Dynamic resource allocation for elastic systems based on scalability modeling

Paulo Bittencourt Moura

Text submitted
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
for the
doctoral degree
in
Computer Science

Advisor: Prof. Dr. Fabio Kon

During this research, the author received funding from CAPES, CNPq, and the CHOReOS project, funded by the European Comission.

São Paulo, 2017, January

Dynamic resource allocation for elastic systems based on scalability modeling

This version of the thesis contains the changes suggested by the Committee Members during the public defense of the original version of this work, which occurred on Mar 17th, 2017. A copy of the original version is available at the Institute of Mathematics and Statistics (IME) of the University of São Paulo (USP).

Committee members:

- Prof. Dr. Fabio Kon (Advisor) - IME-USP
- Prof. Dr. Alfredo Goldman - IME-USP
- Prof. Dr. Daniel Cordeiro - EACH-USP
- Prof. Dr. Raphael Yokoingawa de Camargo - UFABC
- Prof. Dr. Markus Endler - PUC-RIO

Acknowledgement

First, I would like to thank my advisor, Fabio Kon, for the support and guidance along this thesis. He taught me a lot in this period. I also would like to thank professors Maarten van Steen and Spyros Voulgaris, who received me in the Netherlands, first at VU Amsterdam, and later at the University of Twente. I learned a lot from their advices and they also provided me access to the infrastructure in which I executed most of the experiments in this thesis. I thank both of them for the valuable time they invested in helping me with my research. Finally, thank you all to my colleagues from our research group at IME and my family. Specially to my wife, Nathália.

This research received financial support from CAPES, CNPq, and the European Commission.

Abstract

Cloud computing is a new paradigm in which virtual resources are leased in the short-term. Cloud providers publish an API through which users can request, use, and release those resources. Thus, a properly architected system can be quickly deployed and their infrastructure can be quickly updated to better accommodate workload fluctuations and limit expenses. Many services running in clouds comprise an automated resource management unit, which is in charge of requesting and releasing resources without human intervention, as demand changes. The rule based approach, commonlly applied to automate the resource management, is especially problematic in cases of load surge. When of a quick and drastic increase of the workload, the system may take many cycles of infrastructural redimensioning until achieve an adequate state. In this case, the system remains overloaded during all those cycles, affecting user experience. In this research, we investigate how we can properly understand what are the effects, in system capacity, incurred by variations in resource availability, and how this knowledge can be applied to improve elasticity. We propose a strategy that comprises performing scalability tests to model scalability and apply the model to estimate resource need, according to the arriving workload. We introduce a framework for automated scalability evaluation of distributed systems and experimentally evaluate the proposed strategy. We compare the allocation and performance obtained using our strategy with a rule based strategy in a trace-driven simulation and with synthetic workloads. We also evaluate six variations of the model-based approach. Generally, our approach can deliver better performance, while increasing resource allocation and, consequently, cost. The extent of the performance improvement is larger than the cost increment, though.

Keywords: scalability, scalability testing, scalability modeling, cloud computing, elasticity

Resumo

MOURA, P. B. **Alocação dinâmica de recursos em sistemas elásticos baseada em modelos de escalabilidade**. 2017. 91 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2017.

Provedores de serviços de nuvem disponibilizam uma interface através da qual seus clientes podem solicitar, usar e liberar estes recursos. Muitos serviços implantados em nuvens incluem um componente para gerenciamento automatizado de recursos, encarregado de requisitar e librar recursos sem intervenção humana, à medida que a demanda varia. A técnica padrão para o gerenciamento de recursos se baseia em regras sobre utilização de recursos. Quando ocorre um aumento significativo na carga em um curto espaço de tempo, o sistema pode levar vários ciclos de monitoramento e ação até alcançar uma configuração adequada. Neste período, o sistema permanece sobrecarregado. Nesta pesquisa, investigamos como compreender adequadamente os efeitos da variação na disponibilidade de recursos sobre a capacidade de um sistema e como aplicar este conhecimento para melhorar sua elasticidade. Propomos uma estratégia que abrange avaliação da escalabilidade do sistema, visando sua modelagem, e a aplicação deste modelo nas estimativas de necessidade por recursos com base na carga de trabalho. Introduzimos um arcabouço para automatizar a avaliação de escalabilidade de sistemas distribuídos e efetuamos uma validação experimental da estratégia proposta. Comparamos a alocação de recursos e o desempenho obtido usando nossa estratégia e estratégia baseada em regras, fazendo a reprodução de carga real e usando cargas sintéticas. De forma geral, nossa proposta foi capaz de prover melhor desempenho, ao ponto que o uso de recursos cresceu, e consequentemente o custo de utilização. No entanto, a melhora de desempenho foi mais significativa que o aumento dos custos.

Palavras-chave: escalabilidade, teste de escalabilidade, modelagem de escalabilidade, computação em nuvem, elasticidade.

Contents

Acronyms

ANOVA Analysis of Variance. 4

API Application Program Interface. 2

AWS Amazon Web Services. 1

EC2 Elastic Compute Cloud. 6

IaaS Infrastructure-as-a-service. 6

IT Information Technology. 1

MARS Multivariate Adaptive Regression Splines. 12

P2P Peer-to-peer. 20

PaaS Platform-as-a-service. 6

QoS Quality of Service. 3

RIA Rich Internet Applications. 19

S3 Simple Storage Service. 6

SaaS Software-as-a-service. 6

SLA Service Level Agreement. 7

SOA Service Oriented Architecture. 9

SUT System Under Test. 14

USL Universal Scalability Law. 13

VM Virtual Machine. 7

XML Extensible Markup Language. 20

XSLT Extensible Stylesheet Language Transformations. 20

List of Figures

List of Tables

Chapter 1

Introduction

When the Internet was created, it was hard to imagine the significance it would have in our lives and economy, nowadays. The Internet is a common means for people to interact with each other, consume entertainment, and much more. Facebook reported 1.79 billion active users in September 2016, which is an increase of 16% year-over-year [Fac16]. In the same period, Twitter reported 317 million active users, increasing 3% year-over-year [Twi16]. Google's core products, which include Google Search, Android, Maps, Chrome, YouTube, Google Play, and Gmail, have more than one billion monthly users each [Alp16]. Business also experienced a shift to the Internet. Amazon and Netflix are prominent examples; they were born in the Internet. The former as an online bookshop; the latter, with media streaming. Amazon expanded its business to different areas, mostly Internet centered. In 2015, it achieved over US$100 billion in net sales from which more than 70% came from electronics and other products (the remaining include media streaming, cloud, and others). In doing so, Amazon is the fastest company ever to reach that milestone. In the same year, Amazon Web Services (AWS), their cloud computing branch, achieved more than US$10 billion in sales – it is bigger than Amazon was at the same age [Ama16a]. Netflix, with over 86 million users watching more than 125 million hours of TV shows and movies per day, is the leading Internet television network [Net16].

Those companies demand an enormous Information Technology (IT) infrastructure to sustain their daily activities. Understanding what is the relation between their resource need and the received workload is crucial to provide satisfactory user experience and avoid excessive costs. In addition, system architecture must allow that it properly benefit from additional resources [MA01]. However, not rarely, we find systems inadequately prepared to deal with workload fluctuations.

A common case is of a startup that releases a service to a small number of clients and progressively increases its audience, demanding more and more infrastructure with time. Some services, however, are intended to be used by a very large audience since their deployment. In this case, an accurate evaluation is required before releasing the system to guarantee that it will perform properly and that it can support an even larger workload in the beginning of the operation. This requirement is quite relevant because, on release, a considerable amount of the potential users may want to try it, resulting in a usage peak. After this initial phase, the workload might reduce, being distributed over the time. Thus, the system must start operation with the possibility to consume a considerable amount of resources but, afterward, part of these resources could be unbound to reduce costs. Neglecting this step can take to bad functioning, as happened to the USA government www.healthcare.gov service, which allows people to compare private health care plans. The site presented instability during its first two months online, reporting availability of just 43% one month after the release. The situation has been overcome after hardware and software updates[1]. Similarly, on Apple's MobileMe service launching, which included the migration of .Mac accounts, the service stayed unstable or unaccessible for several hours[2].

Another case for the need to understand the relationship between workload and resource demand

[1] http://www.usatoday.com/story/news/politics/2013/12/01/federalexchangmeetsgoal/3795523/
[2] https://www.cnet.com/news/mac-migration-to-mobileme-hits-some-roadblocks/

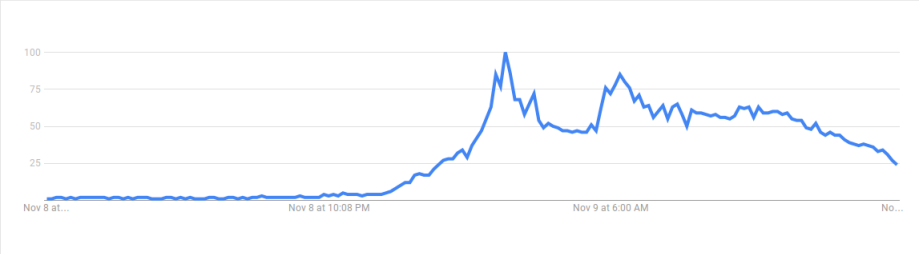

Figure 1.1: *Variation in the frequency of the search for the terms "Move to Canada" made by users in the USA after the election of Donald Trump for president – extracted from Google Trends.*

is on the occurrence of unexpected load spikes; cases in which a service is quickly overloaded by an abnormal number of user requests. This phenomena can happen, for instance, when an article in a large audience site links to a low traffic one, in what is known as crowd effect. For instance, after the United Kingdom voted for leaving the European Union in June 2016, in a popular referendum, a petition for a second referendum, widely shared on social media, received more than 100,000 new signatures per hour. The www.parliament.uk website crashed due to the high demand[3]. Something similar, but not exactly triggered on the Internet, happened to the Kind Campaign website. It was mentioned by the actor Aaron Paul during his speech at the Emmys Award 2014. The Kind Campaign's website crashed with the sudden interest and remained down for several hours[4]. In some cases, the services are simply not prepared to support marketing campaigns the enterprises engage, such as Mozilla campaign on the release of Firefox 3.0, pursuing a record of five million downloads in a day, which overloaded their servers [5]. During the 2015 Black Friday in Brazil, several sites did not support the load spikes and crashed[6]. Also, during the #UberIceCream campaign in Brazil, in July 15th, 2016, when Uber promoted the free delivery of ice cream in selected cities, its service was unavailable in some regions for about two hours, due to high demand [7].

In a last example, we see what happened after the 2016 presidential election in the USA. With the victory of the Republican candidate, Donald Trump, Canada's immigration website crashed due to high traffic. As an indicator of the sudden interest in information about migrating to Canada, we can see the increase in the search for "Move to Canada" made on Google by users in the USA. Figure 1.1, extracted from Google Trends, shows that the frequency of searching the terms was nearly constant, with a sudden rise that, at the peak, was 10 times higher that the average until then.

In the beginning of the rise of the commercial growth of the Internet, in the late 1990s, planning and anticipating demands was essential. To be able to satisfactorily handle increasing workload, enterprises needed to plan, buy, deploy, and maintain more infrastructure in anticipation to their estimations. Notwithstanding, acquisition and maintenance costs were high. On the one hand, underprovisioning could lead to revenue loss, bounding the company's capacity to provide its services to potential customers. On the other hand, overprovisioning could pressure the margins, thanks to the high expenses. Buying infrastructure to support a volume of users which is not materialized could cause trouble as the obligations incurred in buying and maintaining such infrastructure are not supported by revenues below expectations.

In parallel with the boom of companies requiring fast Internet connectivity and nonstop operation, there were rising data center enterprises. Those had a big farm of servers to rent [Bar11]. Yet

[3] https://www.theguardian.com/politics/2016/jun/24/petition-second-eu-referendum-crashes-house-of-commons-website

[4] http://www.smh.com.au/entertainment/tv-and-radio/aaron-pauls-emmys-acceptance-speech-crashes-kind-campaign-website-20140826-108lrf.html

[5] http://www.siliconbeat.com/2008/06/17/firefox-download-stunt-sets-record-for-quickest-meltdown/

[6] http://tecnologia.uol.com.br/noticias/redacao/2015/11/27/black-friday-no-brasil-comeca-com-instabilidade-nos-sites-de-compras.htm

[7] http://www.infomoney.com.br/minhas-financas/consumo/noticia/5319518/uber-fica-instavel-dia-entrega-gratis-sorvete

simplifying the operations of the companies, it could take months to deploy new applications in those data centers. This paradigm changed with the emergence of cloud computing (Section 2.2), which brought flexibility to resource management. The key idea is that, instead of long term leasing of actual IT infrastructure, with help of a dedicated Application Program Interface (API), the companies can quickly allocate and deallocate virtual resources, according to their need, paying per usage, in the short term. Thus, applications could comprise self-management mechanisms. Closely monitoring the performance of applications provides information for updating resource allocation, aiming at a balance between system capacity and the costs to maintain them in operation.

The common approach to automate resource management is based on utilization. For so, the servers are constantly monitored and utilization counters are periodically verified. When a threshold, established in some reallocation rule, is triggered, the action defined by the given rule is executed, requesting or releasing resources. The problem with this approach is that utilization cannot grow beyond 100% and, especially at high loads, the utilization does not increase proportionally to the workload. Hence, it is not possible to know, from the utilization measurements, what is the amount of resources necessary to properly handle the workload, providing adequate performance. Thus, a fixed quantity of resources is set for allocation or deallocation in each rule.

The utilization-based strategy is particularly problematic in cases of sudden workload variation. Consider the case when an e-commerce announces a big sell-off. More and more people receive the advertisement and go to the web site. As the request rate grows at a fast pace, utilization goes to the limit, an allocation task is triggered, and a new server is added to the system infrastructure. But the workload increment was higher than what the new allocation supports and the system remains saturated. Only in the next monitoring cycle, the allocation can grow again. In the mean time, the system is overloaded, potentially affecting the performance, perceived by the users in terms of long response times, and damaging user experience. In the next monitoring cycle, when the allocation grows once again, the problem may still persist.

In this thesis, we address the problem of getting to understand how a given system scales, i.e., how the variation in its resource availability affects its capacity to serve requests, and how to apply that knowledge to improve auto-scaling.

1.1 Objectives

The main goal of this research is to provide means for analyzing distributed systems deployed in clouds, building a scalability model of the system, and applying it to improve its elasticity (Section 2.3), especially by providing better reaction to drastic, unexpected variations of the workload. The objective is guided by the following research questions:

RQ1: Which elements and procedures are required for scalability evaluation of distributed systems?

RQ2: How to produce a meaningful assessment of the scalability of a distributed system?

RQ3: How that assessment can be used to improve a system's Quality of Service (QoS)?

The answer for those questions are pursued by aiming at the following objectives:

1. Studying and understanding how scalability tests can be planned and executed;
2. Studying and understanding how scalability can be analyzed and represented;
3. Studying and understanding how elasticity can be implemented to provide better QoS;
4. Building a framework to create and execute automated scalability evaluations (tests and analysis) of distributed systems;

5. Proposing a strategy for improved elasticity based on the outcome of the scalability analysis;
6. Empirically validating and refining the proposed strategy.

1.2 Contributions

The main contributions of this thesis research, and respective scientific publications, are:

- The Scalability Explorer – a framework to support the creation of automated scalability tests, available as free software, under the Mozilla Public License 2.0, at https://github.com/choreos/choreos_v-v/tree/master/scalability_explorer.

  A preliminary version of the Scalability Explorer was introduced, with Rehearsal, in a paper published in the tool session of the Brazilian Conference on Software: Theory and Practice (CBSoft), in 2012 [BMKM12] A paper presenting an architectural update of the Scalability Explorer, and proposing the application of Analysis of Variance (ANOVA) [CB02] for scalability evaluation, was published in the International Workshop on Automation of Software Test (AST), held in conjunction with the International Conference on Software Engineering (ICSE), in 2013 [MK13]. Another Rehearsal's publication, in the Journal of Systems and Software (JSS), in 2015 [BMKM15], also contained the description of the Scalability Explorer and utilization instructions as part of a test-driven methodology for development of web service choreographies.

- a new set of strategies for system elasticity, enabling quick, dynamic adjustments of the resource pool to provide improved performance maintenance especially in cases of drastic, unexpected variation of the workload as well as an analysis of the cost incurred by these strategies.

  We published a paper presenting the validation of the applicability of the Universal Scalability Law in our strategy at the Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC), held in conjunction with the Symposium on Principles of Distributed Computing (PODC) in 2015 [MKVvS15]. Another paper, presenting the first results and insights obtained from experimental validation of the strategy, was published in the International Conference on High Performance Computing & Simulation (HPCS) in 2016 [MKVvS16]. We are currently working in a journal paper including the empirical validation of our proposal, analyzing six variations of the strategy, and using synthetic workloads and a trace-driven simulation.

1.3 Thesis organization

The next chapter covers basic concepts related to this thesis, namely: cloud computing, scalability, elasticity, quality of service, performance, service oriented architecture, scalability modeling, and scalability evaluation. In Chapter 3, we discuss related works on automation of scalability evaluation and automated resource management in cloud systems. Chapter 4 introduces the Scalability Explorer, which is a framework to support the automation and scalability evaluation and Chapter 5 brings our proposal for dynamic resource management for elastic systems. In Chapter 6, we discuss our findings and threats to the validity of our proposals and validations, and the final chapter brings our conclusions.

