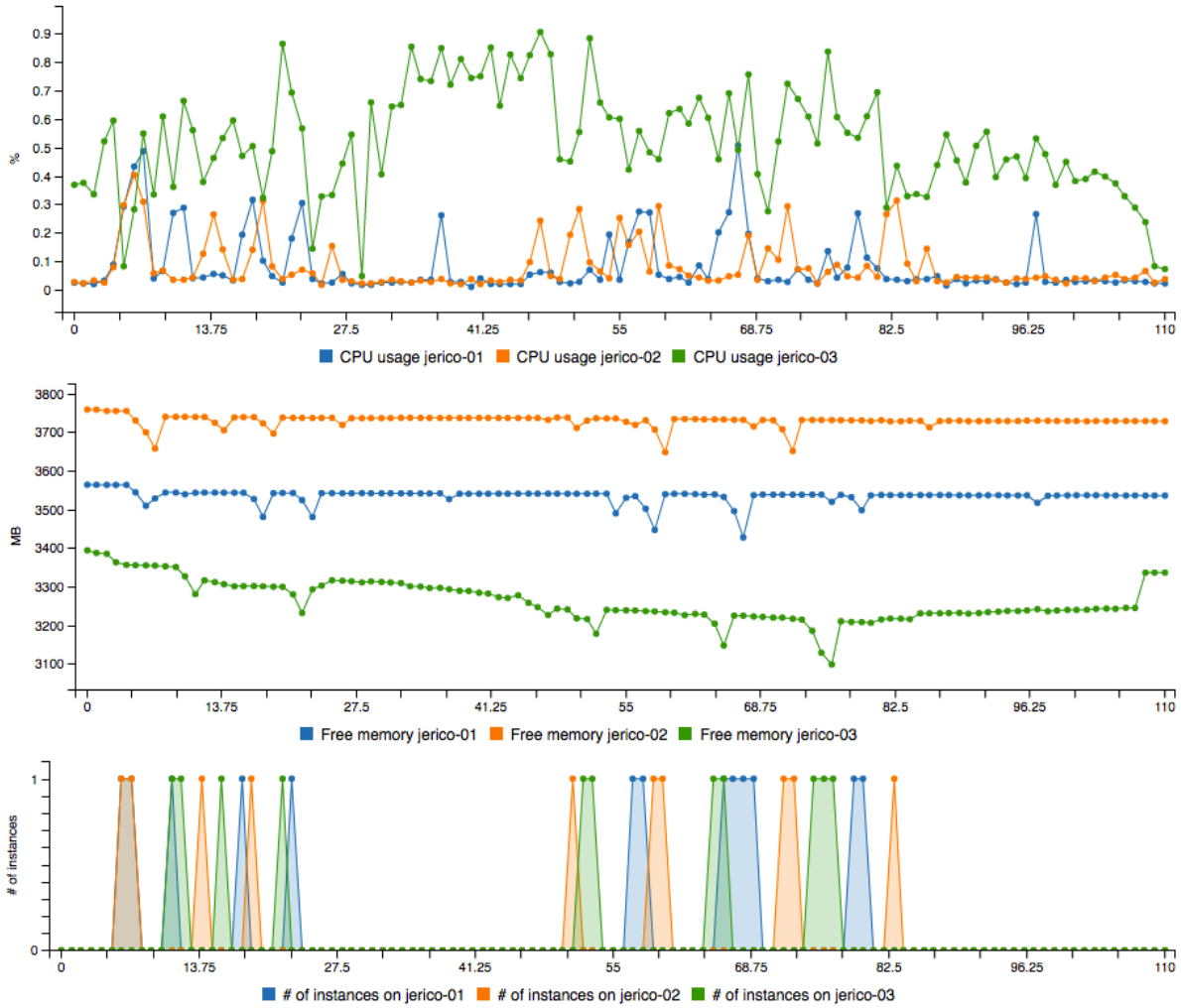
Het Round Robin schema wordt ook getest met de schaalbare FaaFo-applicatie. De gemeten resultaten met de nodejs-agent bedekken de hele procedure, vanaf het creëren van de stack tot en met het voltooien van het *faafo create*-commando en deze worden weergegeven in Figuur **??**.

