Elasticity in the Cloud

Ahmed El Rheddane

▶ **To cite this version:**

Ahmed El Rheddane. Elasticity in the Cloud. Other [cs.OH]. Université Grenoble Alpes, 2015. English. NNT : 2015GREAM003 . tel-01136131

HAL Id: tel-01136131
https://theses.hal.science/tel-01136131

Submitted on 26 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Ahmed El Rheddane

Thèse dirigée par **Prof. Noël de Palma**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Élasticité dans le Cloud Computing

Elasticity in the Cloud

Thèse soutenue publiquement le **25 février 2015**,
devant le jury composé de :

Prof. Noël de Palma
UJF, Directeur de thèse
Prof. Jean-Marc Pierson
LIP6, Rapporteur
Prof. Pierre Sens
IRIT, Rapporteur
Prof. Daniel Hagimont
ENSEEIHT, Examinateur
Prof. Jean-Marc Menaud
EMN, Examinateur

Abstract

Real world workloads are often dynamic. This makes the static scaling of resources fatally result in either the waste of resources, if it is based on the estimated worst case scenario, or the degradation of performance if it is based on the average workload. Thanks to the cloud computing model, resources can be provisioned on demand and scaling can be adapted to the variations of the workload thus achieving elasticity. However, after exploring the existing works, we find that most elasticity frameworks are too generic and fail to meet the specific needs of particular applications. In this work, we use autonomic loops along with various elasticity techniques in order to render different types of applications elastic, namely a consolidation service, message-oriented middleware and a stream processing platform. These elastic solutions have been implemented based on open-source applications and their evaluation shows that they enable resources' economy with minimal overhead.

Keywords. Cloud computing; autonomic loop; elasticity; consolidation; messaging; stream processing.

Résumé

Les charges réelles d'applications sont souvent dynamiques. Ainsi, le dimensionnement statique de ressources est-il voué soit au gaspillage, s'il est basé sur une estimation du pire scénario, soit à la dégradation de performance, s'il est basé sur la charge moyenne. Grâce au modèle du cloud computing, les ressources peuvent être allouées à la demande et le dimensionnement adpaté à la variation de la charge. Cependant, après avoir exploré les travaux existants, nous avons trouvé que la plupart des outils d'élasticité sont trop génériques et ne parviennent pas à répondre aux besoins spécifiques d'applications particulières. Dans le cadre de ce travail, nous utilisons des boucles autonomes et diverses techniques d'élasticité afin de rendre élastiques différents types d'applications, à savoir un service de consolidation, un intergiciel orienté messages et une plateforme de traitement de données en temps-réel. Ces solutions élastiques ont été réalisées à partir d'applications libres et leur évaluation montre qu'elles permettent d'économiser les ressources utilisées avec un surcoût minimal.

Mots-clés. Cloud computing; boucle autonomique; elasticité; consolidation; messagerie; traitement temps-réel.

Acknowledgments

My first thoughts go to my mother, Rabia Essarih, without whom neither this thesis nor its author would have existed.

My special thanks go to my advisor, professor Noël de Palma, for his genuine caring, extraordinary patience and valuable support throughout the duration of this PhD work. My thanks also go to André Freyssinet for his availability and help regarding the messaging part of this thesis.

I would also like to thank professor Jean-Marc Pierson, professor Pierre Sens, professor Daniel Hagimont and professor Jean-Marc Menaud for accepting to serve as my committee members as well as their precious feedback on my work.

A big thank you to all the members of the Erods team and former Sardes team, particularly those downstairs, and even more particularly those in the eastern office. I truly couldn't ask for better colleagues and friends.

I would also like to thank my family and friends for their constant and unconditional support.

Finally, I would like to express my deepest gratitude to all the teachers and professors that have made me, each in their own way, the person I am today.

Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Context

> *"If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry."*
>
> –John McCarthy, speaking at the MIT Centennial in 1961

Computation as a utility. This is the main concept behind the old, yet newly fashionable, cloud computing. Just as for the telephone or electricity grid, clients would be provided with on-demand computational resources on the "cloud", i.e., on the internet, and they would pay depending on their consumption. The physical infrastructure is entirely managed by the cloud provider, which provides its clients with just the right level of abstraction they need, saving them the trouble of handling themselves the issues related to the lower levels.

From the provider's perspective, cloud computing is an ideal way to leverage unused physical resources by "lending" them to consumers that are willing to pay for their use. This multitenancy aspect –a cloud has usually multiple clients– has

led to the virtualization of cloud computing infrastructures. Instead of managing multiple clients on the same physical machine, isolated virtual machine instances are assigned to each client and distributed over the available physical machines, which renders the infrastructure more flexible and easier to manage.

Cloud computing is nowadays widely adopted and data centers are being built everywhere. This naturally increases the energetic impact of such infrastructures and makes energy-efficiency one of cloud computing's most pressing challenges. On the providers' side, energy-efficiency can be achieved to some extent by optimizing the location of data centers to minimize the energy spent on cooling or maximize the use of renewable energy. More energy can be saved by optimally managing the resources within a given data center. However, Further energy-efficiency requires the clients' involvement as well, particularly with regard to the design of their applications.

1.2 Motivation

Cloud computing offers a virtually unlimited sky of distributed resources. To benefit from it, applications on the cloud need to be *scalable*, i.e., they should be able to integrate additional resources and use them efficiently to handle more important workloads. However, a scalable architecture is not enough to have an efficient cloud application.

Applications are often scaled statically, on a worst-case based scenario. While this may guarantee that a given application will meet its performance objectives at all time, it fails to do so efficiently. Real-world applications do not have a uniform workload, websites' traffic for instance changes of magnitude between day and night. In many cases, the workload is even bursty, i.e., it significantly grows in a relatively short period of time, this can be due to a product release or a worldwide event in social network related applications for instance, which makes even estimating the worst case scenario uncertain. Thus, performance-driven static scaling leads inevitably to over-provisioning, which means that, most of the time, an important part of the provisioned resources is simply wasted. This waste translates to both a financial cost for the user paying for the resources and, more considerably, a huge energetic footprint. On the other hand, Statically scaling an application based on its average workload would take the risk of degrading performance in the case of a workload burst, in which case the resources would be under-provisioned.

This issue can be solved thanks to cloud computing's on-demand provisioning paradigm, i.e., resources on the cloud can be provisioned and payed for only when needed. Thus, beyond static scalability, cloud applications should be flexible enough to allow users to add and remove resources as they see fit. This dynamic scalability can rely on various metrics, either those provided by the

cloud provider monitoring tools, or application metrics reported by the applications themselves. Thanks to autonomic computing techniques, these scaling operations can be achieved automatically, without the need of human intervention. This makes cloud applications seem to "magically" stretch their resources in order to adapt to their current workloads, no resources are wasted and none are needed, thus achieving both performance and cost-efficiency. This automatic scaling is known as *elasticity*.

Naturally, since the emergence of cloud computing, many elastic solutions have been developed, and most cloud providers offer an elasticity service, which takes scaling decisions based on defined thresholds on generic metrics such as CPU or memory usage. These generic elasticity tools fail oftentimes to characterize the load of the applications they intend to make elastic, which limits their usefulness. Therefore, elastic solutions should be tailored to applications, or at least application classes, and take into account their custom metrics and internal scalability mechanisms, in order to achieve even more efficiency.

1.3 Contribution

Provided the presented context, our contribution in this PhD work consists in the design, development and evaluation of four elastic solutions:

- *Elastic consolidation:* We made the infrastructure management service in charge of placing the virtual machine instances so as to minimize the number of running physical machines automatically adapt to the size of the cloud to handle, particularly regarding the varying number of virtual machine instances. This is based on the open-source consolidation tool Entropy. This work has led to a conference publication [31]

- *Elastic message queues:* We developed queues that can automatically scale based on the message production load, while maintaining protocol compatibility and reliability standards. This work has been accepted as a conference publication [32].

- *Elastic message topics:* In the same context of message-oriented middleware, we made message topics automatically adapt to the varying number of subscribers. Both our messaging solutions are based on Joram, an open-source message-oriented middleware.

- *Elastic stream processing:* Finally, we developed a stream processing platform with fine grained scaling that independently adjusts the parallelism of

the different components in the processing chain. This solution is an enhancement of the popular open-source stream processing solution Apache Storm.

1.4 Organization

After this introduction, the rest of the document is organized as follows:

Chapter 2 presents the literature related to cloud computing. It present its definition, assets, standard classification as well as the different challenges it faces. One of these challenges is energy efficiency which can be addressed using elasticity.

Chapter 3 sets the basis for elasticity as it presents the context of autonomic computing. It presents the different modules of the autonomic loop as well as the properties an autonomic manager aims to guarantee.

Chapter 4 concludes the state of the art by presenting an overview of elasticity in the literature. It presents a taxonomy of elastic solutions and illustrates it with a variety of related works.

The next chapters present our contribution in this PhD work. Chapter 5 presents our elastic consolidation service. Chapters 6 and 7 present our elastic messaging platforms and chapter 8 our Apache Storm based elastic stream processing.

Finally, chapter 9 concludes this document with a sum up and perspectives of future work.

Chapter 2

Cloud Computing

Contents

2.1 From the Grid to the Cloud

The growing speed of computer networks has made it possible to benefit from the combined computing power of large computer clusters. These have led to the

emergence of *grid computing*. As defined by the CERN[1],"the grid is a service for sharing computer power and data storage capacity over the Internet" [4]. This service can be centralized or distributed over multiple interconnected clusters in different locations as is the case of the French Grid'5000. Grids have been mainly targeted at large-scale scientific initiatives in the context of high performance computing, and can be fairly complex to use. Cloud computing overcomes this and offers the power of data centers to a broader public which does not necessarily have a computer science background.

Cloud computing is a relatively new paradigm and there has been many attempts to grasp its definition [9, 20, 25, 40, 27, 34], each focusing on a particular aspect of the cloud. In [9], some experts join the definition we proposed in our introduction and see cloud computing as the realization of utility computing, while others focus on its user-friendliness and on-demand scalability which distinguish it from grid computing. Knorr and Gruman [20] define cloud computing as the online services it provides and McFedries in [25] believes that what sets cloud computing apart is the massive data centers underneath.

Vaquero et al. [40] synthesize the different definitions they could find in the literature in the following proposition: *"Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs"*, SLAs being the Service Level Agreements that guarantee certain quality of service metrics for cloud users. This definition is also concordant with the ones Sarga presented later on in [34].

2.2 Key Characteristics

From the cloud computing's various definition, a certain set of key characteristics emerges. This section presents what we believe characterizes cloud computing the most, it does not only present the standard characteristics pinpointed by the NIST[2] in their definition [27] but tries to infer the most important features from all the previously cited works as well.

[1]European Organization for Nuclear Research, `http://home.web.cern.ch`
[2]National Institute of Standards and Technology, `http://www.nist.gov`

2.2.1 On-demand Provisioning

On-demand provisioning, we believe, is the single most important characteristic of cloud computing, it allows the users to request or release resources whenever they want. These demands are thereafter automatically granted by a cloud provider's service and the users are only charged for their usage, i.e., the time they were in possession of the resources. The reactivity of a cloud solution, with regard to resource provisioning, is indeed of prime importance as it is closely related to the cloud's pay-as-you-go business model.

2.2.2 Universal Access

Resources in the cloud need not only be provisioned rapidly but also accessed and managed universally, using standard Internet protocols, typically via RESTful web services. This enables the users to access their cloud resources using any type of devices, provided they have an Internet connection. Universal access is a key feature behind the cloud's widespread adoption, not only by professional actors but also by the general public that is nowadays familiar with cloud-based solutions such as cloud storage or media streaming.

2.2.3 Enhanced Reliability

Cloud computing enables the users to enhance the reliability of their applications. Reliability is already built in many cloud solutions via storage redundancy. Cloud providers usually have more than one data center and further reliability can be achieved by backing data up in different locations. This can also be used to ensure service availability, in the case of routine maintenance operations or the rarer case of a natural disaster. The user can achieve further reliability using the services of different cloud providers.

2.2.4 Measured Services

Cloud computing refers generally to paid services. The customers are entitled to a certain quality of service, guaranteed by the Service Level Agreement, that they should be able to supervise. Therefore, cloud providers offer monitoring tools, either using a graphical interface or via an API. These tools also help the providers themselves for billing and management purposes.

2.2.5 Multitenancy

As the grid before, the cloud's resources are shared by different simultaneous users. These users had to reserve in advance a fixed number of physical machines for a fixed amount of time. Thanks to the cloud's virtualized data centers, a user's provisioned resources no longer correspond to the physical infrastructure and can be dispatched over multiple physical machines. They can also run alongside another users' provisioned resources thus requiring a lesser amount of physical resources. Consequently, important energy savings can be made by shutting down the unused resources or putting them in energy saving mode.

2.3 Cloud Layers

Cloud services can be classified based on the level of abstraction they propose. There are three commonly accepted layers of abstraction in cloud computing, the services from the upper layers usually rely on the layers underneath but can also be provided as standalone services.

2.3.1 Infrastructure as a Service (IaaS)

The IaaS layer represents the lowest level of abstraction in the cloud. Users are provided with virtual machine instances that can differ in the operating system they are running, installed software as well as needed resources in terms of CPU –frequency as well as number of virtual CPUs–, memory, bandwidth and storage capacity. The IaaS provider also enables the users to manage the networking of their VM instances either by restricting their access or setting up virtual networks. Furthermore, the users can as well create snapshots of their instances as a means of backup, or use the snapshot of a running VM instance to create a VM template which can be later used to provision new clone instances.

Amazon has been a pioneer of IaaS with its Elastic Compute Cloud (EC2)[3]. It nowadays provides three pricing models for its VM instances: (i) users can classically pay for their *on-demand instances* on an hourly basis, (ii) via a prepaid fee, users can pay less for reserved instances that they plan on using heavily or (iii) users can bid the price they are willing to pay for a *spot instance* and as long as the price of these instances, computed over the different bids of the various users, is less then the one they specified, the instances are provisioned. Naturally, spot instances are to be used solely in order to optionally enhance performance as they can be lost anytime should their computed price change. The users are also

[3] Amazon EC2, `http://aws.amazon.com/ec2`

charged for out-going traffic and storage related to the instances disks' or snapshots. Other IaaS providers such as Microsoft Azure[4] or Google Compute Engine[5] only provide on-demand instances with roughly the same features although they might charge usage per minute instead of Amazon's hourly increments.

2.3.2 Platform as a Service (PaaS)

In order to save the users the trouble of managing themselves the installation and configuration of their machines, virtual as they may be, an extra level of abstraction is needed. Platform as a Service provides the users with integrated environments on which they can develop, test and deploy their cloud applications. PaaS services can also include features related to the maintenance as well as the scalability of these latters. An example of a PaaS solution can be Google App Engine[6] which provides users with familiar development tools as well as underlying services such as cloud-based databases and cache mechanisms.

2.3.3 Software as a Service (SaaS)

At its highest level of abstraction, the cloud's SaaS layer provides the users with ready to use applications, hosted on the cloud. The users are then completely freed of the development and maintenance burdens. This is particularly interesting for professionals that rely on IT for management purposes whereas it is not their core business. SaaS virtually encompasses any website, but usually refers to heavy applications hosted on the Internet such as Salesforce customer relationship management service[7] or the better known Google Docs office suite.

While these layers remain the standard classification of cloud services, specialized services may be referred to as *XaaS* but can still be ranged in one of the above categories.

2.4 Deployment Models

The previous section classified cloud computing with regard to the level of abstraction of its services. In this one, we show the different classes of cloud computing based on their deployment, purpose, and target users.

[4]Microsoft Azure, `https://azure.microsoft.com`
[5]Google Compute Engine, `https://cloud.google.com/compute`
[6]Google App Engine, `https://cloud.google.com/appengine`
[7]Salesforce, `http://www.salesforce.com`

2.4.1 Public Clouds

Unless otherwise specified, clouds refer to public clouds. These consist in huge interconnected data centers managed by a cloud provider and open to use by the general public on a charge-per-use basis. Public clouds are the incarnation of utility computing as they enable the users to rent their computing power instead of buying and maintaining it. This is made possible thanks to the economies of scale that only data centers of important sizes can afford.

2.4.2 Private Clouds

As their name indicates, private clouds are privately managed by an organization for its own use. These may or may not support the cloud's charge-per-use model, but they benefit from cloud infrastructures' flexibility and user-friendliness while guaranteeing the privacy of sensitive data. Many tools exist to leverage physical clusters into Infrastructure as a Service private clouds either open-source such as OpenStack[8] and Eucalyptus[9], both compliant with the Amazon EC2 API, or commercial as VMware's vSphrere[10].