Chapter 2

Concepts

This chapter covers concepts that ground-base the subject of this research. It starts with an overview of quality of service, then cloud computing, which is followed by a conceptualization of scalability and elasticity and a brief description of Service Oriented Architecture, which is a common and recommend approach to build scalable applications, highlighting some implications of modeling distributed systems. Then, it moves to an overview of the main approaches for scalability modeling, also describing in detail the model that we selected to apply in the validation of our proposal for automated elasticity, the Universal Scalability Law (USL). Finally, it brings what are the common tasks for scalability evaluation of distributed systems.

2.1 Quality of Service (QoS) and performance

Quality of Service (QoS) refers to non-functional characteristics and needs of distributed systems. Quality constraints may affect the performance perceived by the user and, thus, some guarantees may be necessary to provide an appropriate user experience, especially in real-time systems. Those needs are expressed in terms of QoS constraints.

QoS management is a means to express requirements for a distributed system so that it performs its operations with appropriate quality, providing an adequate experience to the user. Those requirements are defined in terms of what is necessary from the network, CPU, memory, and other related systems. Originally, the main parameters used in QoS specifications are required bit rate, maximum delay until establishing a session, maximum end-to-end delay, jitter, and response time [TvS07]. But it has been extended to express more directly parameters that affect user experience, such as security and performance [CDKB13].

In multimedia systems, there is a need for the packets containing audio and video data to be processed in a particular order and within a certain rate that makes the media stream play seamlessly. Thus, it is required that the packets arrive in time and that they are completely processed before the next packet is processed. These systems may have associated requirements to guarantee a satisfactory user experience. For instance, if transmitting a media stream at a certain quality demands 1Mbps, sending two such streams would require a bandwidth of at least 2Mbps. If this is not available, one of the streams should be blocked so that the other is served adequately [KR12]. Alternatively, we could consider decreasing the quality of the transmitted media, so that both can go though the current link, or to increase the link to support the required transmission rate.

That reasoning is also valid for other aspects of software services. On the Web, it has been observed that long waiting times to load a page take the users to abandon the site [Nah04]. Hence, consider a service deployed in a given server, which is using 80% of its CPU capacity to serve 1000 simultaneous users. Doubling the workload (to 2000 simultaneous users) would stress the server and the service performance would degrade. The possibilities to maintain the server performance, delivering an adequate user experience would include: denying some connections so that the response time of the accepted ones do not grow beyond the established limit; updating the software, making it lighter and capable of processing the increased workload adequately; or upgrading the hardware,

giving it more processing power.

Commonly used performance metrics [Jai91] that reflect the user experience are:

- **Response time**: the interval between the user request and the system response;
- **Reliability**: usually measured by the probability of the system returning an error or the mean time between consecutive errors;
- **Availability**: the fraction of time the system is available to service user requests.

Other metrics are not directly perceived by the users, but contribute to their experience. These metrics are related to system capacity and the performance from the server perspective:

- **Throughput**: rate (requests per unit of time) at which the requests can be served by the system. For interactive systems, it is measured in requests per second. For networks, packets per second or bits per second;
- **Load**: usually measured by the number of simultaneous users, parallel process, or requests queue size;
- **Utilization (of a resource)**: the fraction of time the resource is busy servicing requests. Some resources, such as processors, are always either busy or idle, so their utilization is measured in terms of ratio of busy time to total time. For other resources, such as memory, only a fraction of the resource may be used at a given time, thus, their utilization is measured as the average fraction used over a time interval.

When load and/or utilization surpasses certain limits (which defines the service capacity) the service is stressed and its performance (e.g., throughput) can stagnate or degrade. With that, the performance perceived by the user is affected, initially in terms of response time increments. If the cause of such stress is not handled, this may lead to reliability or availability issues. Thus, the importance of identifying parameters that contribute to the user experience and making an effort to keep the system working in adequate conditions.

2.2 Cloud computing

Cloud computing [BPcV12] was a game changer. It is a new paradigm, which delivers a utility computing vision where resources are exposed over the Internet and can be accessed on demand. There are usually three roles involved: the **cloud provider** owns the infrastructural resources, providing them as a service; the **cloud customer/solution provider** deploys and delivers services that use those resources provided by the cloud; the **final customer** uses the provided services to its benefit. Thus, we have cloud providers and solutions providers providing resources at different levels, and the solution provider and final customer consuming resources at different levels. In some cases, the same part plays more than one role simultaneously.

The consumers reduce their need to maintain on-premise servers and can adapt their costs more efficiently, on demand [BYV+09]. This is because the resource usage shifted to a short term leasing approach. The consumer can request resources, use them as wish, and release them, paying for the time it has kept the "possession" of the resources.

In cloud computing, different sorts of services are classified according to where, in the solution stack, they are. There are three levels:

- Infrastructure-as-a-service (IaaS): the service is in the form of an API and user interface, giving to the consumer the possibility to directly allocate, configure, use, and release computational resources in the form of storage, processing units, network, etc;

- Platform-as-a-service (PaaS): the service is a middleware capable of managing infrastructural resources. The consumer builds an application using the middleware API and the resource management is transparent;
- Software-as-a-service (SaaS): the provided service is a self-contained software application.

In this work, we focus the attention to IaaS, which is the most flexible in terms of possibilities from the point of view of software development. This trend started in 2006, when Amazon launched two products under the umbrella of AWS: Simple Storage Service (S3), for storage, and Elastic Compute Cloud (EC2), for computing. The legend is that it began due to Amazon's wish to monetize the excess of capacity that was required only in the holidays season. But outsourcing data centers was already a common practice and there is a huge gap between such wish and those products. What actually happened was that they figured out the benefits of the standardization and automation of the process of setting up the common infrastructure. The internal teams' activities, to build and deploy services, would face increasing productivity and reduced costs. So, they created the solution; to comply with internal demand. But they had in mind that such demand was commonplace. Thus, they publicly released the product. Considering the first nine months of 2016 (last report available when this text was written), AWS is the segment of Amazon with fastest growing net sales, with 59% against 29% consolidated growth. Also this is the segment in which Amazon obtains the highest operating margins [Ama16b].

The solutions provided by the cloud service are accessible via an API, which gives to the cloud customer access to the aforementioned automated procedures that promptly provide the necessary resources. In addition to the APIs, another enabler of cloud computing is virtualization. What the cloud users actually get are virtual resources. The servers in the cloud provider's data centers run a hypervisor, which uses the bare metal to offer Virtual Machines (VMs), which are isolated replicas that mimic a real machine. This mechanism enables the cloud provider to multiplex its resources, giving the appearance of infinite capacity [ASZ+10]. In addition, each cloud user has the option to choose operating systems and applications to use according to his own needs.

Another entity involved in this scenario is the Service Level Agreement (SLA). It is a contract between the cloud provider and user, stating guarantees and requirements in terms of QoS. Those agreements define the minimum QoS for some metrics, and the implication of violations to the agreed contract.

Pricing in clouds also has distinguishing characteristics. Due to the dynamicity of resource utilization, charging comes in fine granularity. Depending on the type of resource, charging can be based on metrics such as time slices, blocks of used Gigabytes, Network I/O, and other metrics, or a combination of them.

After a decade of widespread stories of successful adoption, and affecting virtually all aspects of IT, the prevalence of cloud computing is becoming an expected approach [Smi16]. It does not mean the IT infrastructure is completely moving to the cloud, but that, in the near future, corporations with no cloud-based service will be rare.

One case for cloud adoption is certainly that of highly variable or unpredictable workload. One distinguishing characteristic of cloud computing is the simplicity and speed of allocation and deallocation of resources. Benefits of quickly reacting to drastic variations in the workload come both in terms of QoS maintenance and finance. Service providers no longer have to anticipate the extent of occasional load peaks and purchase servers with power enough to support that, but that will be idle most of the time. At the same time, there is no need to decline requests to maintain the server performance [KMN04].

But moving a system to the cloud is not enough to give it access to all the benefits provided by cloud computing. Software engineers must adapt the system to benefit from the cloud infrastructure. The main attributes to enable a system to use cloud resources to properly respond to drastic workload changes are scalability and elasticity, detailed in the next section.

2.3 Scalability and elasticity

Although scalability has been studied in parallel computing, distributed computing, databases, and service-oriented architecture (SOA) [THBG12], we cannot find a general precise definition of what scalability means in Computer Science. Researchers define scalability in the most different ways. While in some cases we can identify misconceptions, in other cases we can consider that researchers simply attain to different facets of scalability.

The most recurrent misconception is to consider that scalability is linear [GGK93, THBG12, GTL+12, BKKL09, JW00, SVRG13], while there are evidences that linearity is hardly feasible [Amd67, Hil90, Gun07]. Also, there is an intuitive understanding that, with unlimited resources, a system should scale infinitely [BKKL09]. However, there are limits to how much a system can grow [Law98, Gun07].

Regarding different facets of scalability, generally, researchers refer to growing workload and may [THBG12, KBW12] or may not [GTL+12, KM12b] consider adding more resources to the system. The first case is sometimes called hardware scalability and, the latter, software scalability. Thus, **software scalability** concerns with making the software capable of achieving the maximum capacity provided by the available hardware. While **hardware scalability** refers to increasing the capacity by providing more resources to the system. In this thesis, unless explicitly stated, when we use the term scalability, we refer to hardware scalability.

The capacity is related to serving more request or clients. It is usually defined as the maximum throughput the system can achieve. The relative capacity gain C, also known as **scaleup**, gives the proportional gain of changing from one to n parallel processes [Gun07]. Throughput X, in turn, is the number of operations the system can process per time unit.

$$C(n) = \frac{X(n)}{X(1)} \tag{2.1}$$

Despite increasing capacity, another common reason to pursuit scalability is increasing performance [KBW12]. Consider a system running in a single process taking $T(1)$ seconds to process a workload. If at least part of this workload can be split in small independent tasks, parallel execution is possible and reduces the execution time. The **speedup** S is the relative gain obtained with the execution split in n parallel processes [Hil90].

$$S(n) = \frac{T(1)}{T(n)} \tag{2.2}$$

The difference between scaleup and speedup is in the point of view. While the former is concerned with doing more work, the latter focuses on doing the work faster. But both call on the same strategy: parallelism. Moreover, if a system can process x tasks per second in a single process, we have that

$$X(1) = \frac{x}{sec} = \frac{1}{T(1)}$$

Considering n parallel processes, we have:

$$X(n) = C(n)X(1) = \frac{C(n)}{T(1)} = \frac{1}{T(n)}$$

And, so

$$C(n) = \frac{T(1)}{T(n)} = S(n)$$

Therefore, we have that scaleup and speedup are numerically equivalent.

One broad definition of scalability is that it is "*a quality of software systems characterized by the causal impact that scaling aspects of the system environment and design have on certain measured system qualities as these aspects are varied over expected operational ranges*" [DRW07]. We consider

that this definition touches the relevant points we discussed here and more: it does not restrict scalability to performance, letting the possibility to relate scalability to other system qualities such as availability, reliability, dependability, and security. In fact, the maintenance of those qualities may be interrelated. For instance,

Scalability is often confused with performance. Although related, they are not the same thing. An algorithmic improvement is a valid approach for increasing system performance. But this is not a scalable strategy. Scalability means that the system can grow and shrink without fundamental changes to it [WG06]. Growing directly relates to performance when it promotes scaleup gain. But, different from an ordinary performance improvement, a scalability improvement can be systematically repeated a number of times [Sch06]. Scalability is, thus, an architectural characteristic. There are basically two approaches for scaling a software system: giving more power to the server, known as vertical scalability; or replicating the software in more of the same hardware and splitting the workload, known as horizontal scalability.

In the context of cloud computing, vertical scaling can be performed with two methods, depending on the provided functionalities. Some cloud providers allow redimensioning, giving the possibility to change VM characteristics, such as available memory and number of virtual CPU cores. In other frameworks, migration is the way to vertical scaling. A new, more powerful, VM is created to replace the old VM.

Vertical scaling looks simpler. Intuitively, changes like more memory, faster CPU, faster I/O, instantly improve the performance of most of the software systems. And keeping the system running in a single machine provides faster inter-process communication. Conversely, distributed systems require more data replication, as the nodes have their own memory space, and the communication is slower as it needs to go through the network bus. But vertical scaling is limited to what we can assemble in a single server, incurring in a limitation to system capacity, when we consider Internet scale. Thus, this is not encouraged, except in cases in which horizontal scaling is not practicable [Sch06].

Generally, the recommended approach is to scale horizontally and, in the context of cloud computing, this is performed by adding more virtual machines to the system. It requires that the system be properly designed to work distributed in a varying number of nodes. Hence, the system can benefit from the possibility to quickly allocate more/less resources to it as the workload varies. These are considered **elastic systems** [GD12]. The main purpose to make an elastic system is to meet performance objectives while minimizing costs, in spite of workload fluctuations.

Elasticity gives to cloud customers the ability to quickly request, receive, and release resources as needed [BPcV12]. Considering the most common scenario, the entry point for the system is usually a load balancer and there is a pool of available VMs running instances of the service (resource pool) to which the load balancer distributes the requests. Elasticity, thus, gives the possibility to quickly update the resource pool by adding and removing VMs. The triggering of those actions can be manual, reactive, or predictive.

In the first case, the responsibility for monitoring, analyzing, and performing the actions are left to the cloud user. The cloud provider only offers the interface through which the client interacts with the cloud infrastructure to request and release resources. This is the solution provided by clouds based on CloudStack[1], OpenStack[2], and others.

The reactive approach is provided by some commercial cloud solutions, such as AWS[3] and Azure[4]. Third party tools, such as Scalr[5] and RightScale[6], can provide this functionality to many different cloud infrastructures. Reactive elasticity relies on traditional control theory [VRMB11]. Several monitors collect measurements from the VMs and feed a controller. The controller makes decisions to operate on actuators, which, in this scenario, are the tools or cloud providers' APIs.

[1] http://cloudstack.apache.org
[2] http://www.openstack.org
[3] http://aws.amazon.com
[4] http://www.windowsazure.com
[5] http://http://www.scalr.com
[6] http://www.rightscale.com

The decisions are based on rules of the type `if CONDITION then ACTION`. The conditions are based on metrics and modifiers such as surpassing a threshold, presence, counting, etc. Each specific solution can provide different metrics and modifiers. Commonly, the conditions are based on metrics regarding utilization of resources such as CPU, memory, network, budget, energy, etc [DRD+16]. Therefore, we will refer to this approach as **utilization-based**.

Predictive solutions apply heuristics to try to anticipate workload variation and decide on how to scale resources. In section 3.2, we summarize research efforts to promote elasticity in cloud systems.

2.4 Service Oriented Architecture (SOA)

Distributed systems usually operate simultaneously in distinct processing environments, communicating by hardware/software protocol stacks over a network. The communication is slower than that using local components that share the same memory space. This limitation has important architectural implications related to latency, concurrency, and partial failure [WWW+94].

As the market economy is getting global, distributed systems are reaching a very large scale, requiring support for heterogeneity, decentralization, and fault tolerance. Service Oriented Architecture (SOA) [W3C04] is an approach for system design that helps systems remain scalable and flexible while growing. One of the major elements of SOA are the *Services*, which are self-contained business functionalities that can be part of one or more processes and can be implemented by any technology on any platform [Jos07].

Many successful online services rely on SOA. It enables the creation of software systems based on the composition of a collection of small services, each a self-contained autonomous unit responsible for a given functionality. Two critical distinctions of SOA are message orientation – services interact by exchanging messages – and coarse granularity – services tend to use a small number of operations [W3C04]. Thus, a service-oriented system is composed of a set of services, each with specific attributes, that communicate via message exchange. Conceptually, it implies the absence of a unified data storage, as usual in layered systems. Each service is in charge of managing data concerning it. Any interaction that services need to perform with data maintained by other service must be done via messages - through service's API.

One downside of SOA, in comparison to traditional layered systems, is that requests may need to go through more levels of the software stack (accessing many services), which can affect performance. On the other hand, functional partitioning is an important issue regarding scalability [Pri08], which may be achieved by adding more servers running service instances and distributing the workload by means of a load balancer. This approach enables each service to scale independently.

Increasing the number of service instances potentially increases the processing capability of the service by the same amount. However, specially when the service comprises a data layer, its scalability may be affected by some factors, as the discussed in the next section.

2.5 Scalability modeling

Modeling, for scalability analysis and evaluation, is fundamental to identify relations among system architecture, computational resources, and quality of service. That is a vast research field. In this section, we give a brief overview of the research and applications of scalability modeling. Then, we dive into one model, the USL, which we selected to use in our thesis research.

2.5.1 Overview

The approaches for scalability modeling are usually split into two categories: analytic models and empirical models.

Analytic models are supported by a theoretical basis that enable extracting useful information related to system performance and capacity. Here, we focus on queuing theory, which is the most common support for analytic modeling, but other sorts of models can be applied, including those

based on process-algebra, Petri-nets, and Stochastic Processes [BDIS04]. Queues [All90, Jai91] abstract the behavior of services or resources. In a queuing system, jobs arrive to a service central, are served by service units, and depart. When all the service units are busy, the arriving jobs must wait (are enqueued), affecting the overall time spent in the system. The main characteristics to describe a queue are the job arrival distribution, the service time distribution, and the number of service units. They may also be described in terms of service policy, number of jobs, and system capacity, although there are default values for those – if nothing is specified, queues are considered to serve in a first-come-first-served (FCFS) policy, with infinite capacity and population. Another common policy applied in software modeling is process sharing (PS) [BJHG04, UPS+05, XP09, DPC10]. Commonly, models consider a single class of jobs, but it is possible to model systems that serve different classes, as well [KB03, BJHG04]. Distinct job classes typify different types of jobs that may arrive to the services. Each class may have different characteristics regarding the queue parameters, such as request rate and service time. For instance, a system may provide operations to search, create, update, de delete items, and each operation has distinguishing characteristics in terms of the frequency they are used, resources they use, and the time the consume of each resource. Although not common in practice, the most commonly used distribution for both, job arrival and service time, is the exponential, due to simplicity and analytic support. This distribution has a memoryless property which simplifies the mathematic modeling, resulting in computable formulas [Szt10].