Ook hier komen de resultaten goed overeen met de verwachte hypothese. De instanties worden in het begin evenredig verdeeld over de verschillende hosts. Ongeveer halverwege de resultaten zijn er enkele metingen met een laag cpu-gebruik, waardoor enkele worker-instanties verwijderd worden. Nadien worden terug hoge waarden gemeten van het cpu-gebruik waardoor nieuwe worker-instanties worden gestart. Deze nieuwe worker-instanties worden geplaatst op de eerstvolgende host in de rij van het Round Robin schema, zonder dat deze rekening houdt met de verwijderde instanties. Hierdoor zijn de instanties naar het einde van de test toe niet meer evenredig verdeeld en dat is ook zichtbaar in de derde en vierde grafiek.

Net zoals bij de Rally-test is de werklast op de OpenStack-controller groter dan bij de andere

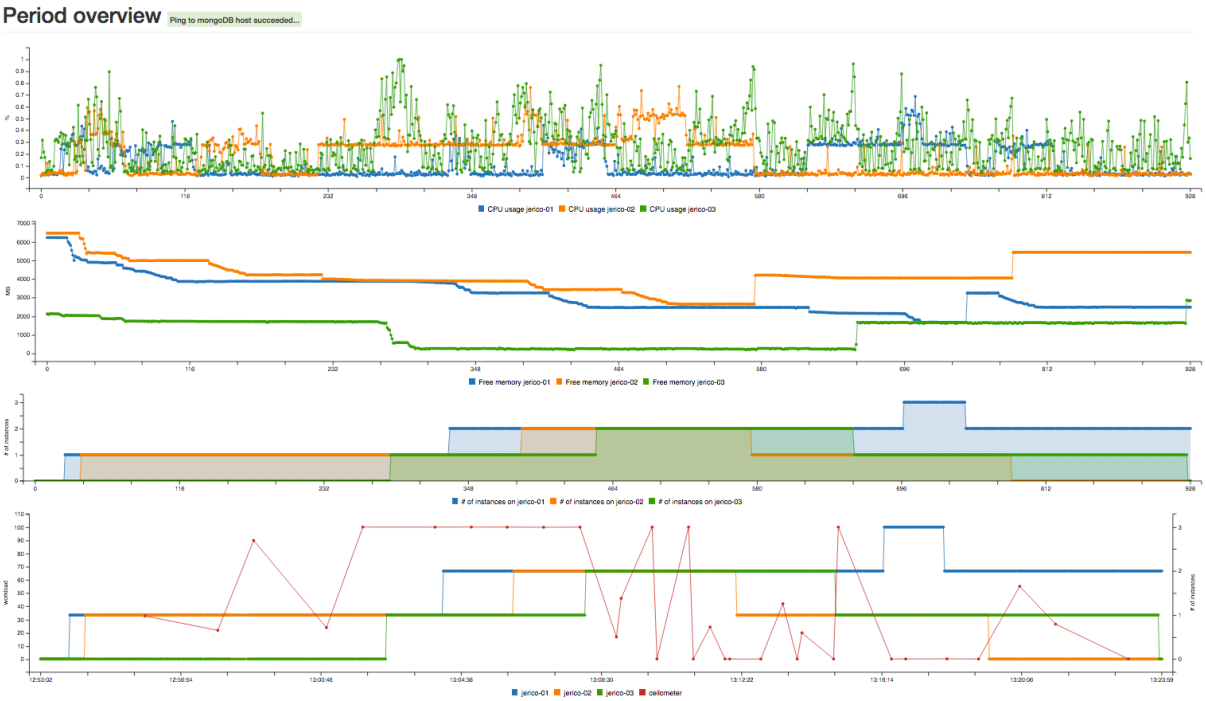


(a) Uitvoer van Rally



(b) Uitvoer van de monitorapplicatie

Figuur 6.1: Evaluatie met Rally - Resultaten



Figuur 6.2: Resultaat van de stress-test met de FaaFo-applicatie

hosts. Ook de hoeveelheid vrij geheugen is beperkt op de OpenStack-controller in vergelijking met de andere hosts. In een worst-case scenario kan het zijn dat de instanties verwijderd worden op alle hosts met uitzondering van jerico-03, waardoor de hoeveelheid vrij geheugen op deze host te laag is om nog een nieuwe instantie te hosten. Indien de volgende nieuwe instantie dan net op jerico-03 gehost wordt, zorgt dit voor ernstige gevolgen zoals bijvoorbeeld het uitvallen van jerico-03 en dus de gehele OpenStack-omgeving.