2.4.3 Hybrid Clouds

Privacy and data security are one of the main motivations for private clouds. However, most of an organization's data and operations are not confidential. Therefore, while having a private cloud for sensitive operations, an organization can benefit from the resources of a public cloud when a high level of confidentiality is not required, thus forming a hybrid cloud. Hybrid clouds can also be put in place as a result of cloud bursting: when the private cloud is short of resources, the extra load can be redirected to a public cloud. More generally, a hybrid cloud can be formed by any two distinct clouds, private or public, in order to enhance reliability or reduce the cost.

Hybrid clouds rise the issue of clouds' compatibility, or the lack thereof, as a standard cloud API is yet to see the light of day. The next section details some other challenges that currently face cloud computing.

[8] OpenStack, `http://www.openstack.org`
[9] Eucalyptus, `https://www.eucalyptus.com`
[10] VMware vSphere, `http://www.vmware.com/products/vsphere`

2.5 Challenges

Due to its lack of maturity, many challenges still hinder cloud computing's full adoption, these range from technical aspects as presented in [3] to more general ones detailed in [34].

In order to benefit from the power of the cloud, a potential user has to migrate its existing applications to the cloud. However, the lack of an established cloud standard puts the users in a potential vendor lock-in as they would be tied to the cloud service they first used. Once applications and data are in the cloud, they have to be both secured and available. Security is still a top concern as, beyond identity management tools that are commonly put in place by cloud service providers, data should be encrypted, which fairly degrades the performance of the cloud. Availability can be enhanced by the use of hybrid clouds which is, once again, hampered by the lack of a standard. For cloud applications to run properly, the cloud's resources need to be able to scale quickly and efficiently, particularly with regard to storage and networking. Last but not least, as cloud applications generally run in data centers that are big enough to offer the illusion of infinite resources, optimizing energy efficiency is all the more important. This is the motivation behind this work and will be presented in chapter 4.

2.6 Conclusion

In this chapter, we have seen how cloud computing offers its users the possibility to focus on their core businesses by taking care of all the IT complexities underneath. In order to do that at the large scale on which clouds generally operate, automation is inevitable. The next chapter presents an overview of autonomic computing, which is behind most cloud management tasks, and would be basis on which we build our elasticity solutions.

Chapter 3

Autonomic Computing

Contents

3.1 Definition

Computer systems have been the ideal solution to replace tedious and complex operations previously entitled to human agents. But as the formers become themselves bigger and more complex, their administration and maintenance gets out of hands. Thus the need for autonomic systems capable of managing themselves. In a 2001 manifesto [14], IBM was the first to coin the concept of *autonomic computing*. To lay the basis for the then new paradigm, IBM set an architecture for autonomic managers in the form of an autonomic loop. It also pointed out the different properties of an autonomic system, each corresponding to a particular purpose. The next sections detail these different elements.

3.2 Autonomic Loop

In [16], IBM proposed a model for autonomic control loops. This model details the different components that allow an autonomic manager to achieve one of the previously discussed self-managing properties, namely Monitor, Analyze, Plan, Execute and Knowledge. The *MAPE-K* loop is depicted by figure 3.1 and discussed in the rest of this section.

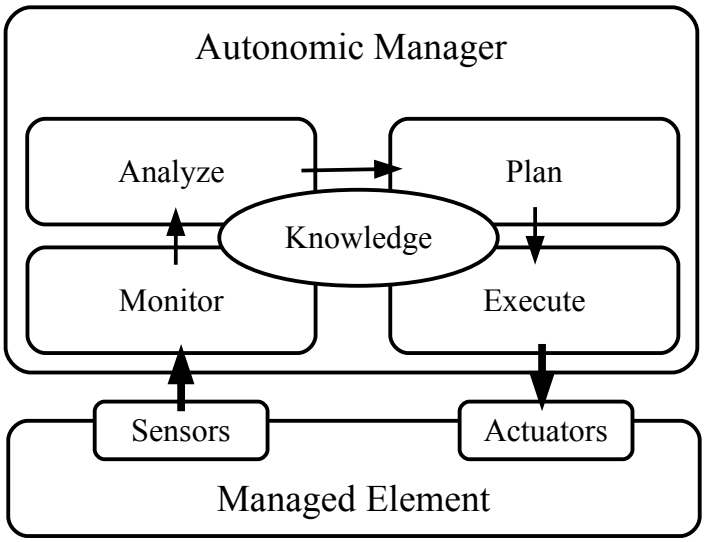

Figure 3.1: Autonomic systems' control loop

The autonomic manager regularly monitors the managed element. This allows the former to be up to date with regard to the evolution of the latter. Monitoring can be done either passively, using provided system tools such as `top` for instance, or actively, i.e., by modifying the code of the managed elements in order to enrich the monitored information. Once the current state of the managed element is captured, it is analyzed and a plan of the changes to be applied is established. Planning follows rules that can be as simple as an event-condition-action policy (ECA), which is easy to implement and fast to compute, or take the form of utility functions, which try to optimize a given property of the managed systems and are, thus, much more compute-intensive and hard to put in place. Planning naturally relies the knowledge available to the autonomic manager. Knowledge is usually a representation of the system. It can be statically put into the manager upon its

creation but can also evolve by including the history of monitored values which can be used by predictive policies. Once a plan is established it is executed on the managed system and the autonomic manager loops back.

Many implementations of the autonomic loop have been made. For instance, IBM provides a framework for autonomic managers' development called Autonomic Computing Toolkit [26] while Parekh et al. [30] present another implementation that focuses on adding autonomic capabilities to legacy applications.

3.3 Autonomic Properties

Self-management properties introduced by IBM have been detailed by Kephart and Chess in [18]. They define the different purposes an autonomic system may have and are detailed below.

3.3.1 Self-configuration

The self-configuration property enables a given system to reconfigure itself based on high-level goals. An administrator would only have to specify a desired outcome and the system would automatically adapt to accomplish it. This can include, for instance, the recognition and integration of new components that would automatically join an existing infrastructure.

3.3.2 Self-healing

An autonomic system should be able to detect and diagnose its problems. These may range from software bugs to hardware failures. If possible, the system tries to fix them. If no solution can be found, the system should report the problem to be fixed by a human administrator.

3.3.3 Self-protecting

The last autonomic property makes a system protect itself from both external intruders or repetitive failures that couldn't be handled by self-healing. Ideally, a self-protecting system would proactively anticipate security threats and act accordingly.

3.3.4 Self-optimization

Self-optimization is the ability to continually seek ways of optimizing efficiency either with regard to performance or cost. This is typically the case of elasticity in

the cloud. Optimization operations can be reactive to the environment's state but can also be initiated proactively.

Autonomic systems may exhibit one or more of these properties. They usually have an autonomic manager for each one of them. In order to avoid conflicts, an extra autonomic manager may be used for coordination.

3.4 Conclusion

In this chapter, we have presented a brief overview of autonomic computing, including its characteristic MAPE-K loop and the different properties it aims to achieve. A more thorough presentation of this field has been proposed by Huebscher and McCann in [15]. In the next chapter, we will see how autonomic computing techniques are used to insure elasticity, for it is nothing but an instance of the autonomic MAPE-K loop with the purpose of achieving self-optimization.

Chapter 4

Elasticity

Contents

4.1 Beyond Static Scalability

Since the emergence of parallel and distributed systems, a great effort has been put into making applications benefit efficiently from multiple computing resources. Scalability, as defined in [17], characterizes this ability. It is measured by speedup, i.e., the performance gain due to additional resources and efficiency, which shows to which extent the resources are made useful and is given by the ratio of the speedup over the amount of used resources. These can be used to scale a given application, that is provision its resources, based on the expected workload.

As the workloads tend to significantly vary over time, scalability needs to adapt automatically, in order to use just enough resources. Thanks to autonomic

computing techniques, elasticity does just that. According to [11], *"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible."* From this definition, we can see that efficiency, or precision when it comes to matching the demand is still a top concern, as it avoids any waste of resources. Elasticity also introduces a new important factor, which is the speed. Rapid provisioning and deprovisioning are key to maintaining an acceptable performance and are all the more important in the context of cloud computing where quality of service is subjected to a service level agreement.

In this chapter, we present the different approaches through which elasticity can be achieved as well as a variety of existing elastic solutions, particularly those related to our contribution.

4.2 Classification

As proposed by Galante and de Bona in [7], elasticity solutions can be arranged in different classes with regard to their scope, policy, purpose and method. Figure 4.1 presents this classification and the remaining of this section is dedicated to discuss it.

4.2.1 By Scope

With regard to scope, elasticity can be implemented on any of the cloud layers. Most commonly, elasticity is achieved on the IaaS level, where the resources to be provisioned are virtual machine instances. Other infrastructure services can also be scaled such as networks [39]. On the PaaS level, elasticity consists in scaling containers or databases for instance. Finally, both PaaS and IaaS elasticity can be used to implement elastic applications, be it for private use or in order to be provided as a SaaS. Most cloud providers offer elasticity mechanisms as part of their services, although these mechanisms alone tend to be generic and fail to provide an efficient framework for applications other than web servers.

4.2.2 By Policy

With regard to policy, the authors of [7] believe that elastic solutions can be either manual or automatic. A manual elastic solution would provide their users with tools to monitor their systems and add or remove resources but leaves the scaling decision to them, we believe that this cannot qualify as elasticity as the latter has to be carried out automatically. Hence, elastic solutions can be either reactive or

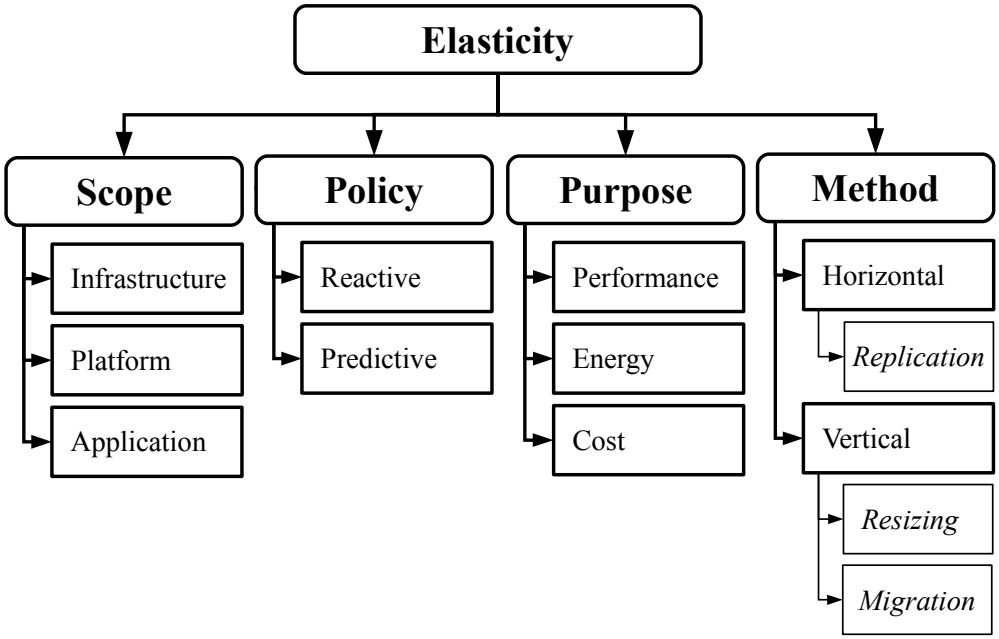

Figure 4.1: Classification of elastic solutions

predictive. An elastic solution is reactive when it scales *a posteriori*, based on a monitored change in the system. These are generally implemented by a set of Event-Condition-Action rules. A predictive –or proactive– elasticity solution uses its knowledge of either recent history or load patterns inferred from longer periods of time in order to predict the upcoming load of the system and scale according to it.

4.2.3 By Purpose

An elastic solution can have many purposes. The first one to come to mind is naturally performance, in which case the focus should be put on their speed. Another purpose for elasticity can also be energy efficiency, where using the minimum amount of resources is the dominating factor. Other solutions intend to reduce the cost by multiplexing either resource providers or elasticity methods. Mixed strategies that take into account different purposes and try to optimize a corresponding utility function have also been put in place.

4.2.4 By Method

Finally, with regard to method, elasticity can be carried out either horizontally or vertically [39]. Horizontal elasticity consists in replication, i.e., the addition –or removal– of virtual machine instances. In this case, a load balancer with an appropriate load balancing strategy needs to be involved. This is the most commonly used method as on-demand provisioning is supported by all cloud providers. Vertical elasticity, on the other hand, changes the amount of resources linked to existing instances on-the-fly. This can be done in two manners. The first one consists in explicitly redimensioning a virtual machine instance, i.e., changing the quota of physical resources allocated to it. This is however poorly supported by common operating systems as they fail to take into account changes in CPU or memory without rebooting, thus resulting in service interruption. The second vertical scaling method involves VM migration: moving a virtual machine instance to another physical machine with a different overall load changes its available resources *de facto*. Once again, migrating a virtual machine on-the-fly, a.k.a., live migration, is technically limited as it can only be achieved in environments with network shared disks.

4.3 Existing Solutions

4.3.1 Elastic Infrastructures

While all cloud infrastructures offer APIs that allow both monitoring and on-demand provisioning of resources, most do not offer an automated elasticity service. Amazon, however, has integrated an elasticity mechanism to its Elastic Cloud Compute (EC2)[1]. An EC2 user can set an Auto Scaling Group (ASG) containing an initial number of instances, he then defines Event-Condition-Action (ECA) rules for scaling, i.e., adding or removing instances from the ASG, based on the system metrics provided by Amazon CloudWatch, Amazon's monitoring tool. Amazon also allows the user to specify elastic load balancers that forward requests to any of an ASG instances. This makes Amazon Auto Scaling a reactive replication-based elasticity solution. Reservoir [2], an open-source IaaS manager, allows the specification of these same ECA elasticity rules as an extension of the Open Virtual Format standard [5]. As for the clouds with no elasticity capabilities, cloud management services such as RightScale[3] and Scalr[4] offer the same reactive

[1] Amazon Auto Scaling, `http://aws.amazon.com/autoscaling`
[2] Reservoir, `http://www.reservoir-fp-7.eu`
[3] RightScale, `http://www.rightscale.com`
[4] Scalr, `http://www.scalr.com`

replication-based mechanisms and present the advantage of being able to stretch over different cloud providers.

Lim et al. [21] discuss the use of thresholds in reactive policies and introduce *proportional thresholds*. The use of a single metric goal can lead to oscillation, i.e., repetitive addition and removal of resources, which can be overcome by the use of two thresholds in order to specify a range goal. However the speedup when adding a second instance, which is roughly 100%, can not be comparable to the addition of the 101st one, which is only 1%. This has to be taken into account when scaling, thus the need for goals, i.e., thresholds, that are adaptive to the current size and performance of the clusters to be scaled.

Gong et al. [10] present PRESS, an elasticity controller that analyzes the workload history using Fourier Fast Transform to detect any repeating pattern which would allow future workload prediction. If no pattern can be found, PRESS uses a discrete-time Markov chain to predict the near future workload. In order to avoid resources under-estimation, which may lead to under-provisioning, PRESS pads the predicted values by a small amount (5-10%). Roy et al. [33] use another statistical model, autoregressive-moving-average, in order to predict future load based on previously witnessed loads.

Vasić et al. [41] propose DejaVu, an elasticity framework based on a predictive policy. The idea behind DejaVu is to organize the different workloads that have been encountered into discrete classes using k-means clustering, remember the preferred allocation setup for each class, i.e., the different virtual machine instances used along with their sizes, and assign each class a signature. Thus, when the system runs, each time it detects a workload change, it computes its signature, go back to its *cache* and fetch the preferred allocation that it applies at once, thus accelerating the allocation of the needed resources.

Meng et al. [28] propose Tide, a tool that offers elasticity for IaaS management itself. The main idea behind Tide is to use the cloud's resources themselves to handle the cloud's management workload. They implemented a custom predictive model that focuses on provisioning needed resources all at once and as soon as possible.

While the previous solutions mostly focus on performance, Sharma et al. [36] propose Kingfisher, a cost-driven elasticity tool. Kingfisher takes into account the costs of different virtual machines instances' configurations and try to optimize the overall cost, for a given workload. It also takes into account the cost of the transition form a given VM instances configuration to another, upon a workload change.

The above works propose different solutions to the elasticity problem. However, as they are meant to be generic, they fail to fit the specific need of particular applications.

4.3.2 Elastic Platforms and Applications

Elastic platforms as elastic infrastructure, tend to be generic enough in order to target a wide range of applications, typically web servers and multi-tier applications. This is the case for the two leading PaaS providers, namely Google AppEngine and Microsoft Azure. Another elastic web hosting platform is the Cloud Hosting Provider proposed by Fitó and Guitart [6], which consists in extending physical resources with resources on the cloud in a cloud bursting fashion.