Some researchers abstract a whole system or a set of nodes in a single queue [MCCS08, XP09, KMN04]. But queues can be interconnected, creating a queuing network [UPS+05, DPC10, BJHG04]. Those networks require parameters for the probability of a job departure in a queue to produce an arrival in another queue and the average number of requests a queue demands from another. Yet, regarding queuing networks, they are considered open systems when external jobs arrive to the network, go through its queues or part of them, and leave the system; or closed systems, when a fixed number of jobs keeps circulating in the system [Jai91]. Among the common metrics calculated from queuing models and networks, we can mention system throughput, mean queue lengths, device utilization, and mean system time. The number of jobs the system can serve simultaneously is sometimes used as a parameter, sometimes as an output. Queuing networks can be used to identify bottlenecks. In some cases, we consider the bottleneck is the queue where the jobs spend more time [All90] and a capacity or performance improvement in the resource represented by that queue should bring better overall performance gain than improvements in other resources. In others cases, we consider the bottleneck is the queue with higher utilization [Jai91] and, to increase the system's capacity, we must increase the capacity of the resource represented by that queue.

As a computer is composed by a set of resources such as processors, main memory, cache memory, storages, network interfaces, etc., a precise model should consider all of it [All90, Jai91]. This would make the process of creating analytic models more complex, though. Analytic models are more commonly used in early stages of software conception. The insights extracted from the analysis can guide architectural decisions. The complexity and precision of the analytic models, though, face barriers at a certain level of granularity, when some components are very complex or we lack information about them [HDV08].

There is no single general way to model distributed systems using queuing theory and there are issues that limit their precision and the modeling power. Notwithstanding, researchers work on recognizing or overcoming them. Kounev and Buchmann [KB03] identified that estimation of response time can lose accuracy, mainly in large load scenarios, because software contention is hard to model with queues. Throughput and utilization results were extremely accurate in their experiments though. Gunther [Gun08] showed that throughput of systems with shared write follows a pattern that is not consistent with those obtained by common queueing models. He proposes modifications to the machine-repairman model, injecting performance loss due to process locks in shared write systems. Servers usually have a limited income buffer that may lead to some requests to be denied/discarded. The requesting application usually handles this issue by re-submitting the request periodically. Urgaonkar et al. [UPS+05] use an infinite capacity queue and a given probability to find a full buffer to model this issue. This probability is estimated as the probability

of dropping a job in a limited capacity queue. Mi et al. [MCCS08] model service time as a two-phase Markovian Arrival Process (MAP) to introduce burstiness in a queueing model. One state of the MAP is associated to low load and the other to high load, each with a corresponding mean service time.

Empirical models are based on the observation of a running application. While analytic models require knowledge about system internals to provide accurate estimations, in empirical models those details are abstracted. Only external observation of system behavior, obtained through measurement of metrics of interest, is used to create the model. In counterpart, building accurate empirical models require an operational application in a production-like infrastructure and time to obtain measurements.

We include two classes of scalability metrics among the empirical models. The first comprises metrics that compute how scalable a system is [GGK93, SR94, CS06, JW00, SVRG13] and can be used to compare two different configurations of the system (either in terms of software updates, infrastructural change, etc.). The second class of metrics compute a ratio related to resource utilization, indicating the probability of the need to re-scale the system [GWV05, LLCK09, WA02]. We see limitations in both approaches: in the former, scalability is considered to be linear; in the latter, we see the same limitation found in the utilization-based strategy for resource allocation – the metric does not identify the extent of the resource pool update.

Other empirical models draw a relation among system performance, workload, and resource availability. These models are usually built by hands of statistical inference. One approach is to model the relation as a sequence of linear functions valid in limited intervals [DPC11, BJHG04]. Some researches apply Multivariate Adaptive Regression Splines (MARS) [CW00, HWSK10], which are, roughly, a set of very small lines. MARS can provide very accurate models as the small range of consecutive lines can capture short term variation, including discontinuity. This irregularity is observed, for instance, in fine-grain analysis, when variable data sizes affect cache usage. However, that precision may require large sample datasets and model coverage is still limited to dataset coverage. These models make no assumption about the relation between dataset dimensions.

In contrast, the so called **scalability laws** consider constraints to scalability. The rationale is that the ideal situation is when performance grows linearly with parallelization but, in reality, the gain is often limited by some factor. The first and most famous one is Amdahl's Law [Amd67], in which the scalability is considered to be bound by non-parallelizable portions of the workload, whose execution time is proportional to the problem size. This constraint leads to sub-linear speedup. Considering the serial portion of the workload s, we have:

$$T(n) = sT(1) + \frac{(1-s)T(1)}{n} \tag{2.3}$$

Replacing Eq. 2.3 in Eq. 2.2, we have:

$$S(n) = \frac{n}{1 + s(n-1)} \tag{2.4}$$

whose shape is the one drawn by the red line in Fig. 2.3, with asymptotic limit at a^{-1}. The contention incurred by the serialization dominates the total execution time as more processors are used. It limits the benefit of increasing parallelism because only the time spent during parallel execution is reduced.

Gustafson's Law [Gus88] considers that the serial portion of the workload does not grow with the problem size and that the amount of work which is parallelizable varies linearly with the number of processors. Hence, the runtime is fixed, instead of the problem size. Thus, speedup gain is linear with the number of processors.

$$S(n) = n + (1-n)s$$

This is considered a scaled speedup, because the problem size grows with n. This is of interest in some kinds of applications for which, when the computational power is increased, instead of

obtaining faster response time it is desirable to increase the problem size to obtain a more accurate solution.

Amdahl's and Gustafson's Laws only consider the portion of the workload that is sequential as a parameter to define the speedup. Sun and Ni's Law [SN90] consider the case of multicomputers, which are composed of a set of nodes, each with its main memory and processors. In this case, the available memory increases linearly with the number of processors. It brings possibility to another case of scaled speedup, the memory-bounded, in which the workload W is increased to keep a fixed rate of memory usage per processor. In this case, both problem size and execution time may increase, and the speedup is given by

$$S(n) = \frac{W(1) + G(n)W(n)}{W(1) + \frac{G(n)W(n)}{n}}$$

where $G(n)$ is an application-dependent function to indicate how the parallel workload is affected by memory increase. Gunther proposed the Universal Scalability Law (USL) [Gun93], which is detailed in the next section, considering time penalties related to synchronization overhead.

As the scalability laws pre-establish the shape of the scalability curve, the model is a mathematical relation that extrapolates the boundaries of the dataset used for inference. The fact is that no model can exactly represent any system. We must seek for a model that provides useful approximation of a system's behavior so that data extracted from it is meaningful. We consider that, in the context of this work, scalability laws are the most appropriate choice and, in the next section, we describe in details the USL, which we chose as a starting point, specially because it considers the performance loss induced by data synchronization.

2.5.2 Universal Scalability Law (USL)

As defined by Gunther, the Universal Scalability Law (USL) [Gun93, Gun08] tries to explain performance improvement via parallelism but also considering constraints that limit the speedups.

One such constraint is the assumption that workload processing is rarely completely parallel. Even if each task execution is independent, there normally are managerial tasks, such as load balancing or splitting and merging data, that run sequentially. Also, often a large number of processes compete for the same processor and occasionally need to wait. Such sequential portions incur contention delays.

Fig. 2.1 shows how contention limits speedup obtained from parallelism. Without contention, changing a system architecture from one to four processes would bring down execution time to one fourth. With contention, however, the reduction in execution time is smaller. Contention limits system speedup through parallel processing because it does not improve the execution time of the sequential portions. The more parallel processes used, the larger the proportion of sequential execution time, because the latter is constant while the parallel execution time drops. Thus, there is a point after which there is no meaningful gain in increasing parallelism, as the total execution time is dominated by the sequential execution time, a fact first described by Amdahl [Amd67].

Besides contention, systems may need to deal with data exchange among tasks executing in parallel, referred to as coherency. Coherency delays are caused by the need to bring shared data into a consistent state. Whenever one of multiple independent processing units needs to save data, it must disseminate the operation so that all units update their data, maintaining consistency among them all. This demands extra time.

Coherency increases the execution time of each parallel processing unit. As all processing units must synchronize with each other, the number of messages exchanged for data synchronization increases quadratically with the number of processing units. Therefore, more parallelism means more time spent by each unit for synchronization. Hence, coherency constraints are more limiting than contention, because after a certain degree of parallelism, synchronization time is so high that performance drops, as depicted in Fig. 2.2. In this example, we ignore contention. But we see that, when splitting the execution into four processes, instead of having the execution reduced to

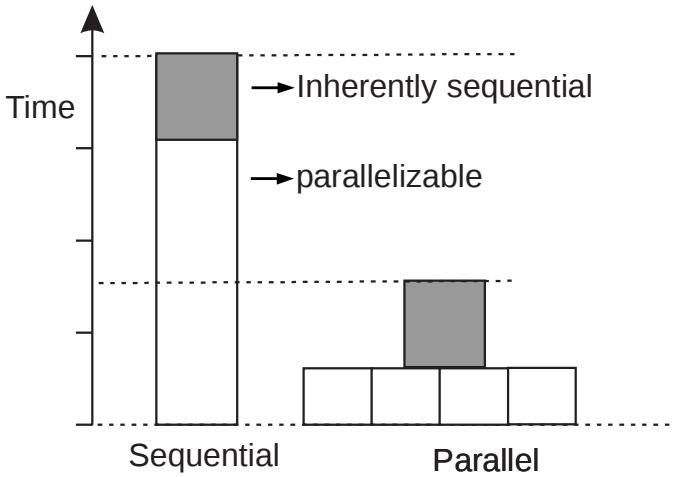

Figure 2.1: *Contention limits performance gains.*

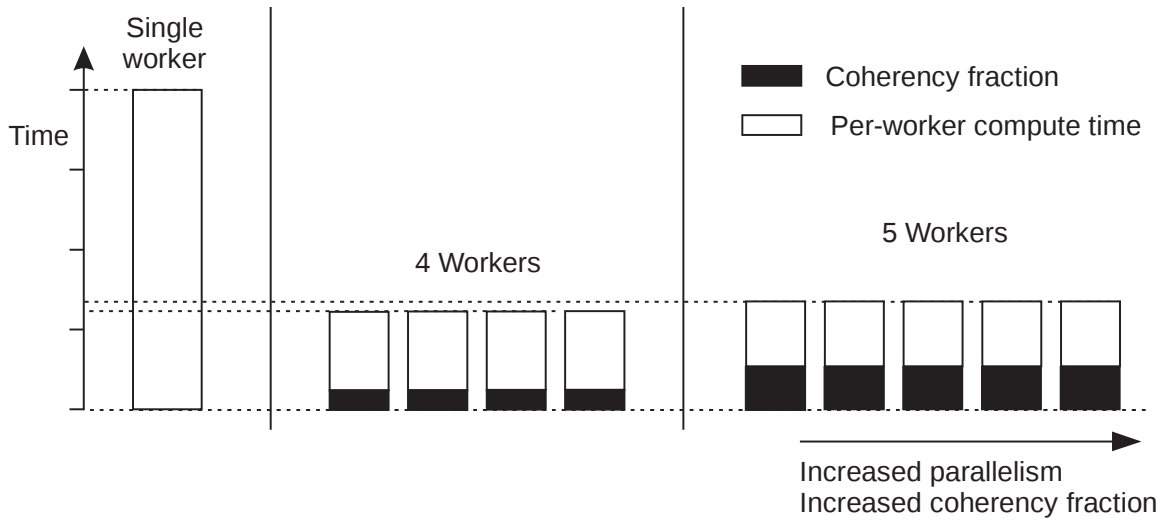

Figure 2.2: *Coherency limits performance gains and can cause degradation.*

one fourth, there is an additional time spent by each process (shown in black) that did not exist in the sequential execution. This is the time each process spends with data synchronization. With increasing parallelism, the synchronization time grows faster and, after a certain level of parallelism, it causes an increment in execution time.

According to the USL model, the relation between performance and parallelism is governed by the following equation, whose variables are described in Table 2.1. We can see this as an extension of the Amdahl's Law (Eq. 2.4).

$$C(n) = \frac{n}{1 + a(n-1) + nb(n-1)}, \tag{2.5}$$

Figure 2.3 shows how contention and coherency limit performance gains. As it can be seen, in the presence of coherency (blue dashed curve), there is a performance peak followed by degradation. In such cases, the number of parallel processes that provide maximum throughput may be calculated using Eq. 2.6, obtained by finding the value of n whose gradient is zero in the derivative of Eq. 2.5 [Gun07].

$$n_{max} = \left\lfloor \sqrt{\frac{1-a}{b}} \right\rfloor \tag{2.6}$$

As an example, with 0.2 of contention and 0.003 of coherency, we have that the maximum

Table 2.1: *Variables definig the USL equation*

Parameter	Description
C	Capacity. This is the relative capacity gain in comparison with a sequential execution. Equivalent to Equation 2.1
n	The number of parallel processing units used by the system
a	Contention factor. This is the ratio of the time spent in intrinsically sequential portions of the execution over the totally sequential execution time
b	Coherency factor. This is the ratio of the time spent in each data synchronization over the totally sequential execution time

throughput is obtained with (n_{max}) 16 parallel processes, providing capacity of ($C(n_{max})$) 3.39, as shown in Fig. 2.3.

Besides the model, Gunther also described a simple procedure to estimate the contention and coherency factors of a given system [Gun07]. For that, one needs to run a series of load tests with the system using different numbers of processors n. We need to keep the ratio N/n constant along all the load tests, where N is the total number of processes executing in the System Under Test (SUT). From the load tests, we have to obtain the mean throughput delivered by the system for the different values of n, including $n = 1$. Hence, we can compute the relative capacity, obtained in each test, using Equation 2.1. Then, we can obtain the inverse efficiencies $n/C(n)$. With simple mathematical transformations, we can go from Equation 2.5 to Equation 2.7, which is quadratic in n.

$$\frac{n}{C(n)} = bn^2 + (a - b)n + 1 - a \tag{2.7}$$

Quadratic equations have the general form $y = c_1x^2 + c_2x + c_3$. We can apply a quadratic regression to obtain the values of the coefficients c_1, c_2, and c_3. Thus, the scalability coefficients can be obtained using the following equations:

$$b = c_1 \tag{2.8}$$

$$a = c_2 - c_1 \tag{2.9}$$

In the following section, we cover the common practices and some applications proposed to perform scalability evaluation of distributed systems.

2.6 Scalability evaluation

Scalability evaluation is the experimentation process of understanding how variations of the workload and aspects of the system environment affect the system's QoS. It is performed with experiments in which an application is submitted to a certain workload. During these experiments, aspects of the workload and/or the application environment change. Quality metrics, generally related to performance and resource utilization, are collected and analyzed. This procedure is, often, performed manually, but it can be laborious, error prone, and difficult to replicate.

Designing the experiment is essential. For so, it is crucial to state its goals and what constitutes the system, delineating its boundaries, because they can affect decisions regarding workload generation and metrics collection. The knowledge gained by the study of the experiments may require the analyst to go back and reconsider some decisions. The complete evaluation may, therefore, consist

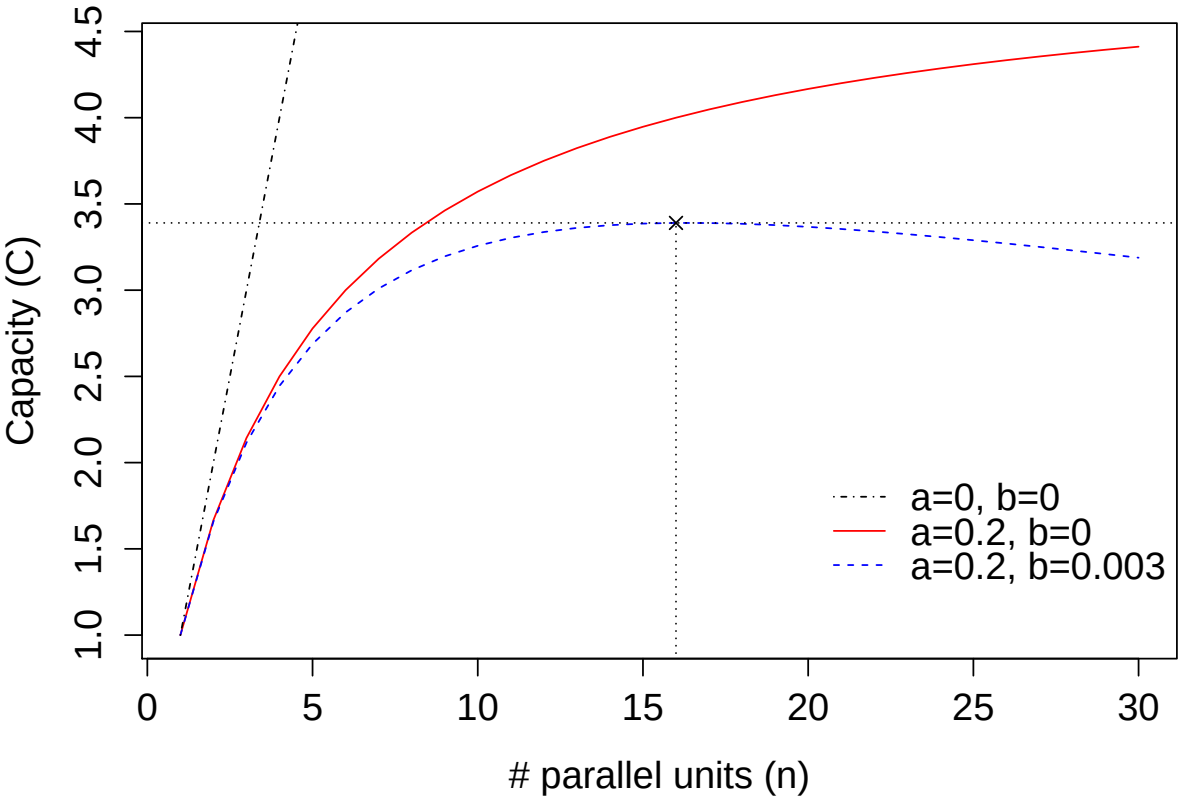

Figure 2.3: *The effects of contention and coherency on speedup.*

of multiple iterations through the steps [Jai91].