6.5 Resultaat van de evaluaties

Een Round Robin schema is simpel en werkt naar behoren in de meeste gevallen. Het grootste nadeel is de contextloze eigenschap van dit schema waardoor er mogelijks problemen optreden bij een tekort aan fysieke resources. De gestelde hypothese voorafgaand aan de verschillende evaluaties is zeer aansluitend aan de verkregen resultaten mits het toevoegen van enkele bijkomende conclusies.

Door het hele proces te overlopen, vanaf het opzetten van de testomgeving tot de verschillende evaluaties, werd de Proof-of-Concept aangetoond. Het grootste deel van de gebruikte configuratie en code kan herbruikt worden bij andere allocatieschema's. Enkel het nieuwe schema moet worden geschreven in Python met de nodige overerving van de Nova-filters. Na die aanpassing is

het mogelijk om het gehele evaluatieproces opnieuw te overlopen om zo de resultaten te bekomen, die nodig zijn voor de evaluatie van het nieuwe schema.

7

Conclusie

De bedoeling van deze thesis is het aanbieden van een eenvoudig en relatief snelle methode om nieuwe cloud-allocatieschema's te evalueren op een echt OpenStack-testbed. Zoals beschreven in het gerelateerd werk in Hoofdstuk **??**, is er de laatste tijd veel onderzoek gebeurd naar nieuwe cloud resource-allocatieschema's. Deze nieuwe schema's werden in de meeste gevallen getest en geëvalueerd op een simulator in plaats van op een fysiek cloud-testbed. Een belangrijke reden voor het gebruik van een simulator in plaats van een fysiek cloud-testbed is de kostprijs en de vereiste configuratie. In deze thesis kan er, met behulp van DevStack, snel en eenvoudig een OpenStack-cloud worden uitgerold op een fysiek cloud-testbed. Het voordeel van OpenStack is dat het een gratis softwarepakket is, compatibel met relatief goedkope hardware. Dankzij DevStack valt er veel configuratie weg en daarmee worden de twee grote redenen om geen fysiek cloud-testbed te gebruiken vermeden. Het te testen allocatieschema moet "vertaald" worden zodat het kan samenwerken met Nova, een OpenStack onderdeel verantwoordelijk voor het zware rekenwerk. Deze samenwerking tussen het te testen allocatieschema en Nova kan, dankzij de open-source code van OpenStack, eenvoudig gebeuren, mits enkele aanpassingen. Na het inpluggen van dat nieuwe schema kan er via de template een schaalbare applicatie worden uitgerold op de cloud. Deze schaalbare applicatie zorgt voor echte werklast op de fysieke hypervisors door het berekenen van enkele complexe fractalen waardoor de applicatie kan in- en uitgeschaald worden door de OpenStack-componenten Heat, Ceilometer en Aodh. De monitorapplicatie bewaakt dit gehele proces en bewaart statistieken in een databank. Nadien kunnen deze statistieken worden

opgevraagd zodat dit alles kan geëvalueerd worden.

Om dit alles te bewijzen is er gedurende deze thesis een Proof-of-Concept uitgewerkt met een Round-Robin allocatieschema. De resultaten hiervan worden besproken in Hoofdstuk **??** en zijn zoals verwacht aansluitend bij de gestelde hypothese. Round Robin is een simpel allocatieschema dat in sommige contexten zeer bruikbaar is, maar het grote nadeel blijft dat het volledig contextloos werkt.

Het eigenlijke besluit van deze thesis luidt dat via OpenStack uitgerold met behulp van DevStack, de schaalbare FaaFo-applicatie, Rally en de monitortechniek er een nieuw schema kan worden uitgetest en geëvalueerd. Het enige dat nog dient te gebeuren is het nieuwe schema inpluggen in Nova en deze service herstarten.

Bijlagen

Bijlage A - OpenStack's installatiehandleiding

Hier wordt een overzicht gegeven van de installatie van OpenStack met behulp van DevStack (commit 713f17c1d29f097d7d65e243c97a026867bf9363[1]) op een controller node en twee compute nodes. [**?**] [**?**] [**?**] De drie nodes zijn elk voorzien van 10 GB HDD, 8 GB vRAM en 2 vCPU's. Als standaard besturingssysteem wordt Ubuntu Server 16.04.02 LTS gebruikt.