As for specific elastic solutions, in the context of messaging for instance, Tran et al. [38] propose Elastic Queue Service (EQS). EQS scaling actions can be triggered by a set of predefined Key Performance Indicators (KPIs). These include throughput as well as the number of connected clients. Scaling involves adding extra queues on extra instances and relocating clients' connections to balance their load on the existing queues. A limitation of this approach is that, at all times, all the messages produced by a single client go the same queue instance whereas it might be better to balance the messages over the different instances.

Taton et al. [37] propose a different approach to scaling messaging queues. Instead of balancing clients' connections over a set of independent queues, they interconnect the different instances of a queue in a cluster. These different queues balance their loads seamlessly in a work-stealing-like manner, i.e., underloaded queues *steal* messages from the overloaded ones. Scaling occurs when the whole cluster is overloaded or underloaded. Where as this approach is interesting, the fact that a message on an overloaded queue has to wait to be stolen does introduce an overhead, moreover, as clients connections are fixed, scaling in, i.e., reducing the number of instances, has to wait for the last client on a queue to disconnect before deleting it.

Stream processing has also been enhanced with elasticity as in the work proposed by Schneider et al. [35]. The authors base their work on a code generation tool for stream processing [1, 8]. The authors present an elastic operator and propose to adapt the number of threads it is assigned to its changing workload. This can naturally increase the performance provided that the total capacity of the underlying resources is not reached.

Vijayakumar et al. [42] focus on adapting CPU allocation to the loads of a generic stream processing application, which is an example of vertical elasticity. Their model involves different linear stages of processing and induces the load of each stage from the difference between its incoming and outgoing flows. This has been inspired by a previous work on TCP congestion [29].

Knauth and Fetzer [19] also apply vertical elasticity to stream processing, this time using migration. The main idea is to gather the processing instances on one physical machine and stretch them over more resources as workload increases.

4.4 Conclusion

In this chapter, we have presented the different approaches used to achieve elasticity in the literature. We have also discussed a representative range of elastic solutions, with focus on those related to our contributions. As we have seen, generic elasticity frameworks do not necessarily fit specialized applications. In the next chapters, we detail our contribution which consists in the development of different specific elastic solutions based on open-source tools of different horizons.

Chapter 5

Elastic Consolidation

Contents

5.1 Context

5.1.1 Consolidation

One of the top concerns when dealing with large datacenters, which is the typical case of cloud infrastructures, is energy efficiency. This aspect of the so-called

green computing, or environment-friendly computing, is already promoted by cloud computing, since gathering the computational resources in one big centralized infrastructure can considerably decrease the energy consumption. However, since cloud computing often relies on a virtualized infrastructure, further energy efficiency can be achieved through "consolidation", which is basically minimizing the number of running physical machines by optimally placing the virtual machines of the datacenter. Since the cloud ecosystem is highly dynamic and the virtual machines' needs in terms of resources (memory and CPU) vary, consolidation has to be done periodically.

5.1.2 Entropy

Our work is based on Entropy [13] which is an open-source consolidation manager. Based on a given infrastructure's configuration (the mapping of the virtual machines on the physical nodes, the virtual machines' current memory and CPU usage and the nodes' total CPU and memory capacity), it would try to minimize the cloud's energy consumption and computes a new mapping, i.e., new configuration, that would minimize the number of used nodes while guaranteeing enough resources for the virtual machines to run normally, and then reconfigure the system accordingly by issuing the appropriate migration commands. In its current version, Entropy can either be executed periodically, as the system's configuration is highly dynamic in the context of cloud computing, particularly regarding the number and placement of virtual machines, or on an event-driven basis, if we want to insure a greater reactivity. Figure 5.1 depicts Entropy's reconfiguration loop.

Computing a new configuration is a linear optimization problem. In order to solve it, Entropy uses ChocoSolver[1]. The optimization problem does not only compute the minimal number of nodes needed to run the current virtual machines, but computes a consolidation plan as a whole, which also minimizes the cost of reconfigurations leading to such an optimal configuration. The cost of a reconfiguration roughly depends on the number of migration operations needed to go from a given configuration to the newly computed one. Entropy also supports live migration to execute the computed reconfiguration plan, which makes the execution fairly faster but requires the virtual machines' disks to be stored in a distributed file system such as NFS. More details on how Entropy works can be found in [13] or in its newer version [12].

While Entropy is undeniably a useful tool, the fact that it relies on solving a linear optimization problem makes it potentially unable to compute an acceptable reconfiguration for realistic configurations' sizes within reasonable timeouts,

[1] ChocoSolver, `http://choco-solver.org`

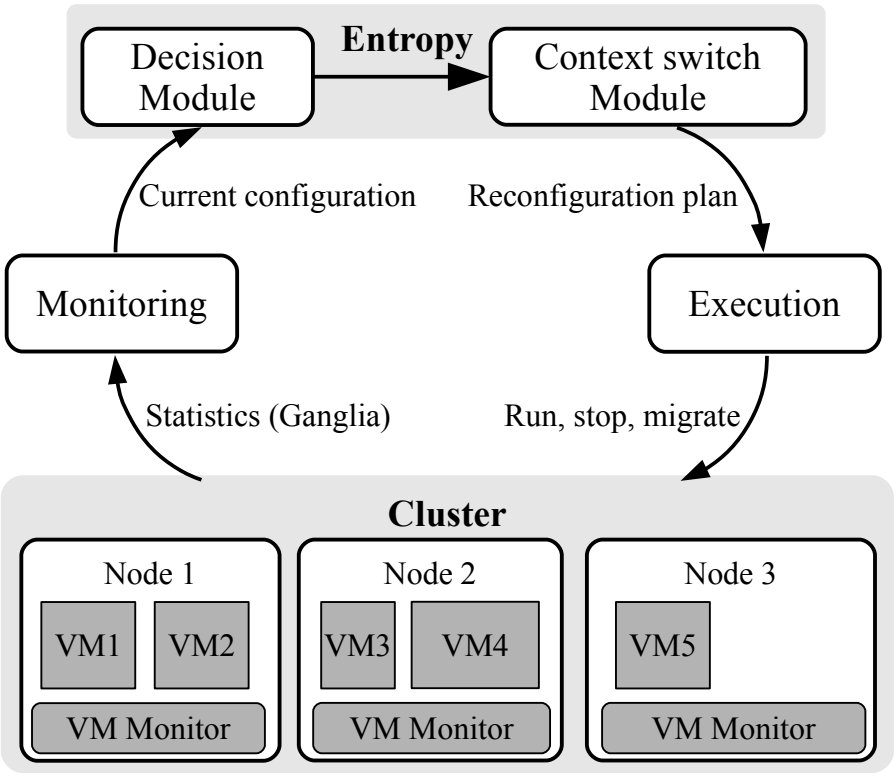

Figure 5.1: Entropy's reconfiguration loop

hence the need to enhance its scalability. Since the cloud's configuration and scalability vary dynamically, we propose to scale Entropy dynamically as well.

5.2 Approach

The scaling of Entropy, as a service provided by the IaaS infrastructure, is driven by the dynamic evolution of both the size and population of the datacenter, which corresponds to the number of virtual machines and physical nodes. An important point is that we do not take into consideration scalability requirements with regard to the clients' requests load.

The main principle behind our solution is to create just enough Entropy instances so as to adapt to the configuration to be processed. To do so, we use (i) a classical autonomic loop design, (ii) node partitioning for scalability purposes and (iii) virtualization in order to enhance the flexibility of managing the Entropy instances and insure elasticity.

5.2.1 Autonomic Loop

Our solution is an autonomous Entropy, that is capable of self-optimization, i.e., creating or deleting instances of itself so as to meet performance requirements. Thus, it has to be both aware of its environment, and adaptive to its changes. It interacts with its environment by the means of a classical MAPE-K control loop. This control loop regularly reports information from the environment to an autonomic manager which analyses it, takes decisions, and applies the necessary changes. As Entropy already has an autonomic loop which periodically retrieves the system's configuration in order to optimize it, with regard to virtual machine instances placement, we integrated our elastic controller which relies the this very same configuration to plan elasticity related actions.

Our elasticity loop works as follows: the information retrieved is the configuration of the system. After analysis, the control loop decides how many Entropy VM instances ,i.e., workers, are necessary to handle the retrieved configuration; execution finally creates the workers, launches them and distributes the configuration to be optimized over the launched workers.

5.2.2 Partitioning

To distribute the configuration over the virtual workers, we made the choice of having independent sub-configurations so as to avoid the very costly synchronization of the whole-system's state between our different workers. To do so,

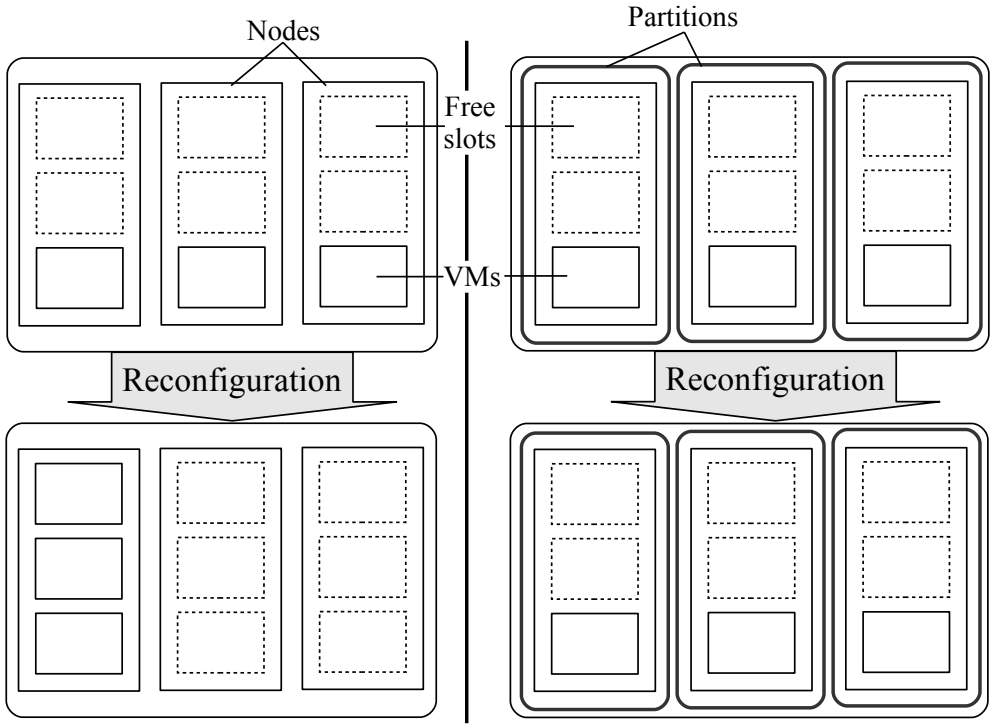

Figure 5.2: The effect of partitioning on consolidation

the currently implemented policy is random node partitioning, which furthermore guarantees that our workers will statistically have the same workload.

Once the number of necessary instances, that we call *workers* in the following, is computed, the nodes are randomly distributed between the different groups. This means that our partitioning is not persistent, even if two consecutive iterations require the same number of instances, the computed sub-configurations will not necessarily be the same. This is possible because Entropy does not store the system's state (mapping, CPU and memory usage), and retrieves it via the monitoring tool [24] anyway. Randomness guarantees load balancing; since the nodes are not equally populated, it guarantees that, statistically, the random groups will have about the same number of virtual machines.

Naturally, partitioning will affect the consolidation's result, and the more partitions we have, the less effective our consolidation is. It is true that load balancing significantly attenuates this effect, but we would still have an error up to the number of created partitions, on the minimum number of nodes needed to run the current virtual machines, as shown by Figure 5.2.

We can see that consolidating the whole cloud enables us to free two physical resources, whereas partitioning the cloud to three sub-clouds makes it impossible for this optimization to take place. However, our evaluation shows that this effect can be neglected in the context of big enough configurations, in which dynamic scalability makes sense.

5.2.3 Virtualization

The created Entropy instances are virtualized, i.e., each instance runs on a dedicated virtual machine. While it might seem that all we do is parallelize the computation as could be done using a powerful multi-core machine, we argue that virtualization for computation's sake, allows us to benefit from the cloud's resources themselves, since the created instances would be on the cloud's nodes. Besides, even pure computational tasks can benefit from virtualization as we witness the emergence of virtualized grids: Haizea[2] for instance, allows scheduling and management of virtualized jobs, i.e., jobs running on virtual machines, to benefit from the increased flexibility and reliability.

Since the cloud will contain both Entropy virtual instances and the clients' VMs and to avoid any confusion in the remaining of this paper, the term VM refers to a client's VM unless we explicitly specify that it refers to an Entropy VM instance.

5.2.4 Scaling

The estimation of necessary Entropy workers is done using the results of prior "gauging" of Entropy: we measure the performance of Entropy with gradually increasing workload and number of workers; this is done once and for all. Note that we do not rely on a predictive algorithm to pre-provision the Entropy workers, since the information that is needed to foresee the right number of instances is itself the information that is processed by Entropy. With this regard, any prediction would only shift the period of our process.

Scaling up can only be done by provisioning extra instances. However, in the case of Entropy, where two iterations are completely independent as Entropy doesn't keep knowledge of the system's state, scaling down can be done in two different ways:

- At the end of each iteration, delete all created instances. This way, every iteration will create just the right number of instances it needs.

[2]Haizea, `http://haizea.cs.uchicago.edu`

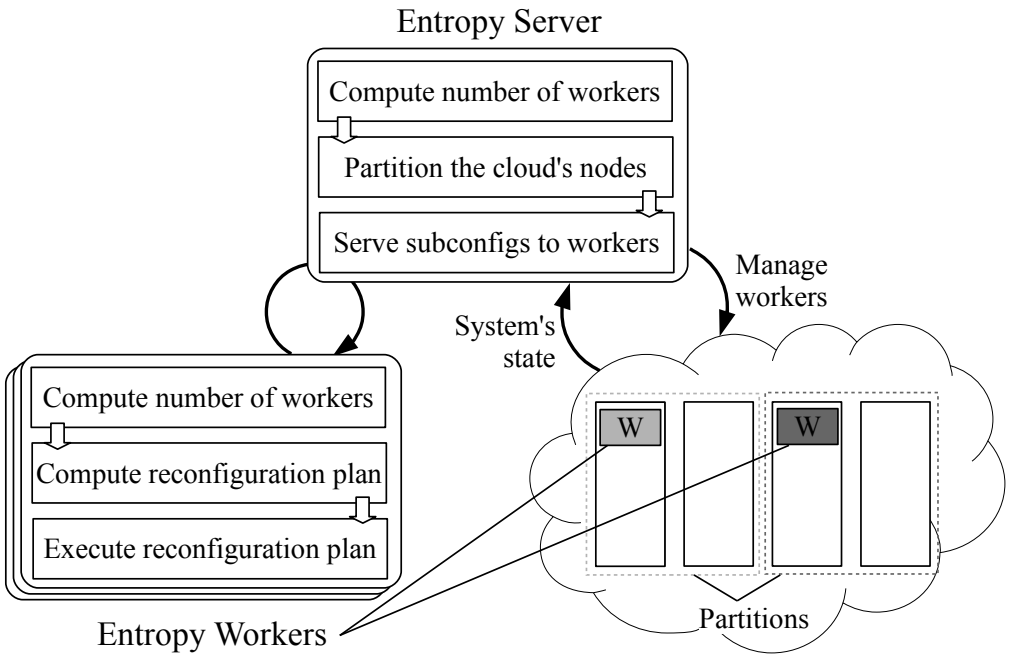

Figure 5.3: Dynamically scalable Entropy

- Instances persist after their creation, and are only deleted if they haven't been needed for the last N iterations, N to be determined empirically.

The final design is given by Figure 5.3. In this design, the original Entropy has been cut into two parts:

- *The Entropy Server,* that is centralized, which performs the partitioning and triggers the creation, deletion or reconfiguration commands to the hypervisors.
- *The Entropy Worker,* which consolidates the sub-configuration it is assigned and issues the migration commands to its' sub-configuration's hypervisors.

The communication between the centralized part and the VM instance is done on a poll basis: once an Entropy VM instance is created, it retrieves its sub-configuration and starts processing it immediately.

5.2.5 Performance Gain

Let S be the size of the cloud (i.e., the number of virtual machines $\times$ the number of physical machines) and $Ent(S)$, the cost of the original Entropy. The cost of

our virtualized version is given by:

$$Ent_{virt}(S) = Ent(S_1) + C_{virt}$$

where S_1 is the size that has been found to be the appropriate load for one entropy instance and C_{virt} is the cost of provisioning an Entropy virtual instance, it is not a function of the number of instances to be created since provisioning will not take place on the same nodes, and can thus be achieved in parallel. Moreover, this constant is amortized thanks to the relative persistence of our virtual instances.