During the process of evaluating the scalability of a system, a series of experiments should be performed. Each experiment should follow these steps:

1. identify the parameters that define the complexity of the problem;
2. identify the parameters that influence the system capacity;
3. choose initial values and the pattern to vary these parameters;
4. select metrics to collect during the experiment;
5. define stopping criteria for the experiment;
6. select the workload to use in the experiments;
7. choose a technique/metric to evaluate the experiment;
8. run the experiment;
9. perform the analysis.

Automating this process has been a research concern and some tools have been proposed to accomplish that (see Section 3.1). Each framework has particularities concerning the kind of supported target systems, how they deal with architectural update and workload variation, and measurement collection.

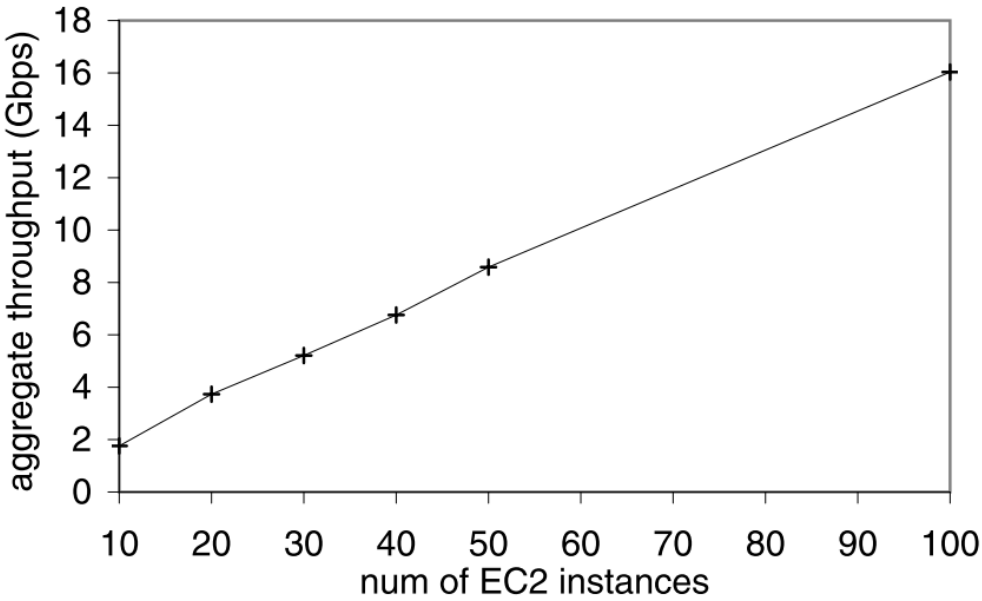

Figure 2.4: *Example of aggregate performance graph [LW09]*

Besides, the techniques commonly applied for analyzing scalability often lack precision. Some researchers propose numeric metrics to express scalability [CS06, JW00, GPBT11], as discussed in Section 2.5. They are similar in the sense that scalability is measured in terms of the ratio of metrics, considering performance and capacity, calculated in two different scales, and assuming scalability is linear – which is hardly reachable [Amd67]. Hence, metric outputs are greatly affected by the choices made in the experiment planning; higher parameter values related to the problem complexity and system capacity lead to higher deviation from linearity.

But charts that show how performance behaves during the experiments are the most commonly used means to analyze scalability. These charts usually present system aggregate performance [LLGZ13, LW09], speedup [RVG+11, CBNEB12] or degradation [CBNEB12, HLM+10, KBW12]. The first consists of analyzing the performance of the system (whose measurement are aggregated with some function such as average or percentile) when receiving different workloads, usually while resources are also scaled, as shown in Fig. 2.4; the second is about the relative performance gain when more infrastructural resources are added to the system, comparing to a fixed workload. Figure 2.5 shows speedup comparison of three experiments; and the third, also known as slowdown, concerns the performance loss for a given system configuration, when the workload increases, as shown in Fig. 2.6.

Different from the mentioned metrics, which provide a comparison between two experiments, the charts comprise any number of experiments in the analysis. Yet, the usual analysis is simply based on arbitrarily deciding whether the performance curve is close enough to the linear ideal to consider the system scalable or not, or for comparisons.

Thus, we see two common goals in scalability evaluations: identifying the most scalable system from a set of options; or verifying if a system is scalable. The former is certainly worth when we need to make a choice among different systems or to determine which update, among a set of considered possibilities, yields better scalability to a system. On the other hand, we identify some issues regarding the latter. All those metrics are relative: there is an ideal (almost unreachable) result and the analysts generally choose an arbitrary threshold to consider that a system is scalable. Besides, the selected parameter values affect the results. Therefore, the validity of the analysis is restricted.

Instead, the analysts should ask how to scale the system, that is, what is the relation between system capacity and performance. Due to non-linearity, this relation changes with size. Thus, the pursuit should be for a scalability function, instead of a single number. In this thesis, we use the

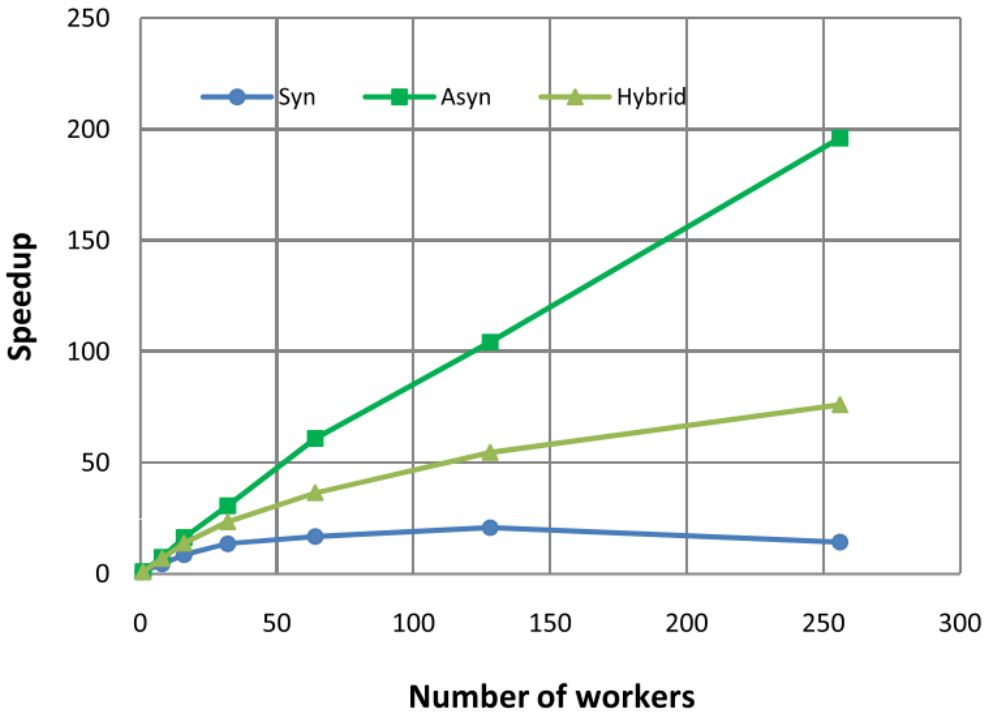

Figure 2.5: *Example of speedup graph [CBNEB12]*

USL (see Section 2.5.2) as the output of the scalability evaluations. This choice also guides other decisions regarding the experiment planning, as presented in Section 5.3.

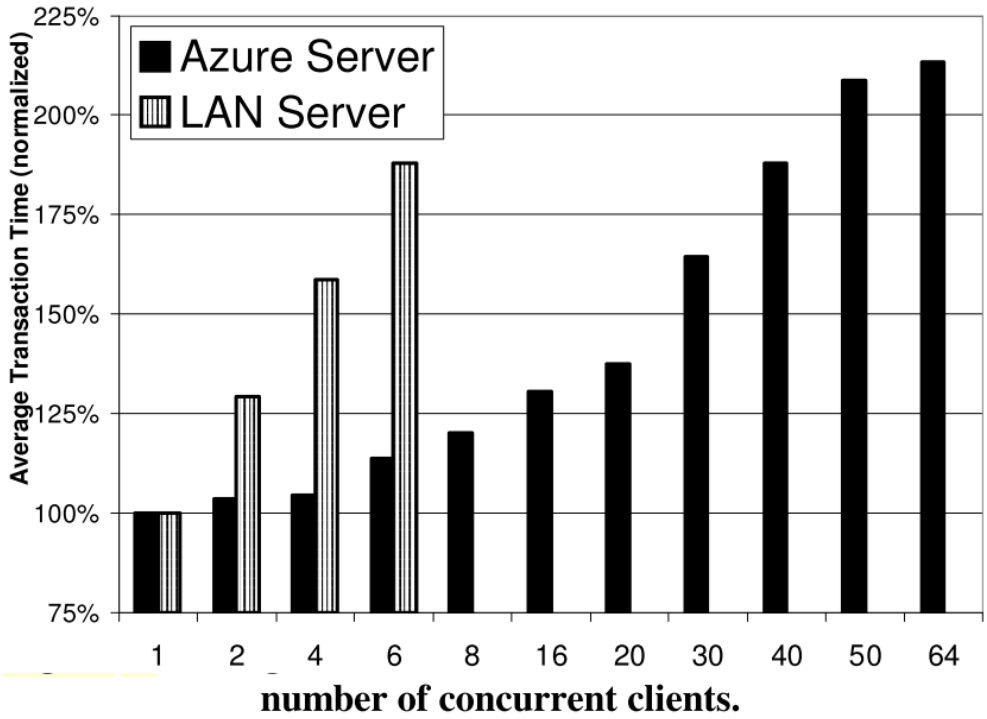

Figure 2.6: *Example of degradation graph [HLM+10]*

Chapter 3

Related Work

Cloud computing leveraged capacity planning [MA01] to real time reactions to workload fluctuation. Fully benefiting from the distinguishing cloud features that enable this shift demands automation. Commercial cloud solutions, such as AWS and Windows Azure, and tools, such as RightScale, generally offer means, either via a Web interface or API, to define auto-scaling rules based on provided metric counters. Their intent is to provide a ready-to-use tool that can be used for virtually any system. But there is not a systematic procedure to define such rules – analysts usually rely on their experience and feeling. In addition, rule based strategies are limited in terms of quick reaction and precision.

We consider that understanding how the resource availability affects system performance is crucial to the improvement of the strategies for auto-scaling. The nonlinearity, common to system scalability, affects the extent of the actions required in response to workload fluctuations at different load levels.

In this chapter, we analyze the research efforts in two main fields related to our research: tools and frameworks for automated scalability evaluation of distributed systems (Section 3.1) and strategies to improve the automated resource management in cloud based systems (Section 3.2).

3.1 Automated scalability evaluation

Even though most of the times we find practitioners manually testing and evaluating the scalability of systems [LLGZ13, ACP12] or using tools or scripts in part of the process [MDLX05, KBXL10, KM12a], there are efforts in automating scalability evaluation of distributed systems. This section analyzes tools and frameworks for automation of scalability evaluation. Their main characteristics are summarized in Table 3.1.

STAS (Scalability Testing and Analysis System) [CS06] is a system that provides scalability analysis of algorithms and systems. The first step, when using STAS, is to run a speed measurement module that collects computing power of the nodes. Then, a pre-analysis component takes the system source code and performs a workload analysis, based on user hints. After, node sets are constructed based on node computing power, doubling the computing power of the set at each iteration. STAS can run the tests, collecting and storing the execution time. The workload size is obtained from the previous analysis. Finally, the isospeed-e scalability metric is calculated, to verify whether the system is scalable. The metric considers the computing power of the nodes and the execution time and compares two configurations, giving their scalability ratio, where 1 means perfect scaling. STAS can be used to analyze the scalability of MPI programs.

Jogalekar and Woodside [JW98, JW00] present a scalability framework that is based on a strategy for scaling up or down a system, controlled by a scale factor that rules the value of a set of variables. They introduce a metric based on throughput, performance, and resource usage. As with the isospeed-e, this metric gives a scalability ratio relating two configurations.

ASTORIA [SAP11] is a framework for automated performance and scalability testing for Rich Internet Applications (RIA). This framework scales load by using virtual machines that simulate

user interaction with the system through a GUI-less browser and logs the response time of each performed action. Throughput is also calculated. It assumes that the provisioning of the testing resources is handled automatically. The interpretation of the collected metrics is left outside of the framework.

Almeida et al. [dASLTV08] presents a methodology and a framework for testing Peer-to-peer (P2P) applications combining functional and non-functional tests due to the volatility and scale variability of P2P. The tests are written in Java, similarly to jUnit tests, using annotations and assertions. The framework applies scalability testing by changing the scale of the application and validating their functionalities. However, each execution scenario must be specified, which can make the test specification cumbersome.

Klems et al. introduced a framework specific to test distributed databases deployed on clouds [KBW12]. It includes mechanisms for set up and decommission of cloud database service systems, to perform horizontal and vertical scaling, and some database specific resources, such as tuning configuration. It also allows deployment of distributed workload executors. This framework can plot speedup and scaleup graphs to support scalability analysis.

Expertus [JSM+12] is a code generation framework. It takes an experiment specification and generates resources to automate the testing process. Expertus has been designed to automate performance and scalability testing in cloud platforms. Experiments are specified in terms of application parameters, target cloud and test scenarios. These information are expressed in Extensible Markup Language (XML) and go through a sequence of Extensible Stylesheet Language Transformations (XSLT) transformations. Each transformation handles a different type (e.g. application and platform) or level (e.g. runtime and deployment) of dependency. The number of transformations vary for different experiments, applications, software stacks, operating systems, and clouds. The result is a set of scripts for deployment, configuration, execution, and data collection. Collected data is stored in a warehouse. The main inconvenience of this approach, concerning scalability experiments, is that one test scenario must be specified for each value of each change in workload or system architecture.

Cloud Crawler [CMA13] is a tool for performance evaluation of systems running on IaaS clouds. It is an engine that executes scenarios described in a declarative language. The Virtual Machines used for the tests must be previously created and it does not provide any support for analysis.

BaaSP [TDP+13] is a self-scalable benchmarking service. It can be used to identify the saturation level of the SUT. Saturation can be detected by response time threshold, error or alarm occurrence, or request throughput stagnation. The user defines load injectors, which generates request streams to the SUT and measures the respective response times. Resource consumption at the servers running the SUT is monitored by probes. The measurements are used in a decision process to vary the workload along the experiment in search for the saturation level. Load injectors can be distributed in a set of virtual machines. Those machines are monitored and, if saturated, a new VM is requested and the load injectors evenly distributed.

Scalar [HPJ14] is a framework that enables the simulation of multiple distributed simultaneous clients. It runs experiments comprised by a sequence of load tests. The user can define, for each test, the load levels and duration of the phases the load go though: warm-up, rump up, full load, rump down, and cool down. Measurements are collected at full load. After an experiment, Scalar aggregates the measurements collected by distributed instances, estimates throughput, and estimates a scalability curve based on the USL.

JMeter [1] is a popular load and performance testing tool compatible with a vast range of service types, including HTTP, HTTPS, SOAP, JMS, and others. It is developed by the Apache Foundation and runs test plans composed of thread groups, logic controllers, samplers, listeners, timers, assertions, and configuration elements. A diversity of implementations of these components is available. JMeter is open-source and can be extended with custom plug-ins. Thread groups controls the threads used to execute the test, simulating concurrent connections. The user can set the number of threads, how long JMeter must take to start all the threads (the interval between each thread start is constant), and the number of times to execute the test. Logic controllers and samplers must be

[1] http://jmeter.apache.org

under a thread group to drive how the test proceeds. Samplers define requests to send to the target application. Logic controllers customize how JMeter decides when to send requests. Listeners store and/or show information collected by JMeter during a test. Assertions allow to verify if what the application returns is correct. Configuration elements modify sampler's behavior. Timers define an interval between requests; by default, there is no pause between requests. Pre- and post-processors can be set to execute before and after the samplers. JMeter provides a proxy that can record traffic and create a test case. A server mode enables distributed testing: one master system controls the test, sending commands to slave systems; slaves send requests to the target system and the test results to the master. Generally, performance is analyzed in terms of response time and the approximate throughput (request/second). Concerning scalability evaluation, JMeter does not deal with reconfiguring the target system architecture.

JStress [2] is performance harness. Performance tests are created extending its `TestCase` class, overwriting `initTask`, `runTask`, and `destroyTask`. Methods `startTimer` and `StopTimer` must be called inside `runTask` to measure latency. `JStress` provides five strategies to fire the requests: simultaneously, constant interval, linearly increasing/decreasing interval, random interval, and sequentially (without interval). The runtime environment can be configured using a property file. No graphical interface is provided. Metrics are printed on screen and saved to a file. `JStress` package includes an R script to plot a chart with request rate, response rate, and concurrent requests (how many requests were in the server at that time).