Controller node

```
$ sudo apt-get update
$ cd /
$ sudo git clone https://git.openstack.org/openstack-dev/devstack
$ cd devstack/
$ sudo cp samples/local.conf local.conf
$ sudo vi local.conf
```

In local.conf plaatst men volgende gegevens:

```
[[local|localrc]]
^^IHOST_IP=192.168.16.118
^^IFLAT_INTERFACE=ens160
^^IFIXED_RANGE=10.4.128.0/20
^^IFIXED_NETWORK_SIZE=4096
^^IFLOATING_RANGE=192.168.16.128/25
^^IMULTI_HOST=1
^^IADMIN_PASSWORD=CaHiBaPa
^^IDATABASE_PASSWORD=CaHiBaPa
^^IRABBIT_PASSWORD=CaHiBaPa
^^ISERVICE_PASSWORD=CaHiBaPa

^^Ienable_service placement-api
^^Ienable_service placement-client

^^Idisable_service n-net
^^Ienable_service q-svc
^^Ienable_service q-agt
^^Ienable_service q-dhcp
^^Ienable_service q-l3
```

[1] https://git.openstack.org/cgit/openstack-dev/devstack/commit/?id=713f17c1d29f097d7d65e243c97a026867bf9363

```
^^Ienable_service q-meta
^^Ienable_service neutron
^^Ienable_service n-novnc #Automatisch geactiveerd sinds 13 maart 2017

^^Ienable_plugin heat https://git.openstack.org/openstack/heat

^^ICEILOMETER_BACKEND=mongodb
^^Ienable_plugin ceilometer https://git.openstack.org/openstack/ceilometer
^^Ienable_plugin aodh https://git.openstack.org/openstack/aodh
^^ICEILOMETER_PIPELINE_INTERVAL=60
^^ICEILOMETER_NOTIFCATION_TOPICS=notifications,profiler

$ sudo vi stackrc
^^I//Wijzig de GIT_BASE van git:// naar https://
$ sudo /devstack/tools/create-stack-user.sh
$ sudo chown -R stack:stack /devstack
$ sudo su stack
$ /devstack/stack.sh
```

De installatie zelf zal ongeveer een halfuur tot een uur in beslag nemen. Na de installatie geeft DevStack een overzicht van de gebruikers, de wachtwoorden en de URL's naar het dashboard en de identity service.
Een belangrijke en goede eigenschap van DevStack is dat het de verschillende OpenStack services start en monitort in verschillende terminals. Bij het gebruik van een SSH-connectie naar de controller node is het interessant dat ook deze schermen beschikbaar zijn voor eventuele debuginformatie of het heropstarten van bepaalde services. Hiervoor moet volgend commando uitgevoerd worden:

```
$ sudo chown stack:stack `readlink /proc/sefl/fd/0`
```

Dit commando geeft de stack-gebruiker toegang tot de verschillende terminals waardoor er eenvoudig via onderstaand commando verwisselt kan worden tussen de verschillende schermen, ook wel *screens* genaamd.

```
$ screen -x stack
```

Compute nodes

```
$ sudo apt-get update
$ cd /
```

```
$ sudo git clone https://git.openstack.org/openstack-dev/devstack
$ cd devstack/
$ sudo cp samples/local.conf local.conf
$ sudo vi local.conf
```

In local.conf plaatst men volgende gegevens:

```
[[local|localrc]]
^^IHOST_IP=192.168.16.117
^^IFLAT_INTERFACE=ens160
^^IFIXED_RANGE=10.4.128.0/20
^^IFIXED_NETWORK_SIZE=4096
^^IFLOATING_RANGE=192.168.16.128/25
^^IMULTI_HOST=1
^^IADMIN_PASSWORD=CaHiBaPa
^^IDATABASE_PASSWORD=CaHiBaPa
^^IRABBIT_PASSWORD=CaHiBaPa
^^ISERVICE_PASSWORD=CaHiBaPa
^^IDATABASE_TYPE=mysql
^^ISERVICE_HOST=192.168.16.118
^^ISERVICE_HOST_NAME=jerico-03
^^IMYSQL_HOST=$SERVICE_HOST
^^IRABBIT_HOST=$SERVICE_HOST
^^IGLANCE_HOSTPORT=$SERVICE_HOST:9292
^^IENABLED_SERVICES=n-cpu,c-vol,rabbit,neutron,q-agt
^^INOVA_VNC_ENABLED=True
^^INOVNCPROXY_URL="http://$SERVICE_HOST:6080/vnc_auto.html"
^^IVNCSERVER_LISTEN=$HOST_IP
^^IVNCSERVER_PROXYCLIENT_ADDRESS=$VNCSERVER_LISTEN
^^Ienable_service placement-api

^^ICEILOMETER_BACKEND=mongodb
^^Ienable_plugin ceilometer https://git.openstack.org/openstack/ceilometer
^^Ienable_plugin aodh https://git.openstack.org/openstack/aodh
^^ICEILOMETER_PIPELINE_INTERVAL=60
^^ICEILOMETER_NOTIFICATION_TOPICS=notifications,profiler


$ sudo vi stackrc
^^I//Wijzig de GIT_BASE van git:// naar https://
```

```
$ sudo /devstack/tools/create-stack-user.sh
$ sudo chown -R stack:stack /devstack
$ sudo su stack
$ /devstack/stack.sh
```