Thus, our approach guarantees a constant cost, with only a small trade-off due to the partitioning mechanism, whereas applying Entropy to the cloud as a whole grows exponentially with its size. He has never seen me work.

5.3 Evaluation

In this section, after briefly presenting the technical context and the methodology of our experiments, we discuss their results in order to validate the efficiency of our elastic consolidation manager.

5.3.1 Technical Context

The infrastructure used for our experiments is a private cloud managed by OpenStack, which is an EC2 compatible IaaS management solution, running on 2 racks with 6 Intel(R) Xeon(R) CPU E5645 @2.40GHz cores and 32 GB of RAM each, interconnected by a 1 Gbit/s isolated LAN. All the managed virtual machine instances are of type m1.small as described by Amazon EC2, i.e., with 2 GB of memory and 1 virtual CPU.

Note that this infrastructure is only used to host the Entropy VM instances, the configurations being consolidated are generated in order to have loads big enough to stress the scalability of Entropy.

5.3.2 Methodology

With this evaluation, our aim is to:

- see if we actually achieve any performance gain by using more workers, keeping in mind the limitation discussed in Section 5.2.2;
- show the interest of having an elastic consolidation manager.

To do so, we consider a cloud infrastructure of 200 homogeneous physical machines, each having 3GB of memory, consuming 100W when idle and at most 200W. For this infrastructure, we generate different virtual machines which have variant requirements (CPU usage: 20%, 40% or 60%; memory: 500MB, 1GB or 2GB).

We first vary the load of our cloud infrastructure and compute the optimal consumption using 1 up to 10 Entropy workers; the load is given by:

$$load = max\left(\frac{totalMemory}{totalMemCapacity}, \frac{totalCpu}{totalCpuCapacity}\right)$$

Our metric when comparing the results given by the different sizes of workers is the error with regard to the theoretical optimal consumption, that is:

$$error = \frac{consumption - optimalConsumption}{optimalConsumption}$$

Note that the theoretical optimal consumption in which we use just enough physical machines is straightforward to have as it is given by:

$$A = totalCpu \times \frac{consMax - consMin}{100}$$

$$B = ceil\left(\frac{totalCpu}{totalCpuCapacity}\right) \times consMin$$

$$optimalConsumption = A + B$$

where A linearly computes the ratio of consumption based on the total ratio of CPU usage and B multiplies the minimum number of necessary physical machines by their minimum consumption.

What is actually hard is to find a configuration that corresponds to this optimal consumption while maintaining the constraints related to the needs of each virtual machine and the capacities of the physical machines, hence the interest of Entropy.

This first experiment will serve as a gauging on which we will base our scaling decisions for the elasticity evaluation. Our program retrieves configurations with random loads and try to compute a result within a given error margin by provisioning as many workers as needed.

Note that for all the experiments, Entropy's timeout is set to 1 minute.

5.3.3 Distributed Entropy

Figure 5.4 shows the results given by Entropy with different numbers of workers, for the same loads which vary from 10 to 50%.

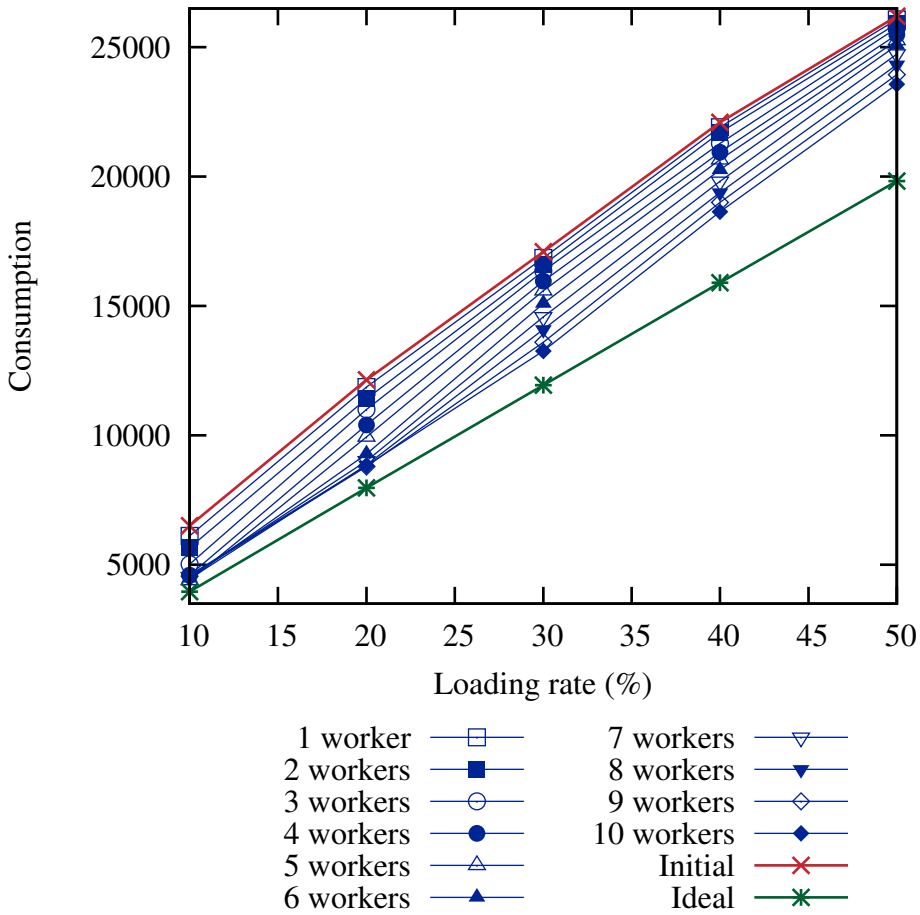

Figure 5.4: Comparing consumption results, for 200 PMs

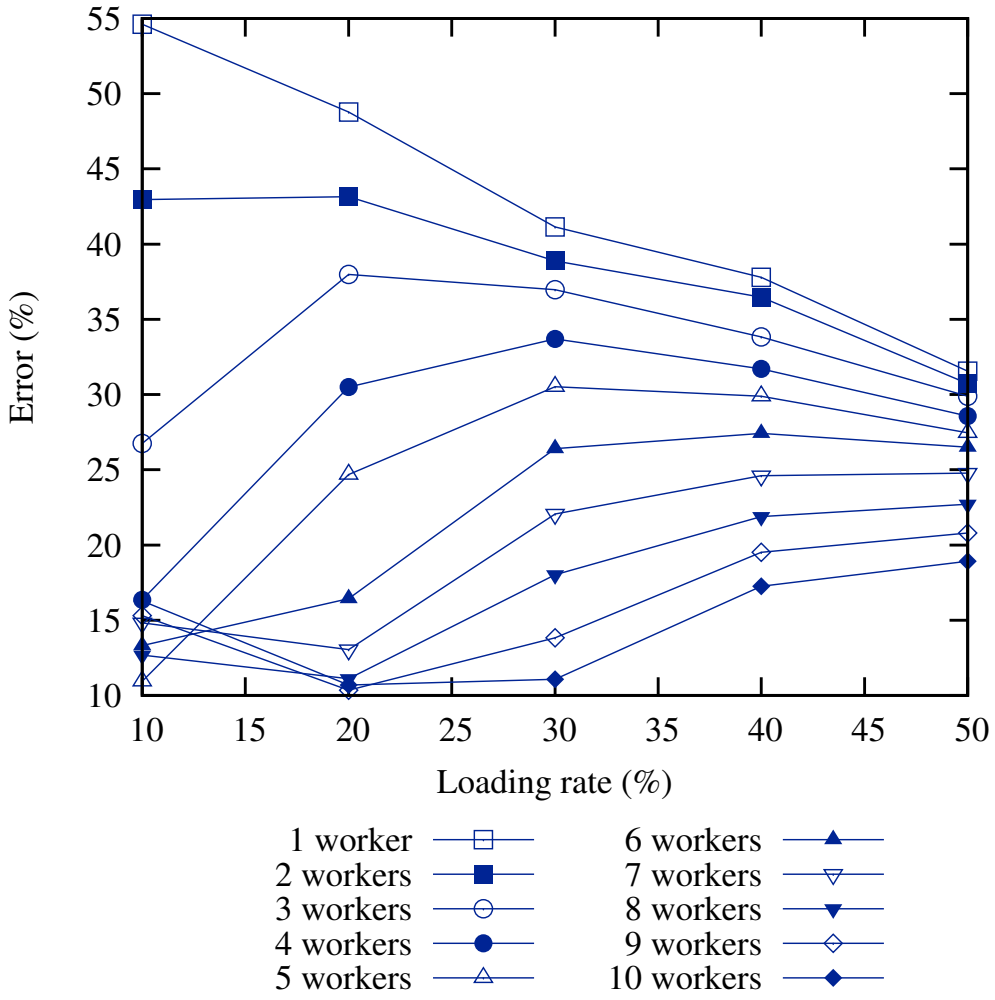

Figure 5.5: Comparing consumption results, for 200 PMs

We can see that for a cloud loaded at only 10%, we do not always improve the result by adding more workers which is justified by the limitation discussed in 5.2.2. On the other hand, for more important loads, we do achieve better optimization by using more workers.

Figure 5.5 depicts the error rates for this experiment. Based on these values, in order to have a result within an error margin of 20%, we will need the following minimum numbers of workers, depending on the load:

Loads	10%	20%	30%	40%	50%
Workers	4	6	8	9	10

5.3.4 Elastic Entropy

Now that we have gauged Entropy, we can use it to elastically adapt to any load in order to provide a result with an error of no more than 20%. Figure 5.6 shows a random load profile and the corresponding behavior of our elastic consolidation manager.

We can see that the number of workers adapts to the load in order to guarantee an error rate of less than 20%. Note that in order to have that same performance with a static provisioning, we will have to maintain 10 workers all the time although the maximum number of 10 workers is only needed once, thus the advantage of having an elastic solution.

5.4 Conclusion

In this chapter, we proposed an elastic consolidation service that adjusts its size to the dynamic needs of the cloud environment. Our proposition relies on virtualizing the consolidation service, which allows easily scaling this service by provisioning dedicated virtual machines to process the consolidation. Since these virtual machines persist for a certain period of time, the provisioning cost is amortized and can thus be neglected. Any of these virtual machines is in charge of consolidating a partition of the cloud environment, defined by a probabilistic node partitioning policy. By doing so, the exponential cost of consolidation becomes quasi-constant. Whereas our solution is quite simple, a performance evaluation shows that it provides good results.

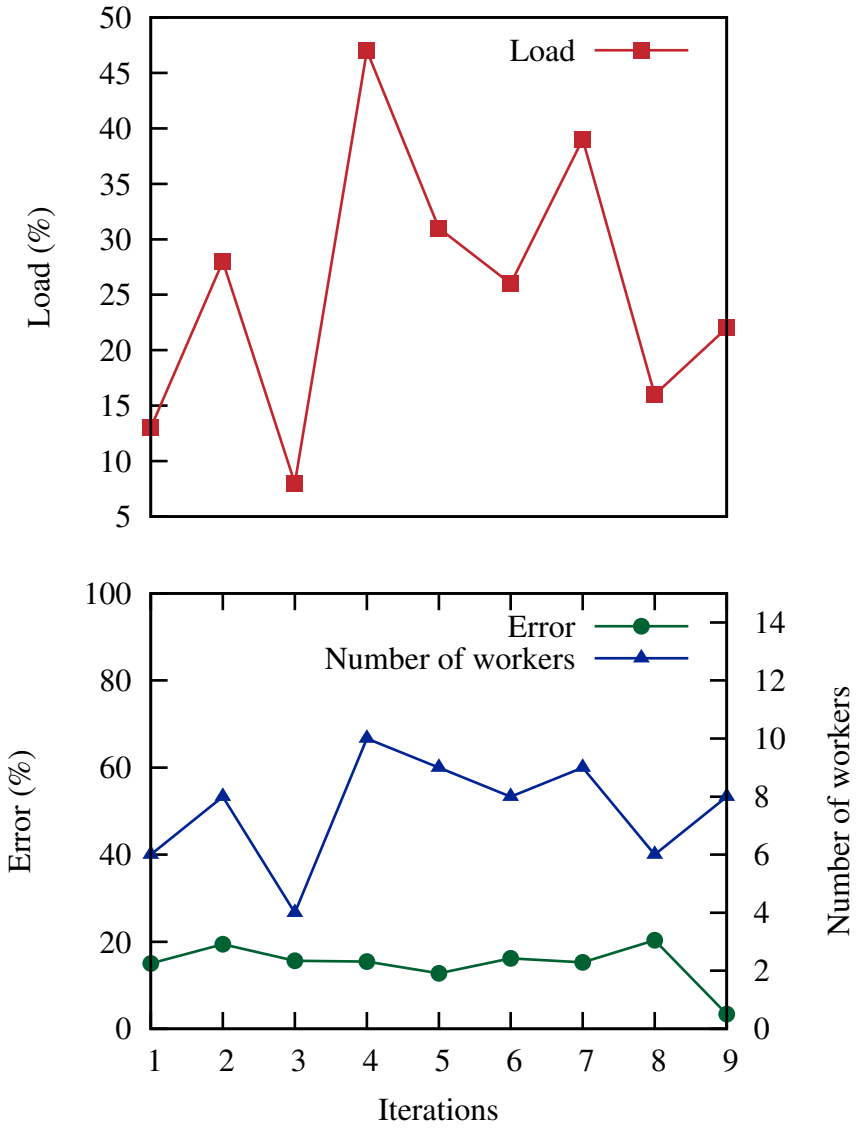

Figure 5.6: Elastic Entropy's error rates and number of workers

Chapter 6

Elastic Queues

Contents

6.1 Context

6.1.1 Message-oriented Middleware

Message-oriented Middleware (MOM) is one of the most commonly used ways to simply yet reliably integrate the different components of a distributed software system. MOMs use messages as the only structure to communicate, coordinate and synchronize, thus allowing the components to run asynchronously. They offer two communication paradigms: *one-to-one*, producers send messages to a queue where they are stored till they are consumed by one and only one consumer; and *one-to-many* or *publish-subscribe*, a producer sends a message to a topic that broadcasts it to all the subscribed consumers. MOMs have been widely adopted due to the guarantees they offer, namely:

- *Asynchrony:* the asynchronous property decouples producers from queues. They do not need to be both ready for execution at the same time. This property enables a deferred access to queues and a loose coupling between producers and consumers.
- *Reliability:* once a message is sent, it is guaranteed to be delivered despite network failures or system crashes.

Message-oriented Middleware has been standardized by different communities on different levels. The most recent one is Advanced Message Queuing Protocol (AMQP)[1], which is an application layer protocol as is Message Queue Telemetry Transport (MQTT)[2]. The latter is intended to be used by sensors and other objects that require low code footprint. As for the Java world, Java Message Service (JMS)[3] has been adopted very early as the standard messaging API.

6.1.2 Java Message Service

JMS is part of Oracle's Java Enterprise Edition platform and aims at allowing different Java applications to interconnect through a MOM. Its architecture is based on the following elements:

- *Provider:* this is the MOM which implements the JMS API and interconnects our Java applications; in our case it is Joram.
- *Client:* this can be any Java application or object that sends or receives messages, it is then respectively a *producer* or a *consumer*.

[1] AMQP, `http://www.amqp.org`
[2] MQTT, `http://mqtt.org`
[3] JMS, `http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html`

- *Message:* this is an object containing the data exchanged between JMS clients.
- *Queue:* it is the structure where sent messages are stored till they are consumed by a JMS consumer. A message is removed from the queue once it has been consumed.
- *Topic:* the topic is a special message destination that broadcasts the messages it is sent to all the subscribed JMS consumers, if a topic has no subscription, the messages it receives are lost.
- *Connection:* it encapsulates an open connection with the JMS provider, it typically represents an open TCP/IP socket between a client and the MOM.
- *Session:* this is finally a single-threaded context for producing and/or consuming messages.