In general, the tools and frameworks proposed to support automation of scalability evaluation are limited to a small scope of possibilities in many aspects. For instance, STAS [CS06] supports MPI, Scalar [HPJ14] supports HTTP, and Klems' framework [KBW12] supports relational databases. Many of the tools do not provide a means to scale the system under test, others require that all the scenarios be specified, one by one. Tools like JMeter and JStress only increase the workload linearly, on the other hand, ASTORIA [SAP11] and BaaSP [TDP+13] use their own heuristics to vary the workload with a specific goal. We propose, is this research, a flexible framework that encapsulates the skeleton of scalability experiments and provides a set of extension points by which the user can customize the experiments according to the system under test and the goals of the experiments.

3.2 Automated resource management

In this section, we focus on software components that deal with the tasks of requesting and releasing resources for an elastic system. Previous research initiatives focused on improving the elasticity strategies commonly provided by commercial solutions. On this matter, Lim et al. [LBCP09] propose proportional thresholds that change based on the number of instances serving the system, while Jamshidi et al. [JAP14] integrates time-series and fuzzy logic, enabling the use of qualitative imprecise thresholds, such as "high" and "low" in rule definitions. Mamani et al. [MPS+15] worked with the cumulative difference between current utilization and pre-established limits to improve the reaction to workload variation. ElasticRMI [Jay13] is a framework to create elastic distributed applications based on Java RMI, which enables developers to programmatically define rules combining resource utilization and application-specific metrics. Chapman et al. [CEM10] proposed a language to manage elasticity which can be used to describe service requirements and to provide rules on how to respond to performance and workload variation.

Salah et al. [SEB12] show that rules based on utilization are not optimal as they can lead to higher allocation than what is strictly necessary to handle a given workload. They propose an strategy based on Markovian models that make decisions based on estimated mean service time. Aljohani et al. [AHA13] model a server as an M/M/n queue (a queue with exponential arrival rate and service time distributions, and n service units), where n is the pool size, and uses thresholds based on the number of requests in the system.

A different strategy to improve the resource management is learning from history on-the-fly. Vasić et al. [VNM+12] experimented with numerous off-the-shelf machine-learning techniques, re-

[2] http://jstress.manamplified.org

porting good results with Bayesian models and decision trees. Gong et al. [GGW10] use signal processing techniques to find patterns in workload and resource usage. In both cases, a traditional strategy is used and improved from the analysis of historical workload and resource usage variations.

Instead of focusing on SLA guarantees in a single service, some researchers work on the resource management of a composition. A negotiation process among the services take place to decide which service, in the composition, better responds to a change in its pool size. The variations happen in unitary steps. Harbaoui and colleagues [HDV08, HSDV10, SDH+10] propose to run load tests to identify the appropriate queuing model for each service. Dejun et al. [DPC10] assumes a single queuing model for all the services, M/M/n/PS (a queue with exponential arrival rate and service time distributions, n service units, and process sharing service policy), and update the service time in a feedback loop, adjusting to variations that are not captured by the model. They consider the possibility of using cache instances. In this case, each service calculates whether it provides better performance by changing the cache or the pool size. They also consider that similar virtual machines can provide different performance in terms of CPU and I/O [DPC11]. In this case, they profile each service to identify those which are CPU or I/O intensive, and profile every VM and consider what it can provide to the services. Mencagli et al. [MVV13] uses a model to estimate response times of each system component, based on the response time of the services it depends on, and makes a number of iterations, when they exchange those estimations, to find the optimal configuration.

As we can see, many research initiatives also consider fixed size updates of the pool size. The proposals based on runtime learning provide dynamic updates, but they cannot predict the demand for load levels higher than what has already been experienced, relying on utilization based strategy in this case. Our work is distinguishing in this aspect. We propose the application of scalability models to be able to predict the resource demand even for workloads still not experienced. Our intention is, specially, to improve the response to drastic variations of the workload.

Table 3.1: *Characteristics of scalability testing frameworks*

Framework	Supported systems	Horizontal scalability	Vertical scalability	Workload variation	Analysis
STAS	MPI	doubling computer power	no	estimated from user hints	isospeed-e metric
Jogalekar's	not specified	according to the scale factor	according to the scale factor	relative call frequencies between entries	Jogalekar's metric
ASTORIA	RIA	no	no	tries to reach the target number of virtual users as fast as possible	none
Almeida's	P2P	defining scenarios with different number of peers	no	no	assertions
Klems'	databases distributed on clouds	defining scenarios with different number of servers	defining scenarios with different server types	workload models divided into multiple phases	speedup and scale-up
Expertus	system deployed on IaaS	defining scenarios with different number of servers	defining scenarios with different server types	test scenarios specification	none
Cloud Crawler	system deployed on IaaS	list of number of replicas	list of server types	list of number of simultaneous users	none
BaaSP	IaaS based	no	no	autonomic	saturation level
Scalar	HTTP, REST	no	no	manually defined per test	USL
JMeter	SOAP, HTTP, JMS, JDBC, and others	no	no	constant increasing the number of simultaneous threads	listeners
JStress	HTTP, REST	no	no	linear	request and response rate

Chapter 4

The Scalability Explorer

The Scalability Explorer is a Java framework conceived to be flexible and contains facilities to support the definition of automated scalability tests. In this chapter, we present its structure, how the elements interconnect in execution, and how to create scalability experiments using it. It is available under the Mozilla Public Licence version 2.0 at https://github.com/choreos/choreos_v-v/tree/master/scalability_explorer.

4.1 Architecture

The framework provides a template to create experiments for evaluating scalability of services with hotspots to customize the experiments according with the SUT characteristics and evaluation goals. Its core is mainly composed of interfaces and abstract classes integrated as shown in Figure 4.1. Implementations for each component of this architecture are included in the framework, hence a scalability test can be built by plugging a set of implementations and writing the code specific to the interaction with a particular system that the user wishes to analyze. Notwithstanding, the user can create new customized implementations for framework interfaces to adapt the experiments to his needs. In the remainder of this section, we describe the components of the framework.

4.1.1 Experiment

An experiment is a sequence of iterations where, for each iteration, scalability parameters related to workload and/or the system architecture vary. In each iteration, a number of requests is performed to the evaluated system. Performance metrics are collected during the execution and, after the iterations, an analysis is performed. The `Experiment` class contains the template for this procedure and the actions performed when it is executed are defined by objects implementing the interfaces detailed bellow or overriding methods defined in this class. When implementing an `Experiment`, the user can override the methods `beforeExperiment` and `afterExperiment`, which are called once, for general setup and finalization, before and after all the experiment iterations.

Experiments are executed by invoking the method `run()`, which receives a label, to identify the experiment in the analysis, and optional logical parameters: `analyse` determines whether the analysis should be executed after the given execution, `store` determines if the data collected from this execution should be stored for analysis. Both parameters default to `true`. The former can be set to `false` to enable running multiple experiments. Using it, one can run different experiments, with distinct parameters or strategies, and compare both in the same analysis. The latter parameter can be used to run a warm-up, starting the execution of a low workload and increasing it to the initial level actually used in the experiment. It avoids over stressing the SUT with an instant high number of requests, and the measurements in this phase can be discarded.

4.1.2 ExperimentStrategy

`ExperimentStrategy` defines the behavior at an specific point in the execution of an `Experiment`.

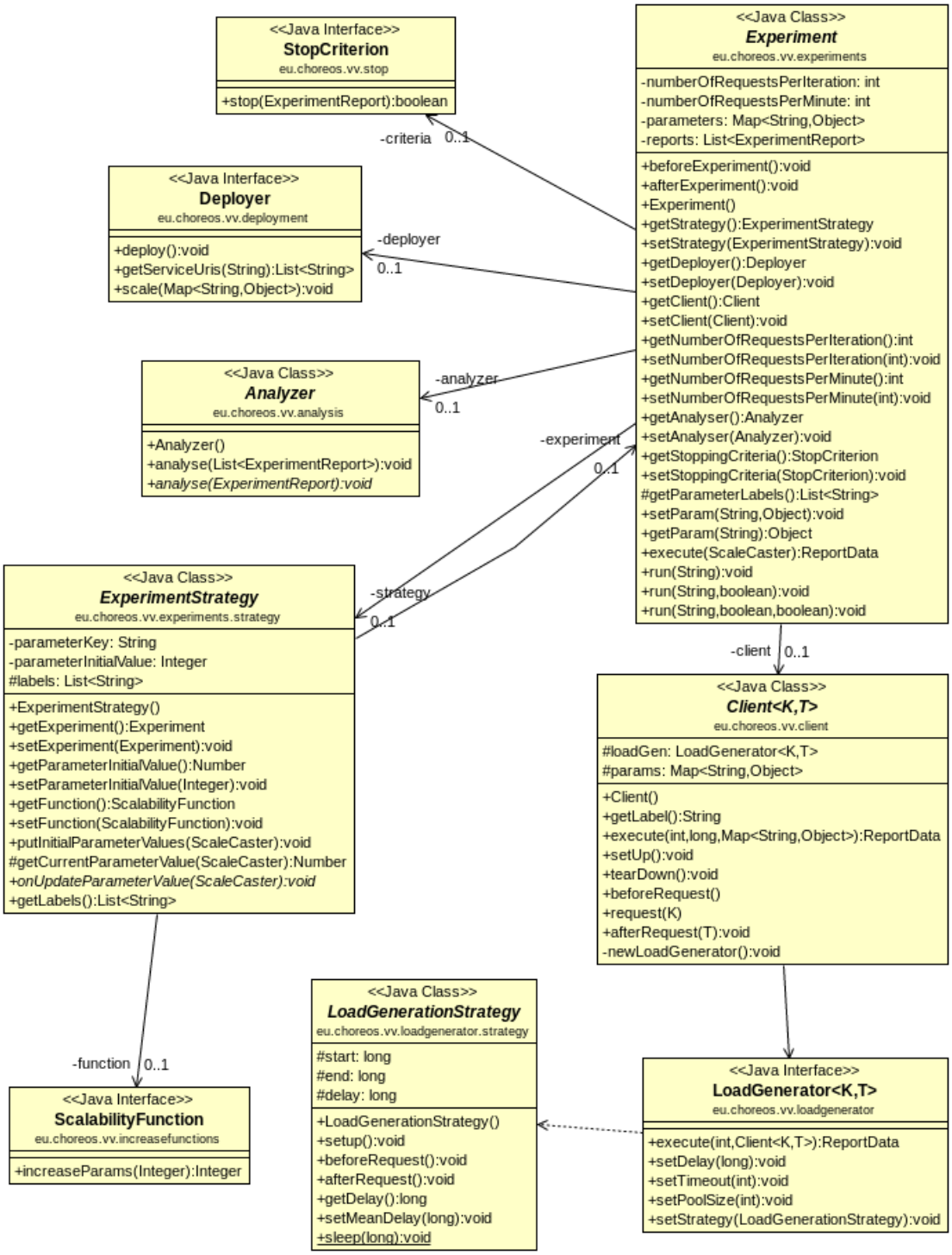

Figure 4.1: *Scalability Explorer Core Structure*

An strategy deals with one parameter whose value is scaled along the experiment iterations. The initial value and the function used to calculate the next values used for the given parameter are set using the methods `setParameterInitialValue` and `setFunction` respectively. Implementations of `ExperimentStrategy` must override the method `onUpdateParameterValue`, which is called when the respective parameter value is updated, defining what is done regarding that parameter.

Three implementations are provided:

- `ParameterScaling`: is a convenient way to define a generic parameter whose value is scaled along experiment iteration and must be retrieved in the implementation of `Experiment` and used as wished;
- `WorkloadScaling`: creates a parameter that is specifically used to set the request rate used in each iteration;
- `ComposedStrategy`: enables the aggregation of a set of strategies in a single object.

The updates of the request rate, when using `WorkloadScaling`, are transparent to the user. However, as the way to perform architectural modifications to the system under test is specific to the system and platform where it is deployed, the Scalability Explorer only deals with updating the parameters' values, as specified with the scalability functions. Those values, then, can be retrieved by the `Experiment`, `Deployer`, and `Client` for adequate use.

4.1.3 Client

The `Client` interface can be used to simulate a client that sends requests to the SUT.

When implementing a `Client`, the user can override the following methods to define the interaction with the system:

- `request`: the user must override this method to perform the requests used to evaluate the system. The framework considers the duration of this method's execution as the response time of the request;
- `beforeRequest/afterRequest`: these are methods called before/after every request, but not considered in the response time measurement. When overriding these methods, the user must consider how they can affect the request rate reproduced by the load generator;
- `setUp/tearDown`: methods called before/after every iteration, when the scaling parameters are updated. When `setUp` is called the scaled parameters' values have already been updated to use in the coming iteration;
- `execute`: this is the method called by an `Experiment` to start the client execution on every experiment iteration. This method returns a `ReportData`, containing the collected measurements.

`Client` is implemented by the abstract class `BaseClient`, which uses a `LoadGenerator` to define the pattern at which it calls the SUT. The `MultiClient` extends `BaseClient`. It provides the `createClients` method to create a given number of clients. When executed, a `MultiClient` executes all the created clients and merges all their results into a single `ReportData`.

4.1.4 StopCriterion

The `StopCriterion` interface defines the method `stop()` to decide whether an experiment finishes. There are two implementations included. `IterationsStop` is used to define a fixed number of iterations for the experiment. `MeasurementStop` is used to define a limit to a performance metric. When this limit is crossed, the experiment ends.

4.1.5 ScalabilityFunction

A `ScalabilityFunction` defines how parameters must vary from one iteration to the next one. A `ScalabilityFunction` implements the method `increaseParam()`, which must receive the parameter's current value and return the next value in the sequence defined by the function. The current implementations are `LinearIncrease` and `ExponentialIncrease`. Both implementations receive a numeric parameter in their constructor, which is used as the common `ratio`. Thus, after n calls to `increaseParam` of a given `ScalabilityFunction`, passing `initialValue` in the first call and the output of the ith call as input of the $(i+1)$th call, where $0 < i < n$, we obtain $initialValue + n * ratio$ in the former implementation, and $initialValue + n^{ratio}$ in the latter.

4.1.6 LoadGenerator

`LoadGenerator` is an interface for a component that sequentially triggers the requests performed during the experiments. The framework provides implementations in which the inter-request intervals are randomly picked from a probability distribution (we provide implementations for Gaussian, Poisson, Uniform, and Degenerated distributions), or an execution trace. The different distributions, as well as the trace-driven workload execution, are implemented as strategies – `LoadGenerationStrategy`. The generation of pseudo-random intervals that fit those distributions is supported by the Apache Commons Math library [1].

A `LoadGenerator` must implement setters for inter-request delay, request timeout, pool size, and strategy. Plus, the method `execute()`, which must trigger the requests to the SUT, by invoking `Client.request()`. Strategies are implemented by overriding the methods `setup()`, `beforeRequest()` and `afterRequest`.

`Client` uses the singleton `LoadGeneratorFactory` to create the `LoadGenerator` used in the experiment. The user can configure the factory regarding the class implementing `LoadGenerator` that will be used and the parameters to configure the instance. Provided implementations of `LoadGenerator` are `ParallelLoadGenerator` and `SequentialLoadGenerator`. In the former, the inter-request interval is measured from the moment a request is triggered, while in the latter it is measured from the moment the reply arrives.

4.1.7 Analyzer

The `Analyzer` is called at the end of the experiments to support examining the results of a scalability test. It processes the collected metrics and scalability parameter values, generating a meaningful output. An analyzer implements the method `analyze()` receiving a single `ExperimentReport` or a list of it. Currently, the Scalability Explorer provides the following analyzers:

- `AggregatePerformance`: aggregates the measurements made in each iteration in a single value using an `AggregationFunction` and plots a chart with the aggregated performance obtained in each iteration using a `ChartCreator`;
- `ANOVATest`: performs hypothesis test [CB02] to verify if the mean performance obtained in all iterations are equivalent (see Section 4.4);
- `SampleSizeEstimation`: estimates the minimum number of request that should be performed per iteration to make the `ANOVAtest` meaningful;
- `ComposedAnalysis`: allows the user to use more then one `Analyzer` in an experiment;
- `SaveToXML`: saves the data in XML format.

[1] commons.apache.org/proper/commons-math

4.1.8 AggregationFunction

`AggregationFunction` is used to summarize a collection of values in a single number. The framework provides classes to calculate arithmetic mean or percentile. It requires the method `aggregate`. These functions are intended to be used in the analysis or to define stop criterion based on the measurements.

4.1.9 ChartCreator

`ChartCreator` is an interface to classes that create graphic visualization of the experiment analysis. They are used by the `AggregatePerformance` analyzer. Implementations must override the methods `createPlotData()`, that produces the data for plotting, and `createChart()`, that creates the graphic.

The framework includes two implementations of `ChartCreator`: `PercentileChartCreator` and `MeanChartCreator`. The former creates a line plot with the specified percentile of the measurement, computed for each iteration. The latter produces an interval plot, where a solid lines shows the average of the measurements and a shade shows the respective standard deviation.

Both implementations use the class `LineChart` to build the graphs using JFreeChart library [2]. Figure 4.2 shows the structure under `ChartCreator`.

4.1.10 Deployer

`Deployer` is an interface for a component used to deploy services during the experiment. Using a `Deployer` is optional. `Deployer` defines two methods that are called by `Experiment`.

- `deploy()` is called at the beginning of the experiment to set the system up;
- `scale(params)` is called before each iteration, passing the architectural scalability parameter as argument.

Another method defined by this interface is `getServiceUris(String)`, which can be used to retrieve the URIs of a given service during the experiment.