Bij de compute nodes zal de installatie minder lang duren als bij de controller node. Op het einde van de installatie zal het IP-adres van de compute node weergegeven worden. Net zoals bij de controller node wordt hier ook gebruik gemaakt van de screens in Linux en deze kan men bereiken via:

```
$ sudo chown stack:stack `readlink /proc/self/fd/0`
$ screen -x stack
```

Om de compute nodes instanties te laten hosten moet op de controller nog volgende twee commando's uitgevoerd worden:

```
$ nova-manage db sync
$ nova-manage cell_v2 discover_hosts
```

De compute nodes zijn nu zichtbaar vanaf het OpenStack Dashboard en ondersteunen het hosten van instanties.

Problemen met Cinder

In sommige gevallen is het mogelijk dat Cinder niet naar behoren werkt waardoor nieuwe volumes falen bij het aanmaken. Om dit probleem op te lossen wordt er best eerst gekeken naar de logs van cinder op 1 van de 2 compute nodes. Hier bevindt zich een volledige *stacktrace* van het probleem met een referentie naar het *loopback*-apparaat waarbij het fout loopt. Meestal zal de fout van volgende vorm zijn:

```
Stderr: u' /dev/loop1: lseek 4096 failed: Invalid argument\n Failed to
write VG stack-volumes-lvmdriver-1.\n'
```

Het probleem bestaat hier uit het feit dat er geen volume kan geschreven worden omdat er onder /dev/loop1 (in dit geval) geen bestandssysteem is. Dit kan eenvoudig opgelost worden door bijvoorbeeld een virtueel bestandssysteem aan te maken met volgende stappen:

```
$ sudo dd if=/dev/zero of=/devstack/filesyst bs=2G count=1
$ sudo losetup /dev/loop1 ./filesyst
```

```
$ sudo mkfs.ext3 /dev/loop1
$ sudo pvcreate /dev/loop1 -ff
$ sudo vgcreate stack-volumes-lvmdriver-1 /dev/loop1
```

Door het koppelen van een virtueel bestandssysteem van 2 GB (of groter) wordt bovenstaand probleem opgelost en kan cinder nieuwe volumes aanmaken.

Verwijderen van DevStack

Om DevStack volledig te verwijderen wordt best een back-up teruggeplaatst alvorens OpenStack via DevStack wordt geïnstalleerd omdat DevStack verschillende systeemaanpassingen doet die niet eenvoudig terugkeerbaar zijn. Indien er geen back-up beschikbaar is, kan DevStack grotendeels verwijderd worden via volgende stappen:

```
$ sudo su stack
$ /devstack/unstack.sh
$ /devstack/clean.sh
$ sudo rm -rf /opt/stack
```

Een reboot van het systeem is nadien soms nodig om de loopback-apparaten terug correct te laten werken.

Rally

Om Rally te gebruiken als benchmarking tool, moet deze worden geactiveerd als plugin op de controller node. Voeg hiervoor volgende lijn toe in local.conf van de controller node:

```
[[local|localrc]]
 # ... andere instellingen
 enable_plugin rally https://github.com/openstack/rally master
```

Bijlage B - Overzicht van de verschillende DevStack screens

Overzicht van de verschillende DevStack screens (= OpenStack-services) [**?**]