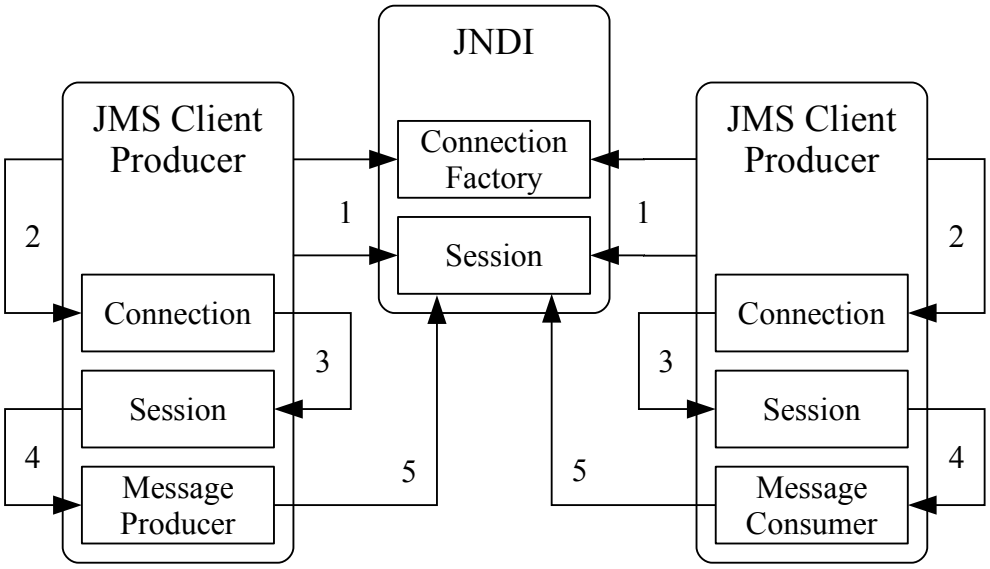

Figure 6.1: JMS architecture and workflow

Practically, as depicated by Figure 6.1, a JMS client would connect to a naming service, in this case a Java Naming and Directory Interface server, in order to retrieve the connection factory and the destination it wants to reach. It then uses

this connection factory to create a connection, and a session within this connection. Finally, based on whether it is a producer or a consumer, it creates the appropriate object and starts communicating with the destination (i.e. either a queue or a topic). Note that a client can create both producer and consumer objects and alternate between the two roles.

6.1.3 Joram

Joram[4] is an open-source pure Java implementation of the JMS standard, built on ScaleAgent D.T.'s distributed agents platform. A cluster of Joram servers corresponds to a set of agent containers, interconnected through reliable channels. Agents can be messaging destinations or client connections for instance. A user would then be able to connect to any of the cluster's servers and reliably connect to any topic or queue, should it be on a different server of the same cluster. Joram also implements other messaging standards such as the AMQP and MQTT protocols.

6.2 Scalablity Approach

In Message-Oriented Middleware, a queue is used to store the produced messages until a message consumer retrieves them. Since the consumers often process the messages they receive, they cannot always cope with the production speeds imposed by the message producers, and messages soon begin to pend on the message queue. In this section, we propose a scalability mechanism that allows producers to seemlessly send messages to a pool of queues along with their consumers and distributes the messages between them. We first detail the scalability mechanism, then study its scalability and finally present our flow control based load balancing policy.

6.2.1 Scalability mechanism

In order to achieve queues' scalability, we introduced the *alias queue*. An alias queue is a special queue on the producer's side, that automatically forwards the messages it is sent to another, generally distant, queue on the consumers' side, see Figure 6.2. It is set to write-only mode as the "real" destination, on which the messages are to be consumed, is the queue to whom the messages are forwarded. Thus, once a producer connects to our alias queue, we will be able to internally change the destination while maintaining the producer's connection to the same

[4]Joram, `http://joram.ow2.org`

queue. We can also add or remove destinations, i.e., queues, and notify the alias queue to take our modification into consideration. The alias queue mechanism does not only insure JMS compatibility, it also guarantees a total decoupling between the producer and the consumers as it completely isolates the producer from the consumption system: the producer will always be able to send messages to its alias queue without taking into consideration any changes in the consumption rate or availability of consumer queues. The system's reliability is also increased as the alias queue includes a fail-over mechanism and can resend a given message to another queue if its initial destination is unavailable.

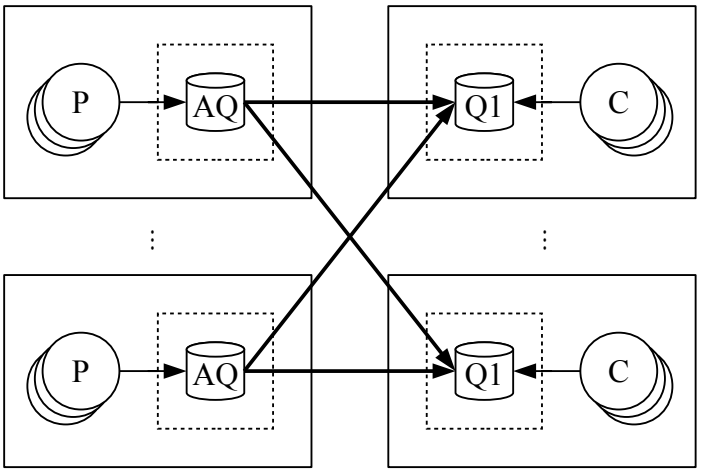

Figure 6.2: Alias queue principle

We will now compare the scalability of this load-balanced queues setup to that of a single queue.

6.2.2 Scalability Study

In this sub-section, we define the parameters that affect the performance of our system, first in the simple case of a single queue, before generalizing the results to the case of load-balanced queues.

Single Queue

Let p be the production rate on the queue and c the consumption rate. l being the length of the queue, i.e. the number of waiting messages, we have:

$$\Delta l = p - c$$

Depending on the result, three cases can be identified:

- $\Delta l > 0$: This means that the queue receives more messages than it is asked to deliver. The number of pending messages grows and we say that the queue is *unstable* and *flooded*. This will eventually cause the unavailability of the queue since it is allocated a finite memory.
- $\Delta l < 0$: In this case, the consumption rate is higher than the potential reception rate. The queue is still *unstable* and we say that it is *draining*. This means that the resources linked to this queue are not optimally utilized.
- $\Delta l = 0$: Here, the consumption rate matches the reception rate and the queue is *stable*. This is the ideal case that we aim to achieve.

The stability of a queue is thus defined by the equilibrium between the messages' production and consumption.

Load-Balanced Queues

In this case, the alias queues, to which the messages are sent, are wired to n queues, on which the messages are received. Let P be the total production rate on all the alias queues, c_i the consumption rates on each of the consumers' queues, and l_i their respective lengths. The scalability of our distributed system can be discussed on two different levels:

- *Global scalability*: Let L be the total number of waiting messages in all the consumers' queues. We have:

  $$L = \sum_{i=1}^{n} l_i$$

  and:

  $$\Delta L = P - \sum_{i=1}^{n} c_i$$

  The overall stability of our system is given by: $\Delta L = 0$. This shows that, globally, our system can handle the global production load. However, it does not guarantee that on each consumer queue, the forwarded load is properly handled. This will be guaranteed by *local scalability*.

- *Local scalability*: Depending on how we distribute the messages between the different queues, each would receive a ratio r_i of the total messages produced on the alias queues. Thus, for each $i \in \{1..n\}$ we have:

$$\Delta l_i = r_i.P - c_i$$

Local scalability is then given by:

$$\forall i \in \{1..n\}; \Delta l_i = 0$$

Note that local scalability implies global scalability as:

$$\forall i \in \{1..n\}; \Delta l_i = 0 \Rightarrow \Delta L = \Delta \sum_{i=1}^{n} l_i = \sum_{i=1}^{n} \Delta l_i = 0$$

This shows that, ideally, the forwarding rates (r_i) of each queue should adapt to its consumers' consumption rate (c_i). Note that we didn't discuss the alias queue's load as, if our system works properly, it shouldn't have any. As explained earlier, the alias queue automatically forwards all the messages it is received.

6.2.3 Flow Control Policy

Our load balancing policy is flow control based. It is a consumption-aware policy that aims at forwarding more messages to the queues with the highest consumption rates. Practically, a controller periodically retrieves the consumption rates on the different load-balanced queues, and computes the new forwarding rates as the ratio between a queue's consumption to the total consumption of our system during the last round. Since this is a reactive policy, a significant change in the consumption rates might result in the overload of some queues. Our policy tries to distribute the overload over all the queues by artificially subtracting the difference between the queue's load and the average load from the number of messages it has consumed: if the queue's load is greater than average, it is forwarded less messages than it can handle so as it can consume some of its pending messages, otherwise, it is forwarded more messages, which would increase its load, and we'd have eventually the same load on all of our queues. This reduces the latency of our system as it minimizes the maximum load per queue, thus reducing the amount of messages that should be consumed before the last pending message can be consumed. If we take up the parameters defined earlier, the forwarding rates based on the values monitored on round k can be expressed as follows, $\forall i \in \{1..n\}$:

$$r_i(k+1) = \frac{1}{C} max(c_i(k) - (l - l_{avg}), 0)$$

where:

$$C = \sum_{i=1}^{n} c_i(k) \; ; \;\; l_{avg} = \frac{1}{n} \sum_{i=1}^{n} l_i$$

This load balancing policy has two key assets: (i) it adapts to the changing consumption rates of the different load-balanced queues, and (ii) it distributes the overload over all the queues. Provided that our static system can handle the production load (global scalability), our flow control based policy guarantees, eventually, the local stability of each of the load-balanced queues. However, if the global load of our system increases beyond its maximum consumption capacity, all what our load balancing policy can do is cause the loads on the load-balanced queues to grow uniformly. Thus the need for a dynamic provisioning of resources to automatically cope with a global load change.

6.3 Elasticity Approach

In this section, we present the different elements of our elastic messaging solution. We first describe when does our elasticity algorithm provision new queues (scaling out) and remove unnecessary ones (scaling in), then we detail our provisioning approach.

6.3.1 Scaling decision

In order to guarantee the scalability of our messaging system with regard to a change in the global load, we have implemented an elasticity controller that periodically: (i) monitors the different loads on the different queues of our system, (ii) potentially adds or removes queues based on the monitored values. Note that we base our decision solely on the queues' loads, since the consumption capacity on each queue may vary, particularly in the dynamic context of a cloud infrastructure.

Scaling up

This is achieved when we monitor that the average load of the system's queues is beyond an acceptable limit $maxLoad$. This guarantees that our system's scalability is beyond the scope of the flow control mechanism as the latter can only bring the queues' loads to this average load, whereas what is needed is to reduce the average load itself, which can only be done by provisioning more resources.

Scaling down

This should be triggered when we see that the system is using too many resources than it actually needs. We can *suspect* our system to be underloaded when all

its queues are underloaded, i.e., almost empty, practically when the average load is below a threshold $minLoad$). This means that we have enough resources for the messages to be consumed as soon as they are produced, possibly *just enough* resources, in which case we shouldn't proceed with the removal of any of the queues. Thus, the scaling down decision can not be made at once.

To make sure that our system is effectively underloaded, when the elasticity controller suspects the system's overload, it elects a queue to be removed, and starts decreasing the amount of messages it is forwarded gradually. If, doing so, the average load goes above the specified limit, the scaling down plan is canceled and the elected queue receives messages normally, as specified by our flow control policy. Otherwise, if the elected queue is no longer forwarded any messages without the average load exceeding $minLoad$, then we can safely assume that this queue is no longer needed and it is effectively removed from our system.

Figure 6.3 outlines our elasticity algorithm.

Now that we have presented when scaling should be done, the next section details how it is actually achieved.

6.3.2 Provisioning

One a scaling decision is made, fast execution is of great importance to the proper working of our solution. Or, provisioning a virtual machine instance (VM) is relatively slow, for instance, it takes about a minute to provision a small Ubuntu instance on Amazon EC2 [22]. To deal with this, our solution relies on co-provisioning queues on a same instance and pre-provisioning a pool of VMs.

Co-Provisioning

Since we are using a cloud computing infrastructure, where the resource unit is a virtual machine instance, an intuitive approach would be to add each queue on a separate VM instance. However, Joram's evaluation shows that due to the internal functioning of Joram, and depending on the size of the VMs, two or more queues can coexist on the same VM instance and still have comparable performance as with a configuration where each runs on a separate VM instance. Figure 6.4, shows the maximum throughput that can be achieved on a small EC2 VM on queues in different setups with regard to persistency of the messages, connection type and co-locality, with a message size of 100B. We can see that in a persistent setup, which is the most reliable, we can fairly co-provision up to 2 queues without significant performance decrease.

Thus, the resource unit is no longer the VM instance, but the available *slots* that we can provision queues on. Co-provisioning allows us to diversify our provisioning policies: if our main concern is performance, we might want to have

```
while(TRUE) {
    sleep(period);
    monitorQueues();

    /* Scaling down */
    if (avgLoad > minLoad) {
        // Cancel scaling down plan
        toRemove = NULL;
    }

    if (avgLoad < minLoad && !toRemove) {
        // Start a new scaling down plan
        toRemove =
            queues.electQueueToRemove();
    }

    if (toRemove) {
        // Continue scaling down plan
        toRemove.reduceRate()
        if (toRemove.rate == 0) {
            queues.remove(toRemove);
            toRemove = NULL;
        }
    }

    /* Scaling down */
    if (avgLoad > maxLoad) {
        queues.addNewQueue();
    }

    /* General case */
    queues.applyFlowControlPolicy();
}
```

Figure 6.3: Elasticity algorithm outlines

	Persistent		Transient	
Setup	localCF	tcpCF	localCF	tcpCF
1 queue	25 309	13 862	41 726	19 669
2 queues	25 052	13 456	33 971	16 828
4 queues	22 318	13 338	25 741	28 450

Figure 6.4: Queues' throughtput with different setups

each queue on a new VM instance, and provided we are using a private cloud, we might even want to create this VM instance on the least loaded physical machine. Other policies might have energy efficiency as the main concern. This is the case for the basic policy that we have implemented.

However, co-provisioning only reduces the impact of VM provisioning lag, for once all the slots on an instance are filled we still have to provision a new VM. Pre-provisioning deals with this issue once and for all.

Pre-provisioning

In order to optimize our solution even more, we have looked into reducing the time needed to add new queues, particularly when it involves provisioning a new virtual machine instance. The solution we propose is pre-provisioning a certain number of unneeded VM instances, which will be maintained as long as our cloud messaging system runs. This means that when a new node is needed, we use a pre-existing node, which renders our system more reactive, the used VM is then asynchronously replaced, which means that the creation of the new VM instance will not affect the latency of our system, thus improving its performance.

In order to evaluate the number of necessary pre-provisioned VMs, we need the Service Level Agreement to specify not only the maximum tolerated latency, which defines our $maxLoad$, but also the maximum supported increase of the production rate during a unit of time. Considering the following parameters:

- $SLA.delta$: The maximum increase of the production's rate in 1s (msg/s^2).
- $VM.startup$: The average startup time of virtual machines (s).
- $VM.capacity$: The maximum consumption capacity of a virtual machine, provided all its slots are filled (msg/s).

The number of virtual machines to be pre-provisioned NPP is given by:

$$NPP = ceil(\frac{SLA.delta \times VM.startup}{VM.capacity})$$

The numerator expresses, in the worst case, the extra production load that might occur during the startup of a virtual machine. This should be handled by our pool of pre-provisioned VMs, thus, it should be equal to $NPP \times VM.capacity$, hence the formula above.

Next, we present the implemented provisioning policy.

Provisioning policy

Our provisioning policy is energy-efficiency-driven and aims at having an automatically consolidated park of queues. Should we have control over the cloud infrastructure as well, this consolidation is achieved on both levels: (i) having our virtual machines on the minimum possible number of physical machines (PM) and (ii) having the provisioned queues on the minimum number of VMs.

This is achieved on scaling up, by always trying to provision the new queue on an available slot in an existing VM instance, and only create a new instance if all the available slots on the last created VM instance are filled; and when creating the new VM instance always try to use the current physical machine and only use another if the first cannot host the new instance. This automatically minimizes both the numbers of utilized PMs an VM instances. On scaling down, in order to maintain the automatic consolidation, we always remove the last added queue, if it was the last one on its VM instance, than we can destroy one pre-provisioned VM and if this VM was the last one on its corresponding PM, the latter can be put into an energy-saving mode.

The next section studies the performance of our elastic cloud messaging system and shows the specific improvement due to each optimization.

6.4 Evaluation

In this section, we validate our elastic messaging system and discuss its performance. We will not only validate our implementation as a whole, but also highlight the performance gain provided by each optimization, i.e., co-provisioning and pre-provisioning. All the following experiments have been done on Amazon EC2, using *m1.small* instances.

6.4.1 Effect of co-provisioning

In these two first experiments, the system is subjected to a production rate that gradually goes up to $750msg/s$, a single worker is configured to consume at most $100msg/s$, our elasticity algorithm's $minLoad$ and $maxLoad$ are respectively $50msg/s$ and $200msg/s$.

Figure 6.5 shows the results of allowing at most one worker per VM, whereas Figure 6.6 shows the results with provisioning up to two workers on the same VM. In both cases, no VM has been pre-provisioned.