The `EEDeployer` is an abstract implementation that encapsulates the interaction with the Enactment Engine [LLGK13, LMC+14], a middleware system that provides a platform for automation of the distributed deployment of Web service compositions in cloud computing environments. `EEDeployer` can be extended by overriding the `enactmentSpec()` and `scaleSpec(int)` methods to return the specification to be sent to the Enactment Engine when `enact()` and `scale(int)` are called, respectively.

4.1.11 ReportData

After each iteration of an scalability experiment, the `Client` produces a `DataReport`. It stores the start and end time of the iteration, the collected measurements and the values of the parameters scaled along the experiment. `DataReport` implements the method `merge` to enable composing a single `DataReport` with data collected from multiple sources. This method gives support to the implementation of experiments with multiple clients.

All the `ReportData` produced in an experiment are included in an `ExperimentReport`, which is a collection extended with extra parameter with information about the experiment: its identification label, the parameter's names, and measurement units. The `ExperimentReport` is used by the `Analyzer` to generate the experiment's output.

[2] http://www.jfree.org/jfreechart

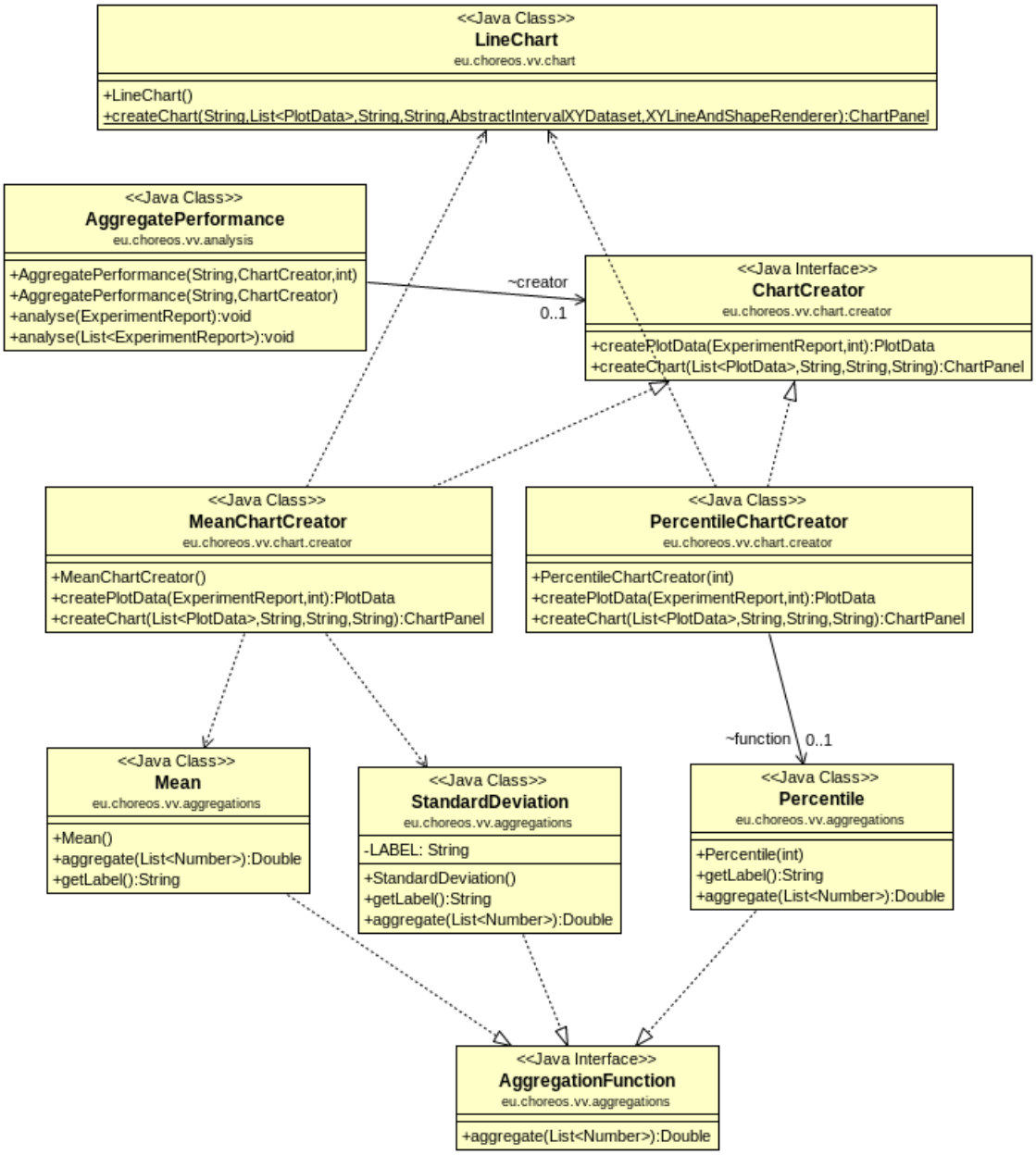

Figure 4.2: *Chart creation structure*

4.2 The mechanics

The sequence diagram in Fig. 4.3 shows hows the components of the Scalability Explorer interact to execute and experiment[3]. A scalability experiment starts when `Experiment.run()` is invoked. It, first, calls `beforeExperiment()` and, if a deployer is set, calls `Deployer.deploy()`.

Then, the experiment iterations have place. In these cycles, it calls `ExperimentStrategy.onUpdateParameterValue()`, calls `Deployer.scale` if a deployer is set, and starts the `Client`. The `Client` calls its `setUp()` method, creates and configures a `LoadGenerator`, and calls its `execute()` method.

The `LoadGenerator`, working according to its implementation and parameterization, starts a sequence of requests to the SUT. When each request is triggered, it calls `Client.beforeRequest()`, `Client.request()`, and `Client.afterRequest()`, in this order, measuring the time spent by the second one.

After all the request in an iteration are finished, the `Client` calls its `tearDown` and returns a `ReportData` to the `Experiment`, which stores it in a `ExperimentReport`, updates the parameters defined by the `ExperimentStrategy` and starts a new iteration if the `StopCriterion` is not satisfied.

After all the iterations are finished (when the `StopCriterion` is satisfied), the `Experiment` calls `afterExperiment()`, stores the `ExperimentReport` if set to do so, and calls the `Analyzer` if set to do so.

4.3 Writing scalability tests

An experiment is created by implementing an `Experiment` and a `Client`. They provide methods that, when overridden, define how the experiment interacts with the SUT. The `Client.request()` method must be overridden to perform the communication with the system; it defines what is done when the `LoadGenerator` triggers a request. Scalability Explorer is agnostic regarding the SUT technology and interface. The user must implement the communication using the appropriate libraries and protocols. Thus, the framework gains in applicability. The drawback is extra work demanded from the user in comparison with other tools that are technology specific. Implementing the other abstract methods defined by `Experiment` and `Client` is not mandatory, but they can be useful for system configuration and experiment setup, for instance.

The method `beforeExperiment()` is the recommended place to set the remaining parameters of the experiment. Listing 4.1 shows one such example. In that, we create two instances of `ExperimentStrategy`: the first linearly increases the number of requests per second triggered by the load generator, starting with 100 requests per seconds and adding 100 more to it as each iteration; the second defines a parameter named as `numberOfInstances` whose value starts at 1 and grows exponentially. Both strategies are composed in a single object and set to the experiment. Additionally, the number of request per iteration is set to 100. The experiment is defined to stop when the mean response time in one iteration is over 300 milliseconds and, at the end of the experiment, a chart with the mean response time in each iteration will be plotted.

```
1 @Override
2 public void beforeExperiment() {
3   ExperimentStrategy estrategy = new WorkloadScaling();
4   estrategy.setFunction(new LinearIncrease(100));
5   estrategy.setParameterInitialValue(100);
6
7   ExperimentStrategy cstrategy = new ParameterScaling("numberOfInstances");
8   cstrategy.setFunction(new ExponentialIncrease(2));
9   cstrategy.setParameterInitialValue(1);
10
11  ExperimentStrategy strategy = new ComposedStrategy(estrategy, cstrategy);
12
```

[3]The diagram was produced with www.websequencediagrams.com

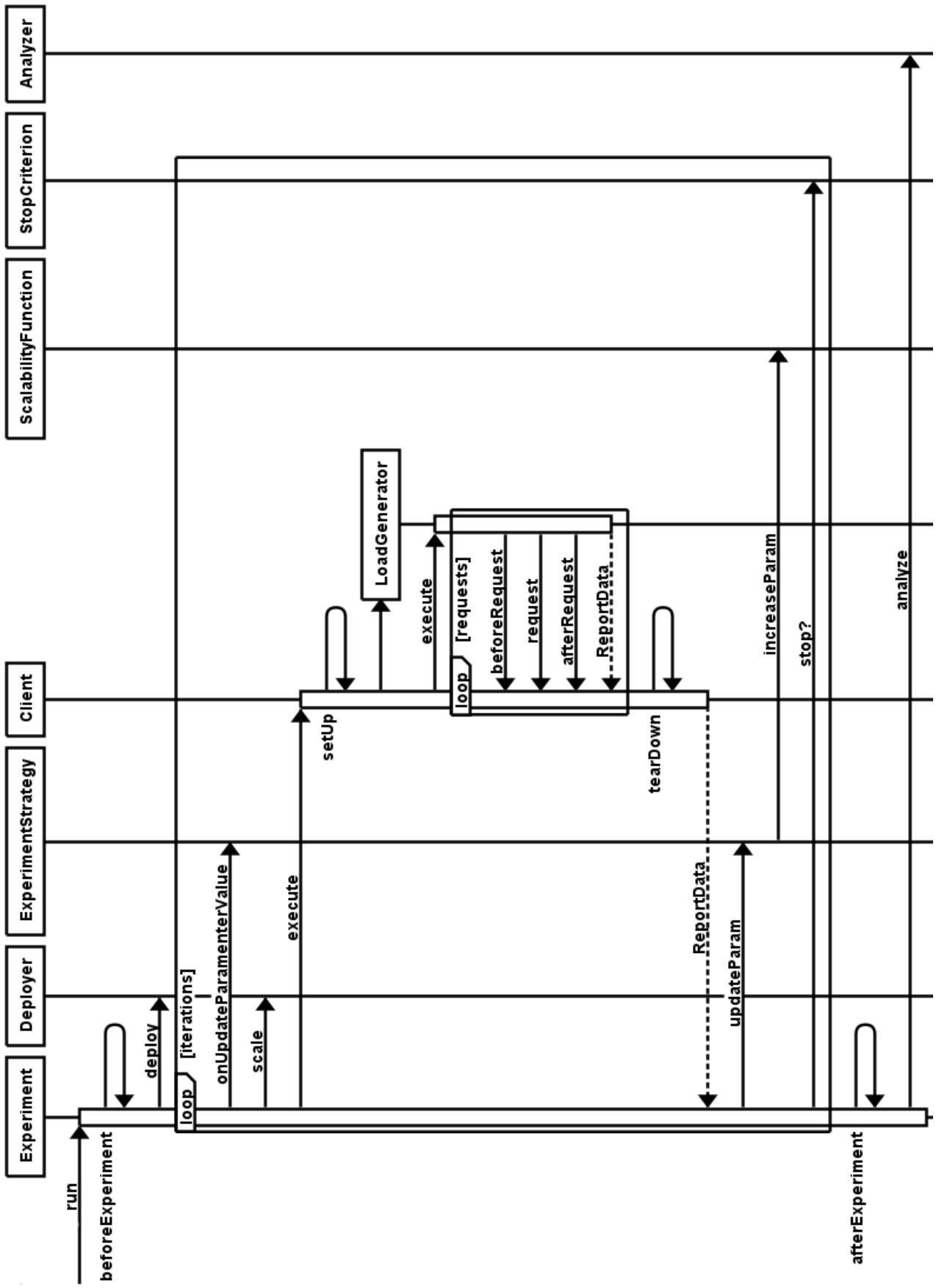

Figure 4.3: *Scalability Explorer's simplified mechanics*

```
13    this.setStrategy(strategy);
14    this.setNumberOfRequestsPerIteration(100);
15    this.setStoppingCriteria(new MeasurementStop(300.0, new Mean()));
16    this.setAnalyser(new AggregatePerformance("Scalability Chart", new
          MeanChartCreator(), 1));
17 }
```

Listing 4.1: *Overriding `beforeExperiment()` to configure an experiment*

The `Client` creates a new load generator in the beginning of each iteration, using a factory. The factory is a singleton and can be configured in the test implementation. Listing 4.2 shows an example in which general characteristics of the load generator are set in the experiment initialization and an `ExperimentStrategy` is defined to scale the load generator's thread pool size in each iteration, to make it able to deal with increased request rates.

```
1 public class MyExperiment extends Experiment {
2
3   @Override
4   public void beforeExperiment() {
5     LoadGenerationStrategy strategy = new PoissonLoad();
6     LoadGeneratorFactory factory = LoadGeneratorFactory.getInstance();
7     factory.setStrategy(strategy);
8     factory.setTimeout(120);
9     factory.setLoadGeneratorClass(ParallelLoadGenerator.class);
10
11    ExperimentStrategy cstrategy = new ParameterScaling("threadPool");
12    cstrategy.setFunction(new LinearIncrease(100));
13    cstrategy.setParameterInitialValue(100);
14
15    this.setClient(new MyClient());
16
17    //...
18  }
19
20  //...
21 }
22
23 class MyClient extends Client {
24
25   @override
26   public void setUp() {
27     LoadGeneratorFactory.getInstance().setPoolSize(params.get("threadPool"));
28   }
29   //...
30 }
```

Listing 4.2: *Configuring the load generation*

We identified two patterns of load generation in load tests. One is when the workload follows a request rate that can be ruled by a probability distribution or reproduce a trace. In these experiments, the load in the SUT may vary along the execution. Using the Scalability Explorer, we can simply configure the load generator in accordance with the planned workload to see it reproduced in the experiment.

In other cases, however, the tester wishes to set a constant load level at the SUT. To provide this possibility, we included an extension of `Client` that enables the execution of multiple parallel clients. In combination with `SequentialLoadGenerator`, we can create a load test of this kind. We can see an example of an experiment with this kind of configuration in Listing 4.3. The `NullStrategy`, used in the example, uses no time delay between requests. In conjunction with the `SequentinalLoadGenerator`, it makes a request being triggered just after the reply to the previous request arrives. This kind of experiment requires one implementation of `Client` to interact with the SUT, and another of `MultiClient` to handle the multiple clients.

```
1 public class MyExperiment extends Experiment {
2
3   @Override
4   public void beforeExperiment() {
5     LoadGeneratorFactory factory = LoadGeneratorFactory.getInstance();
6     factory.setStrategy(new NullStrategy());
7     factory.setLoadGeneratorClass(SequentialLoadGenerator.class);
8
9     ExperimentStrategy cstrategy = new ParameterScaling("numberOfClients");
10    cstrategy.setFunction(new LinearIncrease(1));
11    cstrategy.setParameterInitialValue(1);
12
13    this.setClient(new MyMultiClient());
14
15    //...
16  }
17
18  //...
19 }
20
21 class MyMultiClient extends MultiClient<MyClient> {
22   @override
23   public void setUp() {
24     this.createClients(MyClient.class, params.get("numberOfClients"));
25   }
26   //...
27 }
28
29
30 class MyClient extends Client {
31   //...
32 }
```

Listing 4.3: *MultiClient experiment*

The Scalability Explorer architecture supports the creation of scalability tests with distributed workload generation. The implementation of such tests is similar to what we did with the `MultiClient`. First, the user needs to create an implementation of `Client` that publishes the `execute` operation for remote invocation. Listings 4.4 and 4.5 show examples of interfaces to create clients accessible through RMI and SOAP, respectively.

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 import eu.choreos.vv.client.Client;
4
5 public interface RMIClient<K,T> extends Client<K,T>, Remote {
6
7   T request(K param) throws RemoteException;
8
9 }
```

Listing 4.4: *RMI client interface*

Using distributed clients also demands another implementation that works as a coordinator, similarly to `MultiClient`, triggering the remote executions and merging their return values. As an example, Listing 4.6 shows an extension of `MultiClient` that uses RMI clients. For so, it provides an implementation of the method `createClients` with an additional parameter which receives the list of URIs from where the remote clients are available. Those URIs could be provided by a `Deployer` through `getServiceUris`.