Service	Onderdeel	Uitleg
KeyStone	key	Weergeven van /var/log/apache2/keystone.log
	key-access	Weergeven van /var/log/apache2/keystone-access.log
Glance	g-reg	Het Glance-register (o.a. lijst van images)
	g-api	De API van Glance
Nova	n-api	De API van Nova
	n-cond	Zorgt dat de compute-nodes geen DB nodig hebben
	n-sch	De Nova-scheduler (waar komt nieuwe inst.?)
	n-novnc	De NOVNC-server (toegang tot de instanties)
	n-cauth	Console-auth: toegang tot de consoles van de inst.
	n-cpu	De eigenlijke compute-service (computing)
	placement-api	Zorgt voor de plaatsing van de instanties
Neutron	q-svc	De eigenlijke Neutron-service (netwerk)
	q-agt	Beheert de virtuele switch van de instanties
	q-dhcp	Zorgt voor DHCP-services
	q-l3	L3/NAT-forwarding tussen VM's en externe netw.
	q-meta	Extra services voor Neutron
Cinder	c-api	De API van Cinder
	c-sch	Scheduler: waar wordt vol. geplaatst?
	c-vol	De eigenlijke Cinder-service (opslag)
Horizon	horizon	Het dashboard
Heat	h-eng	De eigenlijke Heat-service (Orchestration)
	heat-api	De API van Heat
	heat-api-access	Bepaalt wie toegang heeft tot de API
	heat-api-cfn	AWS-style query API
	heat-api-cfn-access	Bepaalt wie toegang heeft de CNF-API
	heat-api-cloudwatch	De monitoring van de log-bestanden
	heat-api-cloudwatch-access	De monitoring van de log-bestanden
Ceilometer	ceilometer-acentral	Plaatst de gegevens op een queue
	ceilometer-api	De Ceilometer-API (Telemetry)
	ceilometer-anotification	Waarschuwt Heat bij alarmen
	ceilometer-acompute	Monitort de instanties op de compute-nodes
Aodh	aodh	De eigenlijke Aodh-service (Telemetry)
	aodh-api	De Aodh-API
	aodh-notifier	Waarschuwt Heat bij alarmen
	aodh-evaluator	Bepaalt of een alarm moet afgaan
	aodh-listener	Monitort de instanties

Bijlage C - Overzicht van de scripts in de FaaFo-template

De FaaFo-template bevat twee scripts voor het initialiseren van de instanties: één voor de FaaFo-controller en één voor de FaaFo-workers die hier worden weergegeven.
De FaaFo-controller zal volgende code uitvoeren na opstart:

```
#!/usr/bin/env bash
cd ~
git clone https://github.com/moeyerke/nodejs-agent.git
cd nodejs-agent
curl -L -s
↪  http://git.openstack.org/cgit/openstack/faafo/plain/contrib/install.sh |
↪  bash -s -- -i messaging -i faafo -r api
sudo su
rabbitmqctl add_user faafo guest
rabbitmqctl set_user_tags faafo administrator
rabbitmqctl set_permissions -p / faafo ".*" ".*" ".*"
```

Dit script zorgt ervoor dat FaaFo wordt geïnstalleerd als controller en lost het probleem van RabbitMQ op door het aanmaken (zie Sectie **??**) van nieuwe gebruiker.

De FaaFo-workers zullen volgende code uitvoeren na opstart:

```
#!/usr/bin/env bash
ip_addr1="$1"
ip_addr2="$2"
if [[ $ip_addr == 10* ]] then
^^Iip_addr=$ip_addr1
fi
if [[ $ip_addr2 == 10* ]] then
^^Iip_addr=$ip_addr2
fi
echo "Ip-adress is $ip_addr"
cd ~
git clone https://github.com/moeyerke/nodejs-agent.git
cd nodejs-agent
echo "Going to sleep for 2min so the controller is ready..."
sleep 2m
curl -L -s
↪  http://git.openstack.org/cgit/openstack/faafo/plain/contrib/install.sh |
↪  bash -s -- \
```

```
^^I-i faafo -r worker -e "http://$ip_addr" -m
↪  "amqp://faafo:guest@$ip_addr:5672/"
sleep 10s
supervisorctl restart faafo_worker
```

De eerste stap controleert de meegegeven ip-adressen zodat het IPv4-adres wordt gebruikt voor de communicatie tussen controller en workers. De reden hiervoor is dat OpenStack nog geen IPv6-regels voorziet in de security groups. Vervolgens wordt de FaaFo-applicatie geïnstalleerd als worker zodat de instantie fractalen kan berekenen die zich in de queue bevinden.