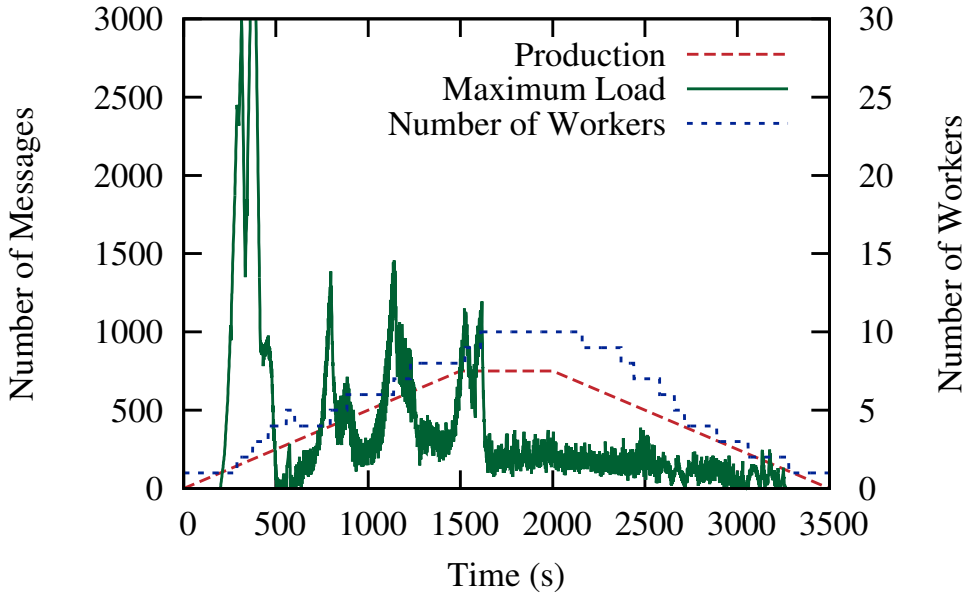

Figure 6.5: 1 worker per VM, no provisioning

As expected, the latency of VM provisioning results in overload pikes that might require the provisioning of extra workers to be handled, even though our algorithm has a safety interval in which he awaits the scaling decision to take effect. We can see comparing both Figures 6.5 and 6.6 that the overload pikes have been halved, as half the times in the second experiment, a worker doesn't have to wait for the provisioning of a new VM but can directly be provisioned on an existing VM. It is as well worth mentioning that in the second case, we only use half the number of virtual machines, which is a significant improvement in terms of energy efficiency.

6.4.2 Effect of pre-provisioning

Using the same parameters as above, we made a third experiment, where, in addition to provisioning two workers on the same VM, we pre-provision a VM. The results are depicted by Figure 6.7.

The pre-provisioned VM completely removed the impact of VM startup latency on our system, as we no longer need to wait for a VM to start: we always have an available VM to use and we replace it asynchronously.

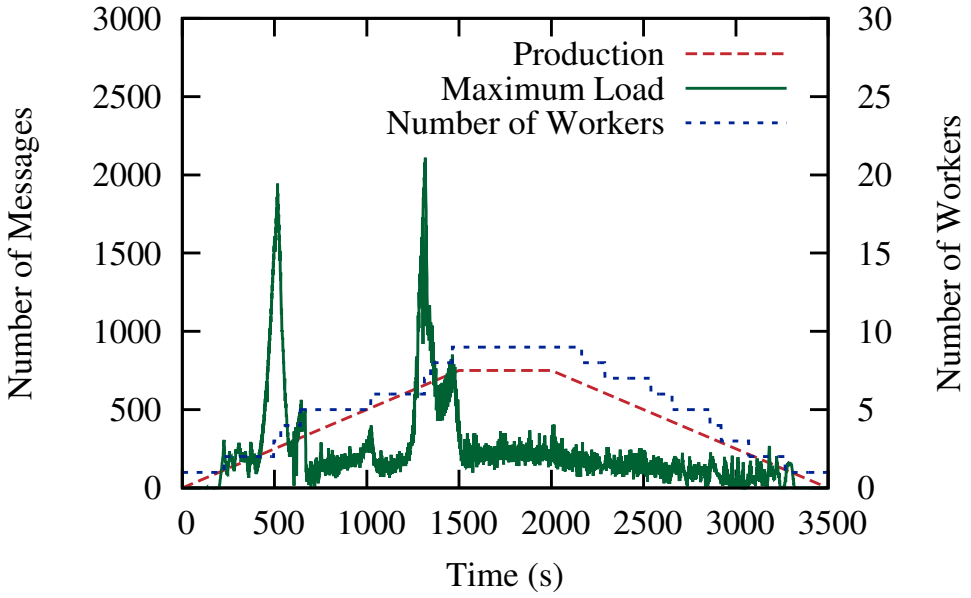

Figure 6.6: 2 workers per VM, no provisioning

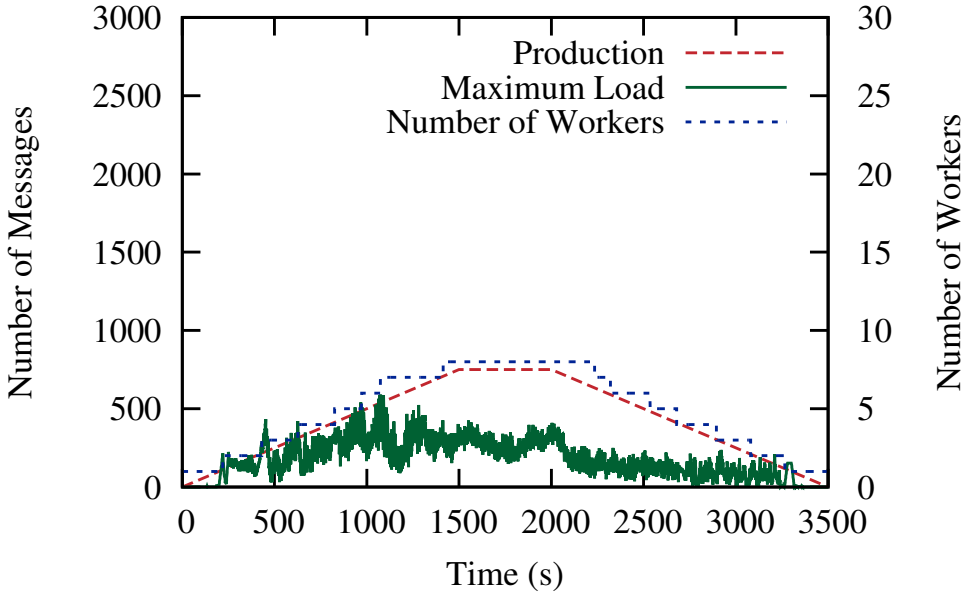

Figure 6.7: 2 workers per VM, 1 pre-provisioned VM

6.4.3 Size of the pre-provisioning pool

In the previous experiment, one VM was enough for our system to work properly, as the production rate's acceleration was not very high. In the following two experiments, we multiply this acceleration by eight. Figures 6.8 and 6.9 show the results with respectively one and two pre-provisioned VMs.

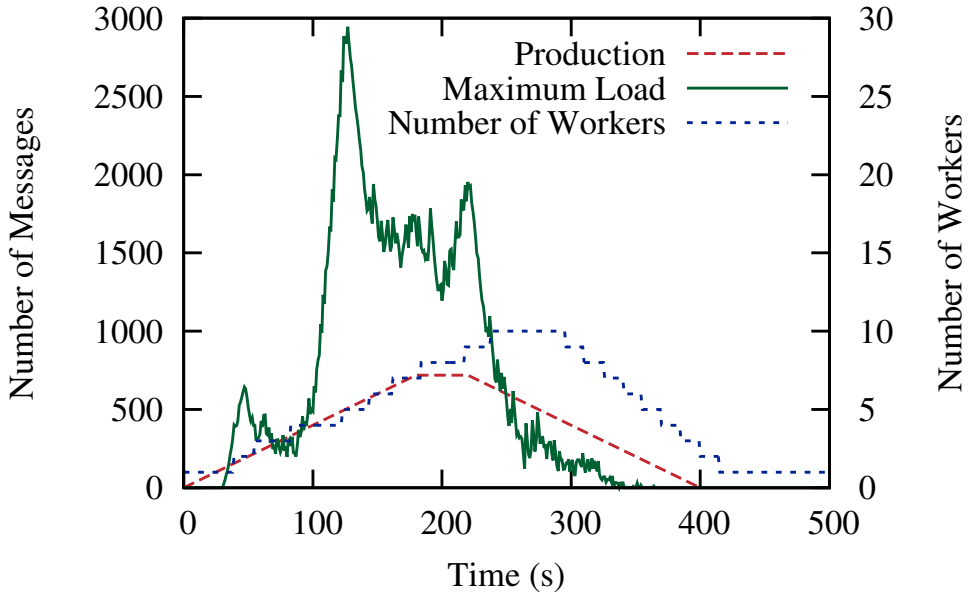

Figure 6.8: 2 workers per VM, 1 pre-provisioned VM

It is clear from Figure 6.8 that, in this case, provisioning one VM isn't enough, the provisioning of the four first workers happens normally, since they use the first VM (the first two), and the pre-provisioned VM (the third and fourth), however, when a fifth worker is needed, it has to wait for the new pre-provisioned VM to start up, as it didn't have enough time to launch.

This can be expected as, if we take up the formula expressed in 6.3.2, the production rate increases by $100msg/s$ each $25s$, which corresponds to an $SLA.delta$ of $4msg/s^2$, and given that the mean startup time of an EC2 linux instance is $VM.startup = 96.9s$ [22], and that each VM contains 2 workers which consume $100msg/s$ each, which means that $VM.capacity = 200msg/s$, the minimum number of pre-provisioned VMs should be:

$$N = ceil(\frac{4 \times 96.9}{200}) = 2$$

Sure enough, pre-provisioning two VMs results in a proper functioning of our system as shown by Figure 6.9.

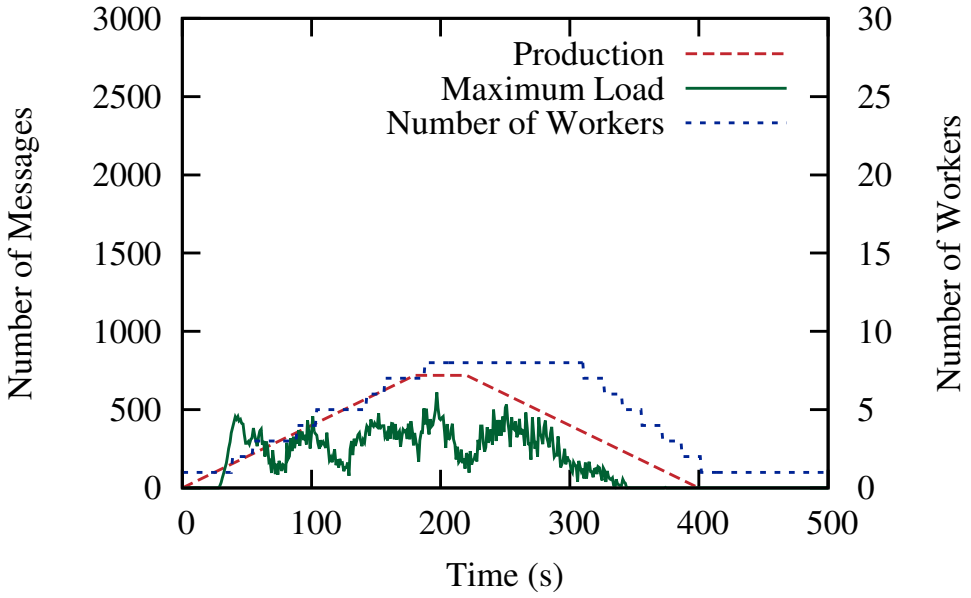

Figure 6.9: 2 workers per VM, 2 pre-provisioned VMs

6.5 Conclusion

We have presented an elastic message queuing system that adapts the number of queues and consumers to the load of messages. Our system has 3 main assets, (i) its flow control based load balancing makes sure that the provisioned resources are used to their maximum capacity; (ii) in the case of overload, our pre-provisioning and co-provisioning techniques achieve high reactivity while minimizing the cost and (iii) removal of unnecessary resources is done gradually in order to minimize the number of wrong decisions which would affect badly the performance of our system. Our work has been evaluated on a public cloud and particular care has been taken to show the benefit of each of our provisioning techniques. In the future, we intend to study the impact of different provisioning strategies on the behavior of our messaging system and generalize our approach to the one-to-many messaging paradigm.

Chapter 7

Elastic Topics

Contents

7.1 Context

In this section, having already presented the general context of messaging middleware as well as Joram, the messaging solution on which our work is based, we will only present some specificities of the publish-subscribe paradigm.

The pure publish-subscribe paradigm, implemented by Joram, assumes that if a publication doesn't find any subscriber, it is simply lost. This is not the case of other brokers where each topic has an underlying queue that stores publications until they are consumed by at least one subscriber. Thus, while consumption from a queue can be done in a transient manner, with a single HTTP request as it is the case for cloud messaging providers such as Amazon SQS or IronMQ,

subscription to a pure topic needs a continuous connection, for the subscribers not to miss any of the up-coming publications. Moreover, each of these subscribers needs to receive a publication as soon as it is available. This makes the number of subscribers to be handled an important scalability concern for topics' providers. Naturally, the load of publications is also a concern, but we believe this can be addressed by a fairly simple adaptation of the solution proposed in the previous chapter.

In this work, we focus on scaling topics with regard to the number of subscribers. We propose a tree-based architecture to limit the number of connections per topic depending on a given tolerated maximum latency or throughput. We assume that the system is able to handle the production load to which it is subjected.

7.2 Approach

This section describes how we managed to make topics automatically adapt to the varying load of subscribers' connections. We first present the criterion on which we base our decision, then our tree-based architecture and how it adapts to the former's variation.

7.2.1 Topic Capacity

When a topic receives a publication, it has to go through all its registered subscribers in order to forward it to them. Obviously, The more subscribers a topic has, the longer it takes for it to forward a publication and the more the last subscriber has to wait before receiving it. Thus, for a given throughput, there is a maximum latency, within the messaging system, that should not be exceeded, if we want all the subscribers, particularly the last one, to be up to date with regard to the pace of publications.

We define for each topic a capacity $C(T_{max})$, function of the maximum throughput of publications it can handle, which represents the number of connections this topic can serve, without exceeding the latency corresponding to T_{max}. Roughly:

$$L_{max} = \frac{1}{T_{max}}$$

As, in the scope of this work T_{max} remains constant, the lever of our elasticity would be the number of subscribers, the more the subscribers the more topic nodes are needed to process them smoothly. To achieve this, we propose an adaptive tree-based architecture.

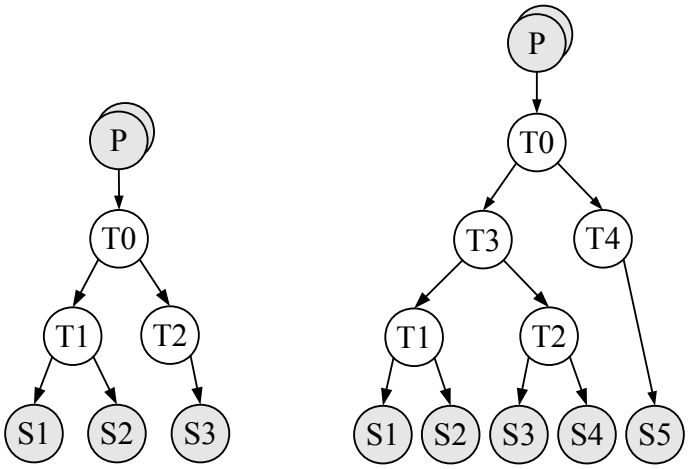

Figure 7.1: Topic trees with different subscribers

7.2.2 Tree-based Architecture

The main idea behind our tree-based architecture is to keep the number of connections to each topic node under its defined capacity. Instead of having the number of subscribers' connections grow linearly on a single topic. We define a front-end topic that receives the publications and forwards them to as many underlying topics as necessary to handle the number of subscribers.

However, doing so would eventually lead the number of underlying topics to exceed the capacity of the front-end topic. Thus the need of a multiple level tree-based architecture. This means that we will have two different types of topic nodes, even though they are functionally the same: intermediate nodes which forward publications to other topic nodes, and final nodes, which represent the leaves of our tree and forward the publications they receive to the subscribers. Figure 7.1 shows setup examples of our topic nodes, with a capacity of 2 connections per topic and different numbers of subscribers.

Naturally, at each additional level of topics, an extra hop is added to the latency of publication delivery. This overhead is however minimized by the fact that our architecture is meant to be set on the provider's side, in a local network. Moreover, the evaluation presented in 7.3 shows that provided a large enough of subscribers, we achieve a better latency using our solution.

Now that we have a scalable topic architecture, the next subsection shows how we adapt it to the variations of the number of subscribers.