```
1 import javax.jws.WebService;
2 import javax.jws.WebMethod;
3 import eu.choreos.vv.client.Client;
4
5 @WebService
6 public interface SOAPClient<K,T> extends Client<K,T> {
7
8   @WebMethod
9   T request(K param) throws Exception;
10
11 }
```

Listing 4.5: *Web Service client interface*

```
1 public class RMIMultiClient<T extends RMIClient<A,B>, A,B> extends MultiClient<T
      , A, B> {
2
3   protected void createClients(Class<T> cls, long qtd, List<String> remotes)
4       throws InstantiationException, IllegalAccessException, RemoteException,
            NotBoundException {
5     clients = new ArrayList<T>();
6     int n = remotes.size();
7     int r = 0;
8     String name = "Client";
9     for (int i = 0; i < qtd; i++) {
10      Registry registry = LocateRegistry.getRegistry(remotes.get(r));
11      r = (r+1) % n;
12      T client = (T) registry.lookup(name);
13      clients.add(client);
14    }
15  }
16
17 }
```

Listing 4.6: *RMI clients coordinator*

4.4 Evaluating scalability

As seen in Section 4.1.7, the Scalability Explorer provides a hotspot to include a component to analyze the experiments, and some implementations are included. The `AggregatePerformance` allows graphic visualization of metrics, which is a common approach to analyze scalability. However, this solution lets the interpretation of the results entirely to the user. Besides, as discussed in Section 2.5.1, common metrics usually assume linear scalability.

In this section, we use an example to show the limitation of the assumption for linearity and propose the application of the ANOVA [CB02] – implemented by `ANOVAtest`. This is a statistical technique to verify the hypothesis that several populations show no difference in their mean values. We apply this test to verify whether the mean response time in all the experiment iterations are equivalent. Another differential of this analysis is that it supports more than two samples.

We illustrate the issue with a SOAP-based distributed matrices multiplication system example[4] deployed on the Amazon EC2. The system consists of a coordinator and a set of multiplication services. The coordinator splits one of the matrices across the services and the other is completely sent to each multiplication service. Those services, in their turn, return the multiplication of their parts to be merged by the coordinator. We executed experiments consisting of a sequence of requests to multiply two square matrices and measured the response time. Along the experiment iterations, we scaled the matrices size and the number of multiplication services. All the experiments were executed in five iterations, with one request per second, in Amazon m1.small instances, which have

[4] Available at https://github.com/pbmoura/matrix_multiplication

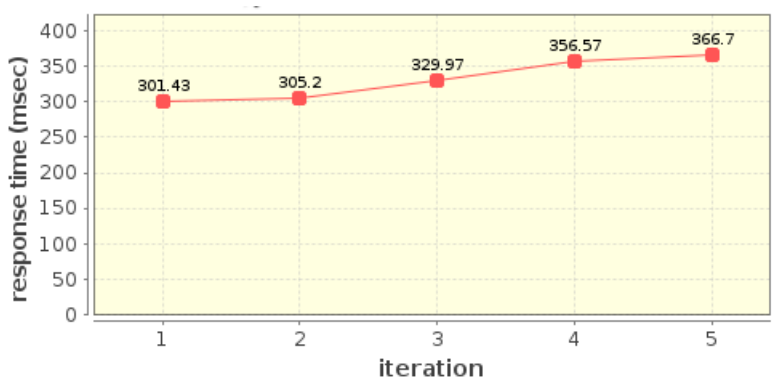

Figure 4.4: *Screenshot: test1 - Aggregate performance*

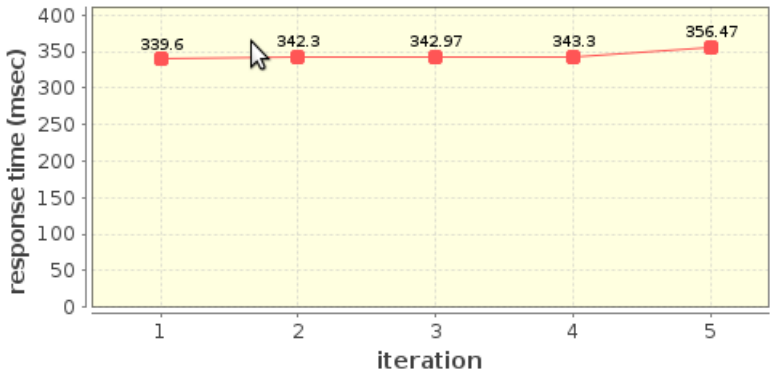

Figure 4.5: *Screenshot: test2 - Aggregate performance*

1.7GiB of memory and 1 EC2 Compute Unit power (equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor).

As we are dealing with two dimensional matrices, a linear growth in their dimension results in a quadratic growth in the input size. Hence, we ran a test linearly scaling the matrices size (test1). For so, while the number of nodes was linearly increased from 1 to 5 (by 1 per iteration), the matrices dimensions were increased in a lower rate so that its square (the matrix size) was linearly increased. The aggregate performance of this test can be seen in Figure 4.4. With this chart we could consider that, although not constant, the variation in performance was acceptable, but we aim to avoid this kind of subjective analysis. Instead, we also submitted the measured performances to ANOVA analyzer, which considered the performances of each iteration not equivalent.

The issue is that even though the data size and computational power were growing in the same proportion, the computation complexity was not. A matrix multiplication is an operation of cubic complexity. On the other hand, the computation is split among a number of nodes. Considering this, another test (test2) was performed with the matrices dimension growing according to the function $dimension = \sqrt[3]{S * n}$ where S was fixed to 8,000,000, that is the number of operations executed in the multiplication of 200x200 matrices, and n is the number of nodes. The aggregate performance of these scalability tests is shown in Figure 4.5 and the performances were considered equivalent by an ANOVA test. Table 4.1 shows the matrices dimensions used in each iteration of *test1* and *test2*.

As we can see with this example, the ANOVA test can be used to assess the scalability of a system without subjectivity. It requires, though, that we identify what is the relation between the input and the performance. In this example, we used the computational complexity of the implemented algorithm. Another possibility is to estimate an empirical scalability function, as we describe in Section 5.3.

Table 4.1: *Matrices dimensions*

iteration	test1	test2
1	200	200
2	283	252
3	346	288
4	400	317
5	447	341

4.5 Final considerations

Tools and frameworks for scalability evaluation of distributed systems (Section 3.1) are usually constrained to specific kinds of experiments, imposing limitations on aspects of the experiments, such as how to vary the workload, when to stop the experiment, what data is collected, or how it is analyzed. In this chapter we presented our contributions in the form the Scalability Explorer: a software framework for the definition and execution of scalability tests of distributed systems. The framework defines hotspots relative to common elements used in scalability tests and provides a few implementations. But it can be extended with new implementations that meets user's needs.

Additionally, we propose the application of ANOVA test for scalability evaluation. This statistical tool can be used to verify if the performance obtained in all iterations of a scalability tests are equivalent.

In the next chapter we present and validate our proposal to identify the relation between the problem size and the resource need and how to apply it to automate the resource management.

Chapter 5

Dynamic resource allocation based on scalability modeling

With our review on scalability evaluation of distributed systems, we faced with a scenario in which, generally, the outcomes of scalability evaluation are vague. Either we get metrics whose values we must compare to arbitrary baselines to state if the system is scalable or not, or we compare two systems or versions of the same system to identify the most scalable. Likewise, self-scalable systems are, usually, based on rules in which thresholds are also arbitrarily set.

Based on those observations, we decided for a strategy in which the objective of the scalability evaluation is to identify how the system scales. We are not interested in making the system more scalable, in terms of maximizing scalability metrics. Instead, we aim at modeling how system capacity changes in response to variations in the resource availability. Then, we can apply that outcome to improve system's elasticity.

To accomplish our goal, we need a model built from experimental observations and that extrapolate the dataset range used in the modeling. Thus, we can estimate what is the scalability pattern that the system exhibits at large scales, even when it had never experienced such levels.

In this chapter, we describe our proposal in terms of the model we used, the runtime algorithm for automated resource management, and the procedure to estimate the model parameters for a given system. Then, we describe the experiments executed to validate the proposal and show the results.

5.1 The model

As we presented in Section 2.5.2, USL draws a function from the quantity of available processing units to the capacity they provide to the system. We are interested in identifying the amount of resources needed to handle a given workload. Thus, we need to invert Equation 2.5. This results in two partial functions (Fig. 5.1): one whose range goes up to n_{max} (red solid curve), and the other, from that on (blue dashed curve). We are interested only in the first portion, which provides values for the number of resources up to the system's maximum capacity, which is modeled by Equation 5.1.

$$n(C) = \frac{b - a + 1/C - \sqrt{(a - b - 1/C)^2 - 4b(1 - a)}}{2b} \tag{5.1}$$

In order to handle varying workloads, we would like to modify a given system to keep throughput in equilibrium with the arrival rate. Otherwise, the system may become overloaded and, if the workload persists, the request queue may grow indefinitely, affecting the system's stability and performance. Since throughput capacity is a function of the number of service instances that are available, we use a control process [HDPT04] to periodically monitor the load balancer and use the equation previously discussed (5.1) to estimate the adequate number of instances required to

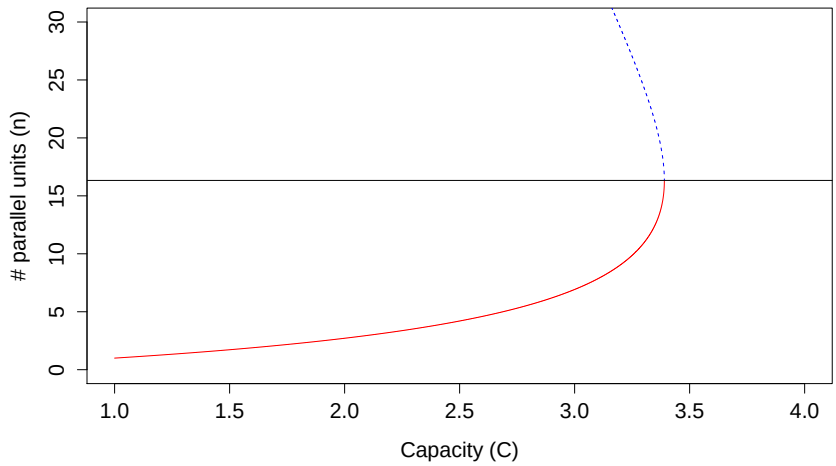

Figure 5.1: *Inversion of USL function*

handle the current load. The goal of this control process is to offer the best performance possible. The procedure is described in the next section.

5.2 Runtime algorithm

In this section we introduce the algorithm executed by the control process. It applies the inverted USL (Eq. 5.1) to dynamically adapt the resource availability, in response to the workload fluctuations, providing the best performance, while limiting costs. Resource management is guided by the algorithm in Fig. 5.2, which is periodically executed. The terms in monospaced (Table 5.1)font are either received as initialization parameter or retrieved from the monitor.

The controller relies on a monitor, which must keep track of three runtime metrics: the queue size (all current requests for the service), the number of arrivals (new requests) in the recent past, and the current pool size. The monitor receives as parameter the monitoring interval, which is the time it waits between two consecutive activations of the controller. The controller receives the model parameters, estimated as described in Section 5.3 (mean response time, maximum throughput, and contention and coherency factors).

Lines 1 and 2 of the algorithm compute the mean arrival rate and estimate the required capacity to handle such workload. If that capacity is higher than what USL estimates as the maximum achievable, the pool size is set to n_{max}; otherwise it is estimated using Equation 5.1, whose result is rounded up because the pool size must be an integer and rounding down might cause under-provisioning (lines 3 to 7). Then, it computes the difference between the estimated and the current pool sizes, to update it accordingly, at line 8. If it is necessary to increase the pool size, the required number of instances is promptly requested, in line 13.

We analyzed different strategies to deal with two of the parameters in the algorithm. Regarding `estimateInterval`, we consider to use an interval equals to the monitoring interval and to use only the last 20% of the monitoring interval to estimate the arrival rate. The option for choosing a shorter estimate interval aims at obtaining more accurate estimate, discarding variations that might have happened earlier. Regarding `availablePoolSize`, we consider to use the total number of nodes in the pool size and to compute what is the available portion of the pool. In the second case, the availability is computed in the following way: in the profiling phase (Section 5.3), we estimate `nodeCapacity` which is, on average, the number of simultaneous requests a node can process. The positive difference between this capacity and the current load in a node is its available capacity. For instance, if a node has capacity 5, when it is processing 2 requests its available capacity is 3. Then, the service availability is the sum of the positive available capacities of all the nodes in the pool.

Table 5.1: *Parameter used by the control process to estimate resource demand*

Parameter	Description
`arrivals`	Number of requests received by the service in the interval used to estimate arrival rate. Measured by the monitor.
`estimateInterval`	Duration of the time interval used to estimate arrival rate. Defined by the user.
`supportedArrivalRate`	Maximum arrival rate supported by the service when it is running in one node. Set by the user. This value is estimated in a procedure described in the next section.
`currentPoolSize`	Number of nodes available to the service. Maintained by the monitor.
`availablePoolSize`	Quantity of available nodes.
`queueSize`	Number of requests currently handled by the service, i. e., requests that arrived to the service and have not being replied yet. Measured by the monitor.
`nodeCapacity`	Estimate of the number of requests as instance can process in parallel. Obtained multiplying the `supportedArrivalRate` by the mean response time.

1: $arrivalRate \leftarrow$ `arrivals`/`estimateInterval`
2: $capacity \leftarrow arrivalRate$/`supportedArrivalRate`
3: **if** $capacity >= C(n_{max})$ **then**
4: $\quad estimatedPoolSize \leftarrow n_{max}$
5: **else**
6: $\quad estimatedPoolSize \leftarrow ceil(n(capacity))$
7: **end if**
8: $diff \leftarrow estimatedPoolSize -$ `availablePoolSize`
9: **if** $diff +$ `currentPoolSize` $> n_{max}$ **then**
10: $\quad diff \leftarrow n_{max} -$ `currentPoolSize`
11: **end if**
12: **if** $diff > 0$ **then**
13: $\quad addToResourcePool(diff)$
14: **else if** $diff < 0$ **then**
15: $\quad$ **if** `queueSize` $> estimatedPoolSize *$ `nodeCapacity` **then**
16: $\quad\quad diff \leftarrow$ `queueSize`/`nodeCapacity` $-$ `currentPoolSize`
17: $\quad$ **end if**
18: $\quad removeFromResourcePool(diff)$
19: **end if**
20: `arrivals` $\leftarrow 0$

Figure 5.2: *Runtime algorithm for dynamic dimensioning the size of the resource pool available to the running system.*

An estimate by the inverted USL (Equation 5.1) will not result in a value higher than n_{max}. However, as with the availability-based strategy the portion of the pool that is busy is not considered when estimating the adequate pool size, the overall size could extrapolate the scalability limit (n_{max}) for the service when summing the current pool size with the new allocation. For this reason, the overall pool size must be explicitly verified and limited. Hence the validation in line 9 of the algorithm.

In case of estimating decreasing workload, when negative difference is obtained, there is an additional verification in line 15 to improve the extent of the deallocation. When we use the total number of nodes in `availablePoolSize`, this extra verification can avoid drastic pool size reduction when the queue size is still high. On the other hand, when `availablePoolSize` only considers the available portion of the pool, the verification avoids conservative pool size reductions as, with a decreasing workload, the trend is that more of the pool size capacity is becoming available, which is not considered in the initial estimate.

5.3 Estimating model parameters

We propose to estimate the service-related parameters required to execute the procedure presented in the previous section (`supportedArrivalRate`, `nodeCapacity`, the contention factor, and the coherency factor) by means of load tests, in an off-line profiling procedure. In a first phase of the procedure, the goal is to identify the capacity limit for a single node, which is number of concurrent requests supported by a single instance. We run a sequence of load tests, keeping the service configuration static, with a single instance processing all the requests, and increasing the request rate.

The simplest approach for doing that is starting with a single simulated client and adding more clients at each subsequent run. Then, we observe the throughput variation looking for an inflection point. The capacity limit should be the last point before the inflection. There is a possibility that we can use USL to find the supported arrival rate: we can replace n by the number of simultaneous clients in Equation 2.5, when keeping a fixed infrastructure. Then, we can estimate contention and coherency and apply Equation 2.6 to find the maximum capacity [Gun07]. But it is possible that hardware affect capacity before the theoretical limit. USL is ruled by the number of parallel processing units, the proportion and serial and paralelizable portions of the algorithm, and read/write speed and locks. The availability of main memory, for instance, could bound system capacity before the estimated limit. We can use additional information to find the physical limit. For instance, we can measure resource utilization, estimate the growth pattern, and calculate the limit. Then, we can run the load test at levels around the limit to validate it and collect the necessary measurements. Tchana et al. [TDP+13] applies an heuristic that, from an initial experiment with one simulated client, estimates the maximum supported workload assuming the SUT uses one processing unit. Then, it increases the number of simulated clients to test that limit and, if the system does not saturate, an estimate with one additional processor is made and tested, until reaching saturation.