7.2.3 Scaling Decision

As it is the case in our previous contribution, elasticity is achieved by periodically retrieving the state of the system, and take actions based on the guarantees we would like to preserve.

In this case, the single rule that has to be respected is that the number of subscribers per topic node should not exceed its capacity. As we retrieve the state of our system, we consider two particular values: N_S, the total number of subscribers, and N_{FT} the number of final topics, i.e., the leaves of our topic tree. C being the capacity of a topic node, we have the following cases:

- $N_S > N_{FT} \times C$: An extra topic node should be added.
- $N_S < (N_{FT} - 1) \times C$ and $N_{FT} > 1$: We can remove one topic node as it is no longer needed to handle the current number of subscribers.
- $N_{FT} - 1 \times C < N_S < N_{FT} \times C$: No scaling decision is needed.

Once the number of topic nodes is adapted, the subscribers are redistributed uniformly over the final topic nodes. This fills the empty slots left by unsubscribed subscriber and/or new ones due to the addition of resources. The next subsection shows how this is practically carried out.

7.2.4 Implementation Details

This section discusses some aspects of our elastic topic implementation.

Subscribers' redirection

This first aspect is key to our solution as it allows us, upon subscription, to redirect a subscriber to one of the final topic nodes -on a round-robin basis-, thus relieving the load on an otherwise over-capacity topic.

This is achieved by wrapping the subscriber's client. All what the user has to do is create a subscriber to given topic, which would correspond to our front-end node, and start seamlessly receiving publications. What happens beneath is that our subscriber's wrapper would have received, upon its connection to the front-end node, a special message with information about the new topic node to connect to. Special messages are naturally not reported to the users, i.e., they are not subjected to their defined callback method.

Redirection also occurs when our system tries to re-balance the subscribers. In this case, our subscriber's wrapper creates a new connection to the new node, and keeps the first one open till it receives the first publication from the new topic node. This guarantees that (i) no publication is lost during the re-connection process and (ii) that each publication is processed at most once by the user's callback method.

Subscribers' re-balancing

As stated earlier, the fact that subscribers can end their subscription forces us to re-balance the subscribers over the topic nodes.

In order to minimize the redirection impact during re-balancing, we do so by sending re-balancing orders only to the over-capacity topic nodes. These orders contain a list of topic nodes, which are obviously under-capacity, along with how many subscribers to redirect to each one of them. These values are computed based on the difference between the number of subscribers on each topic node and the defined capacity.

7.3 Evaluation

In order to validate our implementation, we have conducted two experiments. The first experiment shows that there is indeed a gain in using an extra level of topics, in spite of the overhead introduced by the new hop between topics. In the second experiment, we vary the number of subscriber and show how our elasticity mechanism keeps latency within an accepted range.

These experiments have been carried out on Amazon's public cloud, Elastic Cloud Compute, using exclusively instance of type *m1.small*[1].

7.3.1 Scalability validation

In this first experiments, we set different configurations by varying both the topics' layouts and the number of subscribers. In the first setup all subscribers classically connect to a single topic, on which the messages are produced. Next, we add different numbers of children topics, to whom the messages produced on the root topic are forwarded. Each topic runs on a separate VM instances, and the subscribers are evenly distributed across 6 VM instances. The performance metric we are interested in is the maximum latency, i.e., how much time does it take for the last subscriber to receive the message. The results are given by Figure 7.2.

We can see that, provided a big enough number of subscribers, the overhead of the extra hop between the root topic and the children is overcome by the gain due to a lesser number of subscribers per topics. Naturally, using a single child topic always performs worse than directly subscribing to the root as it introduced the overhead without any counterpart. It is also worth noting that using more children always reduces the maximum latency. However, it comes at the cost of provisioning extra resources.

[1] EC2 instances, `http://aws.amazon.com/ec2/previous-generation`

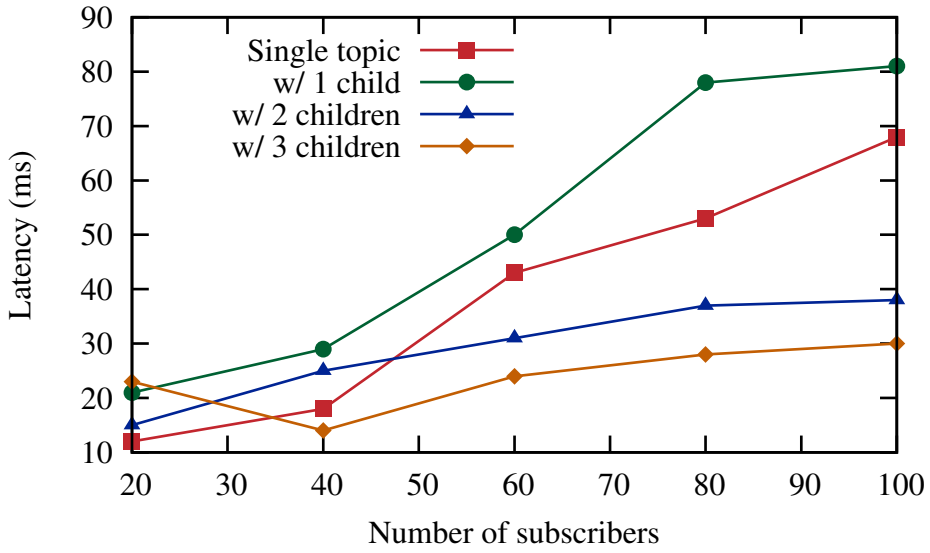

Figure 7.2: Topics' scalability

7.3.2 Elasticity validation

This experiment shows how our elasticity controller adapts to the variation of the number of subscribers in order to maintain a reasonable maximum latency. Initially, our topics' configuration consists in a root topic along with one sub-topic, later on extra sub-topics are added as the number of subscribers grows. As in the first experiment, the subscribers are created in a pool of 6 virtual machine instances. For the sake of this experiment, the maximum number of subscribers per topic is set to 50.

As shown by figure 7.3, at first, the latency increases as the number of subscribers grows. However, the addition extra sub-topics and the balancing of the subscribers over the existing sub-topics stops the linear increase of latency. Naturally, as the number of subscribers goes down, the no longer needed topics are deleted. The latency graph also shows punctual latency pikes occurring upon the addition of removal of new sub-topics. This is due to the balancing of subscribers: When adding the second sub-topic for instance, half of the first sub-topics' subscribers have to be redirected, and messages sent during this process are delayed. Likewise, a sub-topic cannot be removed prior to the redirection of all its subscribers. This overhead is however to be minimized as it only occurs when a relatively important number of subscribers have to be redirected at once.

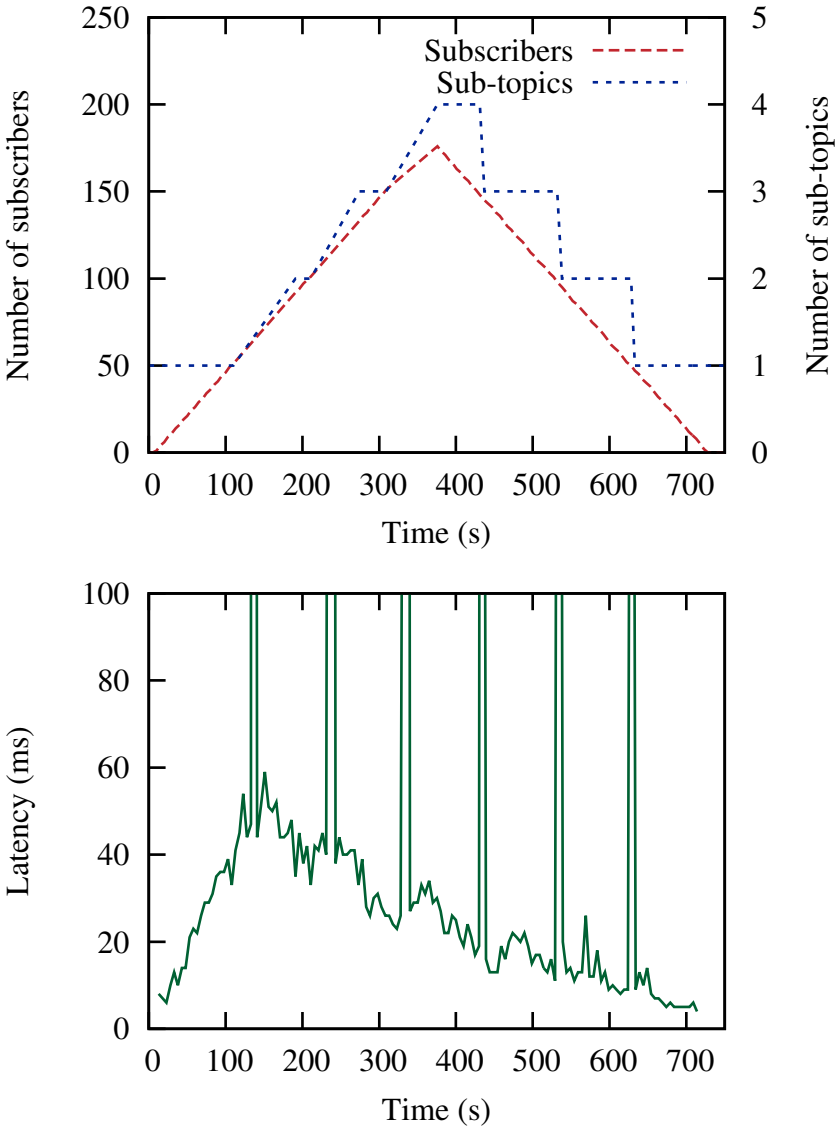

Figure 7.3: Topics' elasticity

7.4 Conclusion

In this chapter we have proposed a mechanism that automatically adapts messaging topics to the varying load of their subscribers. Our solution relies on limiting the number of subscribers per topic, and redirecting the subscribers to new topics in order to respect this limit. The evaluation carried out on Amazon's public cloud validates our solution and highlights the performance gain it provides.

Chapter 8

Elastic Stream Processing

Contents

8.1 Context

8.1.1 Stream Processing

With the proliferation of connected devices, data is generated at an ever-growing rate. This *big data* can be handled in two different ways. The first one to emerge is *batch processing* where data is stored in huge databases to be later processed, as

a whole, usually on distributed computing infrastructures and using scalable programming models such as Google's MapReduce. However, data is most valuable right after it is created, in Finance for instance, one can not afford to wait much long before making a trading decision, the market fluctuates just too fast and its analysis has to be reactive enough to follow. In other contexts, data is just too big to be stored. This is the case for instance of the data generated by the CERN's Large Hadron Collider whose observations must be at least filtered in real-time. Hence the need for the second big data paradigm: stream processing.

Stream processing platforms analyze data as it arrives in order to make the most out of it. They put the emphasis on reactivity even if it is achieved on the expense of precision. This is often not a problem as in the case of analyzing trending topics on social media where the exact number of occurrences is not of prime importance. The recent years have witnessed the emergence of many stream processing tools such as Yahoo's S4, which is based on the actors' model; Spark Streaming, which mimics its batch processing older brother Spark by gathering streams into small time-windowed batches. However, the most widely adopted stream processing platform remains Apache Storm, which has first been developed internally at Twitter.

8.1.2 Storm

Storm is a fault-tolerant distributed event processing framework. It allows the definition and deployment of component-based processing chains called *topologies*. A topology is composed of two types of components: *spouts*, which retrieve data from a given source and format it as a stream of *tuples*, i.e., a list of named objects; and *bolts* which receive one or more streams as inputs, define how each tuple should be processed and emit a new stream of tuples. Storm also allows the user to specify the parallelism of a component, i.e., how many instances of this same component would be deployed. Doing so, the user should specify the load balancing strategy to apply as well. Storm's load balancing ranges from a random policy, to sticky policies that group tuples by one or more fields. Figure 8.1 shows an example of a topology.

Architecture-wise, Storm has a central node called *nimbus* to which the topologies to be deployed are submitted. Nimbus takes then care of deploying the topologies onto the Storm cluster's nodes. Nimbus also receives some monitoring metrics from the *supervisors*, i.e., monitoring and configuration daemons on each of the Storm nodes. For its distributed configuration, Storm relies on Zookeeper. As Storm is a fault-tolerant platform, the failure of one node doesn't result in the whole system's failure. Particularly, even if the nimbus node fails, the rest of the Storm cluster would continue to function properly. Storm also discovers any new nodes joining the cluster or the departure of others and *rebalances* the run-

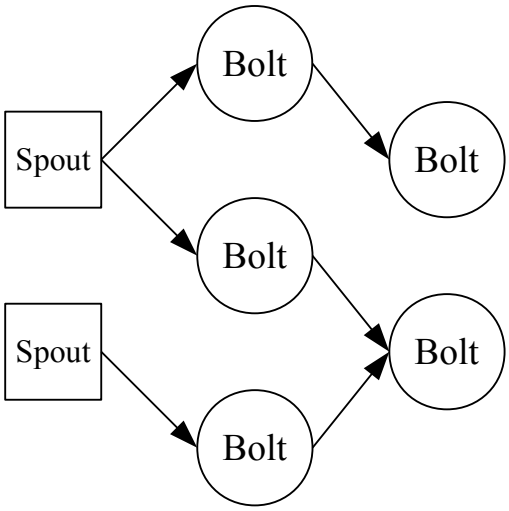

Figure 8.1: Storm topology example

ning topologies accordingly, although it does so simplistically in a round-robin fashion.

While Storm allows scaling either by setting the parallelism of a component or by changing the size of the cluster on the fly in order to adapt to load variation, it falls short of doing so automatically. In this work, we propose a solution that periodically monitors the state of a Storm topology and scales in or out one or more of its components should the need to do so arise.

8.2 Approach

In order to make Storm elastic, i.e., able to scale dynamically, we had to address three concerns: (i) how to monitor the platform and retrieve its current state, (ii) when should scaling operations occur, and (iii) how to achieve the scaling, i.e., leverage the provisioned resources. This subsection shows our approach to each of these concerns.

8.2.1 Monitoring

The first step prior to any scaling decision is to monitor the topology and select the metrics that we consider relevant to characterize the load of a given component. A Storm cluster can be monitored in a variety of ways. As for any cluster, system-level metrics can be fetched using a tool such as Ganglia. Moreover, Storm also

exposes application-metrics, such as the number of emitted or executed tuples. We consider that these Storm metrics are the most fit to reflect the load of a Storm component.

Basically, the state of a component is described by the difference between the tuples it receives, i.e., the tuples that are emitted by all its input components, and the tuples it is able to process within a given period of time. While this is straight forward to compute, it takes into account virtually any type of bottlenecks a component might encounter: should the processing of a tuple be CPU-intensive or should it rely on an external web service or database connection, it would invariably tell if a given component is able to keep up with the pace of its in-coming stream of data.

8.2.2 Scaling Decision

Once we have the Storm metrics of the components and provided that load balancing among different instances of the same component is guaranteed, through a random policy for instance, the scaling decision goes as follows:

- **Scaling out:** which is the addition of an extra component instance, should occur when a given component receives more messages than it is able to process, i.e., if 8.1 is verified, n being the current number of the component's instances.

$$\Sigma_{inputs} emitted > \Sigma_{i<n} executed_i \tag{8.1}$$

  Before scaling out, we store the last witnessed capacity of the component's instances as described by 8.2. This will be later used to initiate scaling in.

$$capacity = \frac{\Sigma_{i<n} executed_i}{n} \tag{8.2}$$

- **Scaling in:** which is the removal of an unnecessary component instance, should take place if we can guarantee that, with one instance less, the component's instances would still not be overloaded. Using the previously stored capacity, that is updated upon each scaling out, scaling in is instigated upon 8.3 verification.

$$\Sigma_{inputs} emitted > (n-1) capacity \tag{8.3}$$

Obviously, changing the number of component instances wouldn't have the desired effect on performance unless it is coupled with a correspondent change in the amount of provisioned resources.

8.2.3 Provisioning

Elasticity is possible thanks to the flexibility of cloud computing infrastructures, which allow on-demand addition and removal of computing resources. Thus, the resource unit to be managed is a virtual machine instance.

As stated earlier, Storm deploys a topology by dispatching its components evenly on available nodes, which would result in our components' instances affecting each others' performance and bias our scaling decision. It is of course necessary to mutualize resources in the case of a cluster of powerful physical machines, but cloud computing provides us with the choice of the amount of resources to provision not only in terms of total power, but also in granularity as cloud providers usually have a wide range of virtual machine instances' sizes. In this work, we propose to put each component's instance in a separate small virtual machine instance, which isolates the components without wasting computational power. The scaling and provisioning processes are thus transparent as adding a component instance provides it automatically with the extra resource to run on thus improving the performance of the component. Likewise, removing a component instance automatically saves the cost of the virtual machine instance it used to run on.

Of course, a multitude of policies can be imagined, and easily integrated to our solution, favoring either performance or cost-efficiency. But the proposed provisioning policy, despite its simplicity, proved to be good enough as will be shown in Section 8.3. Now that we have presented the key principles of our solution, this section presents its global architecture and details some of its implementation aspects.

8.2.4 Architecture

Our elastic Storm solution consists of two main parts:

- **Elastic Controller:** It is the external agent that periodically fetches the Storm cluster's metrics and decides whether or not scaling should be done. If scaling is needed, the controller requests the addition of a virtual machine instance, then it changes the parallelism of the component to be scaled. The deployment of the new component on the new virtual machine is then handled by our custom scheduler, on nimbus' side.

- **Custom Scheduler:** Storm allows users to plug a custom scheduler in order to customize their deployment strategies. We took advantage of that and developed a Storm scheduler that makes sure that each component's instance is granted one and only one virtual machine instance. This can be used to implement any other provisioning policy.

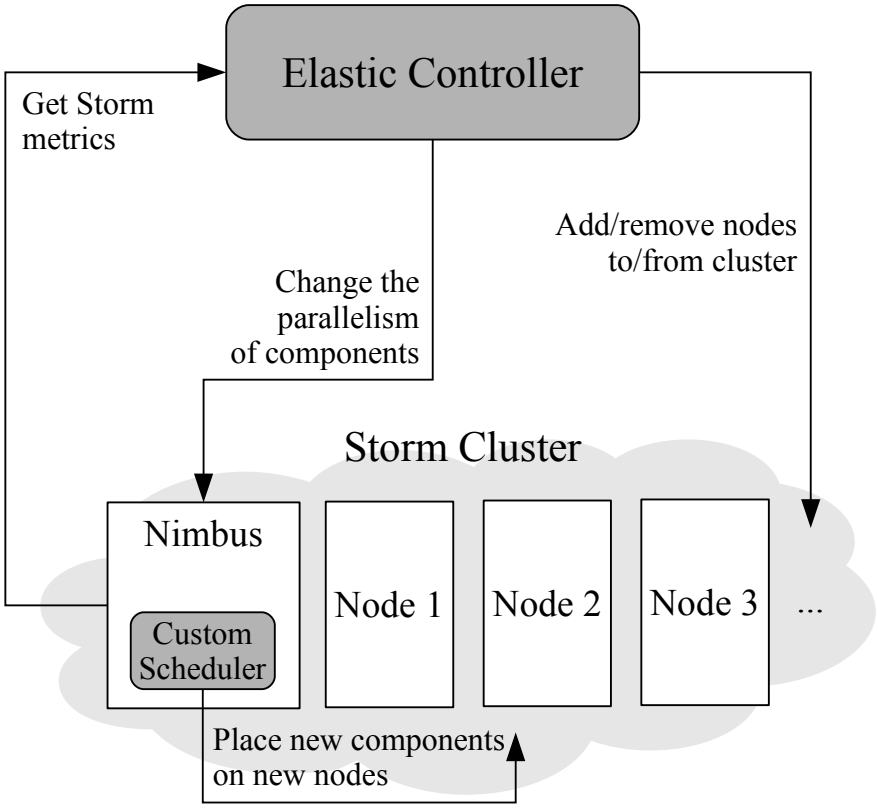

Figure 8.2: Elastic Storm architecture

Figure 8.2 shows the global architecture of our solution, and how it integrates with a regular Storm cluster.

8.2.5 Implementation Details

We will now discuss some technical points about how we made Apache Storm elastic.

Monitoring

There are different ways of monitoring a Storm cluster. Storm provides a UI showing the metrics of its topologies, mainly number of executed tuples per component, i.e., spout or bolt, and the average execution latency in different time windows. This is how our elastic controller computes the capacity of each bolt on which it bases its scaling decision. Our solution also uses Ganglia to retrieve

system metrics, even though they are not currently involved in the scaling process. Finally, for the sake of completeness, JMX API can as well be used to monitor Storm as it runs on a Java Virtual Machine.

Scaling

So far, we have discussed the parallelism of components as the number of instances of a component. Storm does actually have two levels of parallelism for its spouts and bolts:

- **Tasks:** which correspond to the instances of a bolt or a spout running concurrently.
- **Executors:** these are the threads on which a component's tasks will run.

While Storm allows the number of executors to change on the fly, the number of tasks per component is to be set once and for all throughout the lifetime of a topology. In our solution, the number of tasks per each component is thus set to a large enough number, and we scale the number of executors. Tasks are then deployed evenly on the existing executors.

8.3 Evaluation

In this section, we use the DEBS Grand Challenge data-set to evaluate our work. We first present our evaluation context before detailing the results of our different experiments.

8.3.1 Context

Infrastructure

Our experiments are carried out on a VMware vSphere run private cloud, using virtual machine instances with 1Mhz CPU, 2GB memory and 10GB disks.

Data-set

The data-set we use for our experimentation comes from the DEBS Grand Challenge 2014. It represents values reported by various home consumption sensors in the format: *id, timestamp, value, property, plug_id, household_id, house_id*, respectively identifying the measurement, its value, whether it corresponds to load or work, the plug the reporting sensor is plugged to, its household and its house identifiers.

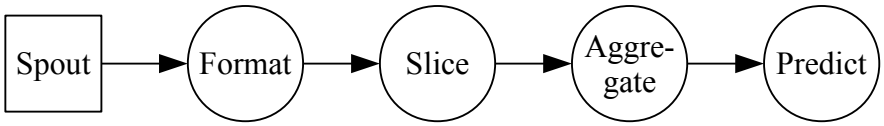

Figure 8.3: Load prevision topology

Application

Our application is a Storm topology that computes load's prevision based on the previously described data-set. Figure 8.3 describes the different components of our topology.

The spout retrieves the data from an MQTT topic where it is published line by line, these lines are then formatted into a list of values by the Format bolt. Then, only the load-related tuples are filtered and enriched with a time slice based on their timestamp and aggregated per house and time slice. Prevision is then made given the recent time slice history and previous time slices and stored in a Cassandra database.

8.3.2 Elasticity Validation

This experiment shows how our elastic solution is able to follow the variation of the in-coming throughput by adding new nodes to the Storm cluster. The initial cluster is set so as to have one worker per component of our topology, i.e., 6 virtual machine instances including the Nimbus node. The in-coming throughput is then varied gradually in order to see its impact on the behavior of our topology.

Figures 8.4 and 8.5 show the results of this experiment, components that have not needed scaling have been omitted. We can see that at times, there is a decrease in the number of emitted bolts (figure 8.4). This is detected by our elasticity controller which instigate the corresponding scaling operation (figure 8.5). The pikes following each scaling are due to accumulated non executed tuples when scaling is carried out. Likewise, when the spout's throughput is low enough our controller removes unnecessary instances.

Note that different components are scaled separately. Format has been the first to be scaled, and as it became able to process all the tuples its received, it emitted more tuples which resulted in Slice being overloaded later on. Slice's throughput is roughly half the throughput of the previous components as it filters only load related measurements which accounts for half the total reported measurements.

Thanks to its elasticity, our solution has been able to efficiently support the increasing dataload it to which it has been subjected.

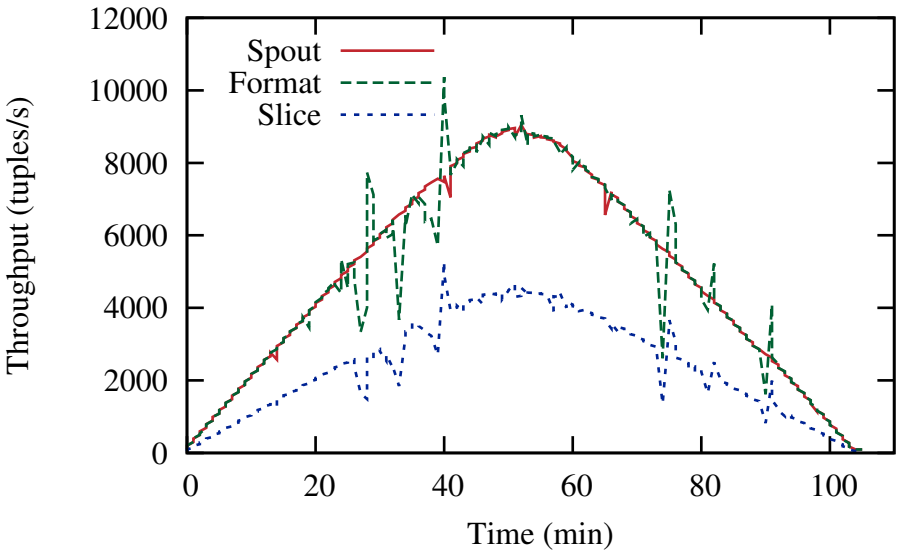

Figure 8.4: Throughput per component

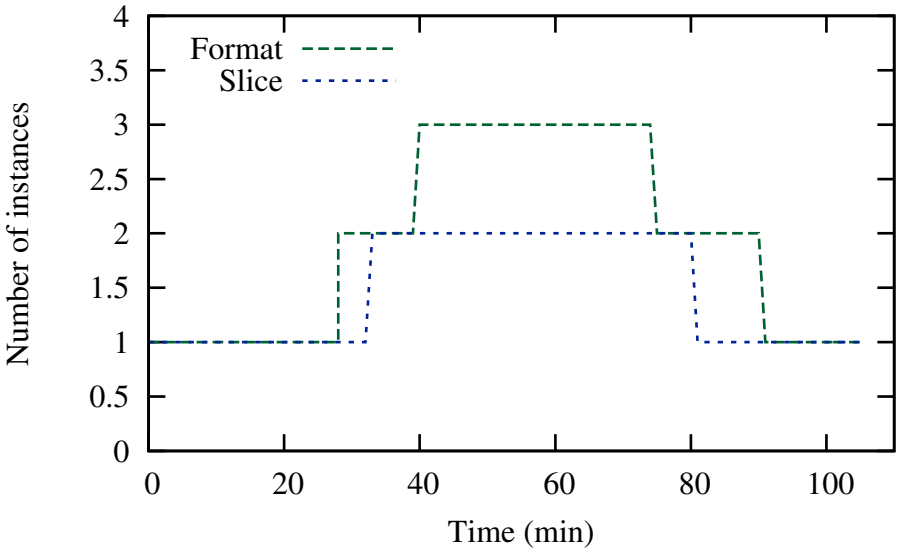

Figure 8.5: Parallelism per component

8.4 Conclusion

In this chapter, we have proposed an elastic stream processing solution based on Apache Storm. It is non intrusive and allows Storm users to define their topologies normally as well as benefit from the enhanced reliability of Apache Storm. The different processing components are scaled separately when the need to do so arises. Finally, this work has been validated through evaluation on a VMware vSphere private cloud solution using a real-life dataset.

Chapter 9

Conclusion

Contents

9.1 Summary

Elasticity is necessary to achieve energy efficiency in cloud computing environments. After a survey of existing elastic solutions and the techniques they put in practice, we have experimented with elasticity using applications from different horizons. In our elastic consolidation solution, the configuration to be consolidated is partitioned and fed to an estimated number of necessary workers, in order to speed the consolidation plan's computation up. Our elastic queues handle the load of published messages by introducing a flow control based load balancing policy, and highlight the utility of co-provisioning, i.e., having more than one replica per virtual machine instance and pre-provisioning, i.e., having a pool of idle virtual machine instances in order to speed scaling up. The elastic topics we proposed provide a seamless way of reconnecting JMS subscribers to dynamically created sub-topics, in order to keep the delay of message reception acceptable. Finally our elastic stream processing solution automatically scales different components independently in order to keep up with the varying throughput of in-coming data. These works have been evaluated using private clouds run by OpenStack or VMware vSphere, as well as Amazon EC2, the leading public cloud provider.

9.2 Perspectives

9.2.1 Different elasticity approaches

In this works' contribution, we have limited ourselves to reactive elasticity policies as we believe it to be adapted to every type of workload. However, experimenting with predictive policies, using different models, and comparing them to reactive policies can be of great interest. Moreover, in our elastic solutions, the scaling decisions have been mainly motivated by performance, defining utility functions that take into account other parameters as well, and comparing their result would be a relevant future contribution.

9.2.2 Advanced stream processing elasticity

With the outburst of big data and the increasing need for efficient stream processing tools, we intend to focus our efforts on improving our elastic stream processing tool. This can be achieved by integrating predictive models for scaling, as scaling a component inevitably impacts the components that receive its results. We can also integrate works on components' placement such as [2] into our solution, or go beyond them to achieve a better suited placement policy to elasticity. Finally stream processing can also be improved by tuning the load balancing policies of the different components, based on the frequency of particular tuple values for instance.

Bibliography

[1] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, pages 27–37. ACM, 2006.

[2] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218. ACM, 2013.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[4] CERN. The grid. 2006. `https://cds.cern.ch/record/976156/files/it-brochure-2006-002.pdf`.

[5] Clovis Chapman, Wolfgang Emmerich, F Galan Marquez, Stuart Clayman, and Alex Galis. Elastic service management in computational clouds. *CloudMan*, 2010.

[6] Josep Oriol Fitó, Inigo Goiri, and Jordi Guitart. Sla-driven elastic cloud hosting provider. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 111–118. IEEE, 2010.

[7] G. Galante and L.C.E. de Bona. A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 263–270, November 2012.

[8] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. Spade: the system s declarative stream processing en-

gine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134. ACM, 2008.

[9] Jeremy Geelan. Twenty-one experts define cloud computing. *Virtualization Journal*, January 2009. `http://virtualization.sys-con.com/node/612375`.

[10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[11] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, pages 23–27, 2013.

[12] Fabien Hermenier, Julia Lawall, and Gilles Muller. Btrplace: A flexible consolidation manager for highly available applications. *IEEE Transactions on dependable and Secure Computing*, page 1, 2013.

[13] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.

[14] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology. 2001.

[15] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, 2008.

[16] IBM. An architectural blueprint for autonomic computing. Technical report, IBM, 2003.

[17] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 11(6):589–603, June 2000.

[18] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[19] Thomas Knauth and Christof Fetzer. Scaling non-elastic applications using virtual machines. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 468–475. IEEE, 2011.

[20] Eric Knorr and Galen Gruman. What cloud computing really means. *InfoWorld*, April 2008. `http://www.infoworld.com/d/cloud-computing/what-cloud-computing-really-means-031`.

[21] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM, 2009.

[22] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012.

[23] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52. IEEE Computer Society, 2010.

[24] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[25] Paul McFedries. The cloud is the computer. *IEEE Spectrum Online*, August 2008. `http://spectrum.ieee.org/computing/hardware/the-cloud-is-the-computer`.

[26] Brian Melcher and Bradley Mitchell. Towards an autonomic framework: Self-configuring network services and developing autonomic applications. *Intel Technology Journal*, 8(4), 2004.

[27] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), September 2011.

[28] Shicong Meng, Ling Liu, and Vijayaraghavan Soundararajan. Tide: achieving self-scaling in virtualized datacenter management middleware. In *Proceedings of the 11th International Middleware Conference Industrial track*, pages 17–22. ACM, 2010.

[29] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling tcp throughput: A simple model and its empirical validation. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 303–314. ACM, 1998.

[30] Janak Parekh, Gail Kaiser, Philip Gross, and Giuseppe Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, 2006.

[31] Ahmed El Rheddane, Noël De Palma, Fabienne Boyer, Frédéric Dumont, Jean-Marc Menaud, and Alain Tchana. Dynamic scalability of a consolidation service. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 748–754. IEEE, 2013.

[32] Ahmed El Rheddane, Noël De Palma, Alain Tchana, and Daniel Hagimont. Elastic message queues. In *Cloud Computing (CLOUD), 2014 IEEE Sixth International Conference on*. IEEE, 2014. (To appear).

[33] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507. IEEE, 2011.

[34] Libor Sarga. Cloud compting: An overview. *Journal of Systems Integration*, 3(4):3–14, 2012.

[35] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[36] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 559–570. IEEE, 2011.

[37] Christophe Taton, Noel De Palma, Sara Bouchenak, and Daniel Hagimont. Improving the performances of jms-based applications. *International Journal of Autonomic Computing*, 1(1):81–102, 2009.

[38] Nam-Luc Tran, Sabri Skhiri, and Esteban Zimányi. Eqs: An elastic and scalable message queue for the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 391–398. IEEE, 2011.

[39] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *SIGCOMM Comput. Commun. Rev.*, 41(1):45–52, January 2011.

[40] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[41] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. *ACM SIGARCH Computer Architecture News*, 40(1):423–436, 2012.

[42] Smita Vijayakumar, Qian Zhu, and Gagan Agrawal. Dynamic resource provisioning for data streaming applications in a cloud environment. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 441–448. IEEE, 2010.