At the end of this step, we can estimate the maximum throughput supported by one instance, which is equivalent to the supported mean arrival rate (*supportedArrivalRate*), and the mean response time. Multiplying these two values, we obtain *nodeCapacity*, which is an estimate of the number of requests an instance can process in parallel.

Then, we can move to the second phase of tests. It comprises another sequence of runs, starting with one instance available in the service pool and increasing the pool size at each subsequent experiment. The workload should be *nodeCapacity* in the first run and grow linearly with the pool size. We can then compute the throughput of each run and combine the results with the respective pool sizes to estimate contention and coherency, following the method proposed by Gunther [Gun07], described in Section 2.5.2. The method, which does not require exhaustive experiments nor reaching the capacity limit (the author suggests at least six data points), uses statistical regression to estimate contention and coherency factors based on the pool size used in each step of the load tests and the respective provided throughput.

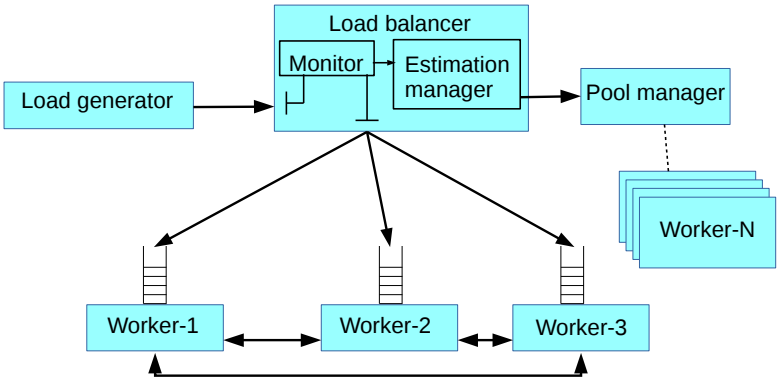

Figure 5.3: *Experimental setup*

Contention and coherency estimates are used by equations 2.5 and 5.1, used at lines 3 and 6 in the algorithm shown in Fig. 5.2, respectively.

5.4 Experimental validation

To evaluate our proposal, we executed experiments on DAS-5[1], a cluster composed of 68 nodes with two 8-core processors working at 2.4GHz, 64GB of RAM, and interconnected via InfiniBand and Gigabit Ethernet. In this section, we describe the experimental setup implemented for the validation, experiments to assess the applicability of USL, and experiments to assess our proposal. In the first experimental phase, we are interested to see whether USL can be used to predict the performance at high load even when we had only ran load tests at a few selected levels of low load. In a second phase, we focus on verifying how the application of our proposal impacts in resource allocation and delivered performance. The results obtained with our proposal are compared to baselines in which we executed the same workloads applying a utilization-based strategy to manage the pool, in which the pool size changes in steps of one node.

5.4.1 Experimental setup

To validate our proposed strategy and algorithm, we implemented a basic experimental setup composed of a load generator, a load balancer, and workers. The load balancer receives requests from the load generator and distributes them among the workers. The requests are enqueued in the workers and each worker has a pool of threads picking up requests from the queue and emulating its execution. Request execution is emulated by sleeping during a specified time and exchanging synchronization messages with the other workers. This setup is used in the first phase, for the USL assessment, and for profiling in the second phase, when we estimate the model parameters for the setup.

Other elements are included to enable dynamic resource allocation, allowing the validation of our proposal. They are: a system monitor, an estimation manager, and a pool manager. The monitor, running within the load balancer, tracks the arrival rate at the load balancer, at regular intervals, and the queue at each worker, providing information to the estimation manager, which requests and releases workers from/to the pool manager. The complete experimental setup is pictured in Fig. 5.3. The source code for the system and experiments are available as open source software[2].

In the simplified, static setup, the load balancer distributes the requests in a round-robin fashion. The elasticity-enabled load balancer, in its turn, forwards the requests to the worker with the lowest queue. In addiction, for parameter estimation, the load generator produces uniform workloads, varying the number of simultaneous processes to simulate concurrent clients.

[1] http://www.cs.vu.nl/das5/
[2] https://github.com/pbmoura/scalability_experiments

Table 5.2: *Validation of USL modeling for batch execution*

Contention	Coherency	Peak	Data points
0.3	0.003	15	6
0.3	0.002	18	6
0.2	0.003	16	6
0.2	0.001	28	6
0.2	0.01	9	6

Table 5.3: *Validation of USL modeling for request streams*

Contention	Coherency	Peak	Data points
0.06	0.004	15	8
0.06	0.01	9	6
0.05	0.002	21	15
0.1	0.005	13	7

The experimental setup receives parameters to configure duration of task time, synchronization time, contention, and the number of threads in the workers's pool.

5.4.2 Validation of USL applicability

Two scenarios were considered to validate the possibility of applying USL to predict the performance of a distributed service: batch execution and request stream. In the first case, the load balancer component works as a coordinator, receiving a batch of jobs, evenly distributing job execution among the workers, and merging the output produced by them. In the second case, the load balancer receives a continuous request stream and distributes the requests among the workers using the round-robin strategy.

We experimented different parameterizations of the setup, leading to different contention and coherency factors. For each parameterization, we ran a set of experiments increasing the number of workers handling the batch, from 1 to 30, with unitary increments. The initially selected parameter sets applied for the batch execution scenario, listed in Table 5.2, produced configurations for which the performance peak was achieved with between 9 and 28 workers. The goal was to verify the possibility to predict the performance peak and provide an approximate performance curve, in comparison with measurements. In the batch execution experiments, in all those cases, no more than the first six executions were required to achieve our goal.

Then, we also ran an experiment with lower impact of contention and coherency, for which the performance peak was estimated to occur with 43 workers. In this case, we needed the eight first measurements to approximate the estimates to the measurements in the batch execution.

In experiments with request stream, we used lower values for contention and needed between the first 6 and 15 measurements to obtain accurate models, as shown in Table 5.3. Figure 5.4 is related to the experiment with request stream, in which we configured the experimental setup to suffer contention factor of 0.06 and coherency factor of 0.004. In this experiment, we used the 8 first measurements to estimate the model drawn by the solid line. The dashed line shows the expected capacity based on the parameterization of the contention and coherency, and the circles mark the measurement.

Gunther suggests to use at least six measurements to estimate the USL parameters for a system. To simplify our search for an accurate model, we started our estimates with the first six data points. and increased the number of consecutive data points until we found adequate parameter values. When the model estimated with the first six measurements was inaccurate (e.g, in Fig. 5.5), we estimated with the first seven, eight, and so on, until we got correct estimates for the performance peak and a capacity curve whose values were close enough to the measurements. The suggestion, however, is that the data points are not sequential. An estimate could be executed, for instance, using

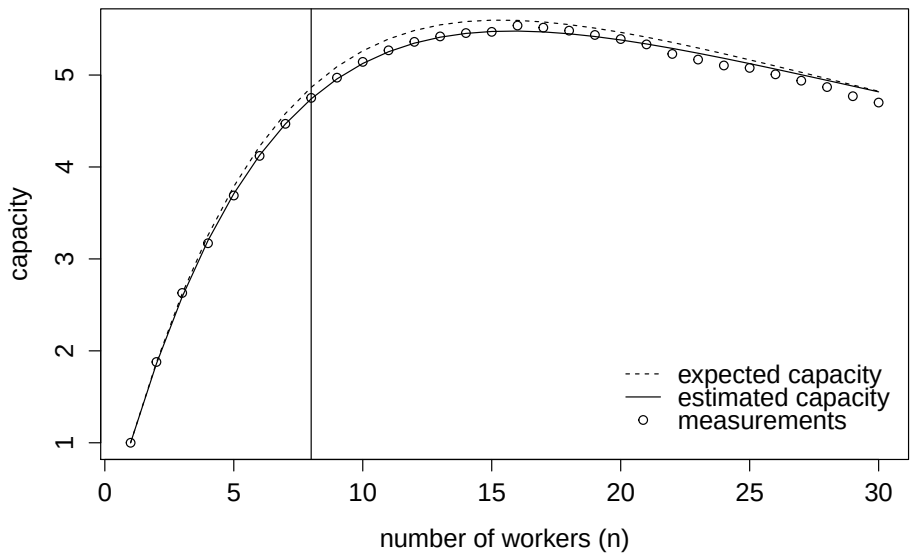

Figure 5.4: *USL estimate for a request stream experiment*

measurements with 1, 2, 4, 8, 12, and 16 workers, or growing the number of workers quadratically, increasing the coverage of the samples. We consider viable the application of USL in the context of this research, as we were able to obtain accurate models in our experiments. But, as it is not possible to know about the precision of the model only with the estimates and the measurements used in the inference, we suggest to measure the capacity at other data points for validation.

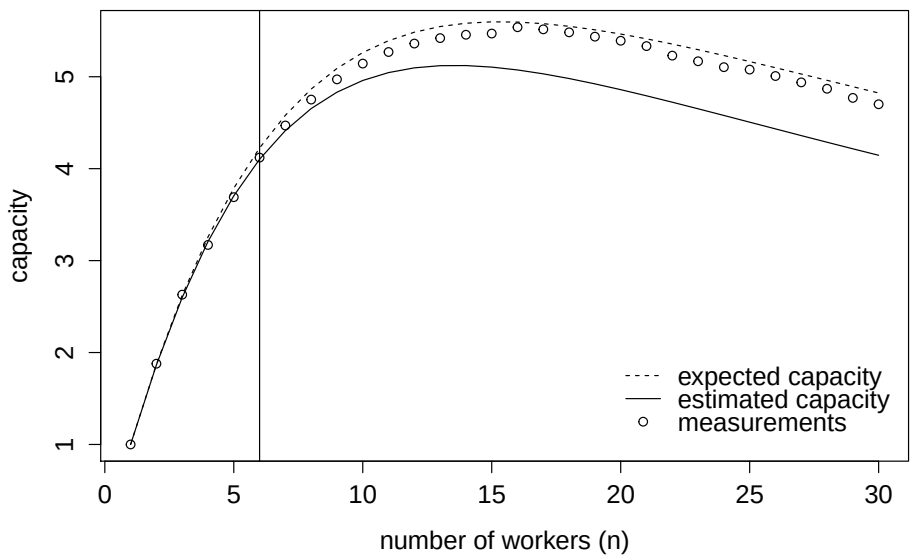

Figure 5.5: *Example of a somewhat inaccurate estimate of model parameters*

Batch execution is not usually considered a scenario where elasticity is required. A fixed set of tasks is prepared and executed. The number of instances used to process the tasks defines the overall batch execution time. A scalability model is helpful, though, for planning. We can use the model to choose the most appropriate pool size, considering the balance between execution costs

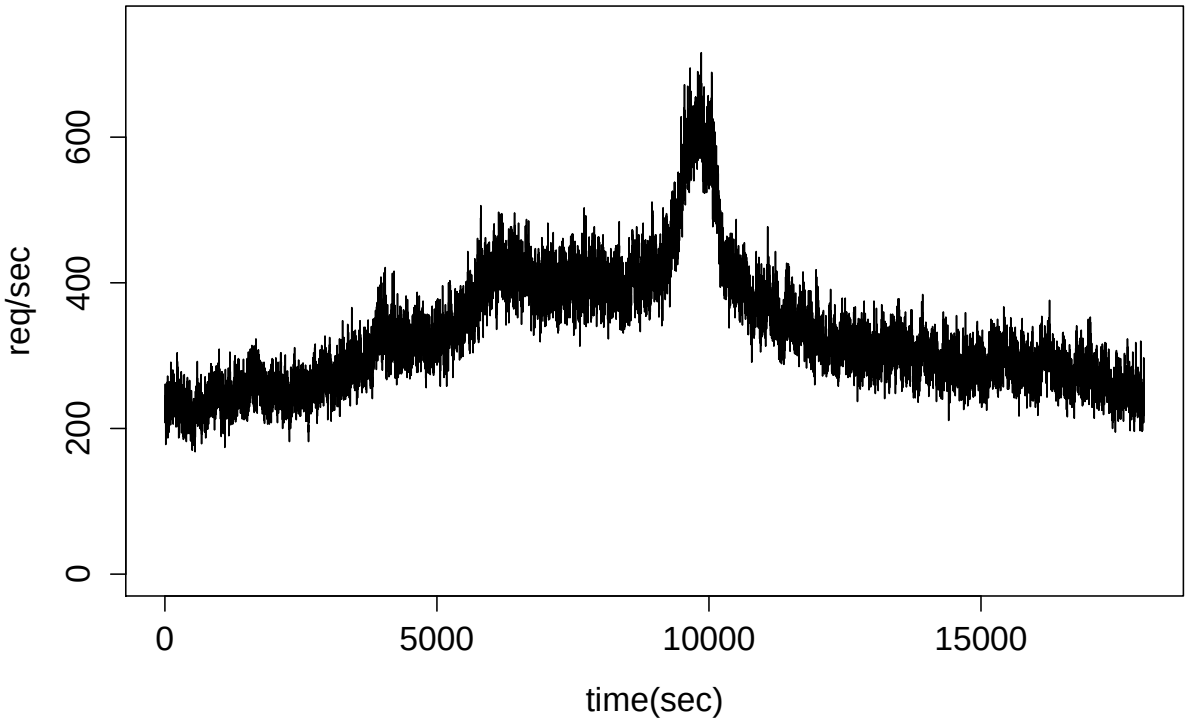

Figure 5.6: *Extract from FIFA World Cup 1998 workload used in the experiments*

and execution time.

On the other hand, for systems that handle online request streams, we cannot usually estimate beforehand what is the appropriate pool size. As the workload varies along time, the resource demand changes. Hence, the pursuit for automated resource management with dynamic system reconfiguration. The experiments described in the next sections are based on request streams. In those, we assess the applicability of our proposal for automated resource management.

5.4.3 Trace-driven experiments

Here, we present the experimental results obtained with a real trace to assess the applicability of the proposed strategy for resource management under a realistic workload pattern. In these experiments, we used the setup described in Section 5.4.1 (Figure 5.3) and made the load generator reproduce requests in a rate equivalent to the one in a portion of the workload received by the website of the FIFA World Cup 1998[3]. Generally, the website received increased workload during match times, with a spike around the end of the match. We used a portion of this workload in which 5,956,710 requests were received by the servers in a period of nearly 5 hours. The request rate variation we used is depicted in Figure 5.6. The parameters estimated from the experimental setup, following the procedure described in Section 5.3, are listed in Table 5.4. We set service time to 200ms, sequential time to 16ms, synchronization time to 0.4ms, and each worker running 20 threads to process the requests. Executing load tests with a single worker, we obtained maximum throughput around 98.9 req/s, with mean response time of 201ms. It gives us a node capacity of 19.88 simultaneous requests. Then, after running the load tests increasing the number of workers and the workload, we estimated contention and coherency factors of 0.08 and 0.002 respectively, using the algorithm described in Section 2.5.2.

[3]available at http://ita.ee.lbl.gov/html/contrib/WorldCup.html

Multiple experiments were executed with this workload to evaluate the different strategies described in Section 5.2. They are listed in Table 5.5. The column **Arrival rate estimate interval** informs the extent of the interval used to estimate the arrival rate: *full* indicates that all the monitoring interval is used and *last 20%* indicates that the 20% most recent requests of the monitoring interval are used. In **Available resources** we use the value *all* when all the pool size is used to calculate how many VMs to request or release; we use *available* when only the available capacity of the current pool, as defined in Section 5.2, is used in the calculation. A value *no* in the column **Pool size limit** indicates that we suppressed the validation that limits the pool size to n_{max} (lines 9 and 10 of the algorithm in Figure 5.2), allowing larger pool sizes. Hence, we could validate the effect of overallocation, when the pool size is too large and the synchronization time degrades the performance. The parameterization used for these experiments implies in a limit of 21 workers. For baseline, we executed an experiment in which resource allocation is based on the utilization. Given the impossibility of measuring CPU utilization of our experimental setup, as it emulates task execution with sleeps, we measure utilization as the proportion of the threads (from the workers's thread pool) effectively in use. We used thresholds of 90% and 45% to request and release one VM, respectively. The baseline is identified by the label *single-step* as the pool size is always updated in steps of one node.

The initial pool size for all the experiments was defined by calculating the mean arrival rate for the first 2 minutes of the workload and applying the proposed estimate procedure, i.e., dividing it by the maximum throughput estimated for one worker and calculating the inverted USL (Equation 5.1). In doing so, we set the pool size to start with three workers.

We ran experiments with 10 minutes and 30 minutes of monitoring interval, which is the time elapsed between two consecutive executions of the monitor. We defined an SLA of 3 seconds and calculated the percentage of SLA violations. The cost of an execution is calculated as the sum of the total time of workers allocation, in seconds. For instance, if in an experiment lasting 60 seconds, one worker is allocated for the entire experiment and another for 40 seconds, the cost of this execution is 100. Since the IaaS cloud providers usually charge per allocated time, based on the type of instance, we can use the cost metric to estimate the expenses incurred from running experiments. For instance, in an experiment with cost 90,000, we have that it consumed 25 hours of VM allocation. If the used instance costs 2 cents per hour, we have an estimated cost of 50 cents.

Experiments with monitoring interval of 10 minutes

The measurements obtained in the experiments with monitoring interval of 10 minutes are summarized in Table 5.6. We tracked the variations in pool size, queue size, and response time. Cost is calculated based on the pool size variation and, therefore, mean pool size and cost are strongly correlated. The mean response time is correlated to the mean queue size, as queuing is the major factor in the increase in response time. Both these correlations are very high, over 0.99. The SLA violations have strongest correlations with the mean pool size and mean response time (0.97 and 0.95, respectively), and inverse correlation of -0.94 with the cost. It suggests it is virtually impossible to improve the offered quality of service without increasing costs. But we can see that the relation between cost and SLA violations is not linear.

In comparison with the baseline, we see that, using *USL* and *USL-short*, the cost reduction was slightly better than 30% but, while the first caused response time increment of 116%, the latter

Table 5.4: *Estimated parameters for trace-driven experiments*

Parameter	Value
mean response time	201ms
maximum throughput with one worker	98.91197 req/s
contention factor	0.08457
coherency factor	0.002

Table 5.5: *The 6 strategy variations evaluated in our experiments*

Label	Arrival rate estimate interval	Available resources	Pool size limit
USL	full	all	yes
USL-short	last 20%	all	yes
USL-avail	full	available	no
USL-avail-short	last 20%	available	no
USL-limit	full	available	yes
USL-limit-short	last 20%	available	yes

Table 5.6: *Measurements of trace-driven experiments with 10 minutes of monitoring interval*

Metric	Single-step	USL	USL-short	USL-avail	USL-avail-short	USL-limit	USL-limit-short
Mean pool size	8.2	5.3	5.4	12.2	12.7	11.2	11.5
Max. pool size	11	18	14	38	39	21	21
Mean response time (ms)	4562	9856	5412	543	806	456	459
Std. dev. of response time	15375	27125	11510	457	1588	172	161
Mean queue size	1811	4222	2617	156	165	144	146
Max. queue size	42417	67677	20810	439	528	366	332
Cost	142501	96602	97802	219619	229522	201608	206407
SLA violations %	9	25	21	1	4	0	0
Cost variation over single-step	1	0.68	0.69	1.54	1.61	1.41	1.45
Response time variation over single-step	1	2.16	1.19	0.12	0.18	0.1	0.1

caused an increment of only 19%. On the other hand, *USL-limit* and *USL-limit-short* reduced the response times in 90% with cost increment of 41% and 45%, respectively. Hence, we have that *USL-short* is the best choice when limiting cost is fundamental, while *USL-limit* provides best response time and SLA violations maintenance.

Figure 5.7 shows the response time and pool size variations along the experiment with utilization-based pool management (*single-step*). We see a high disturbance in the response times, which happened with the load spike. Figures 5.8 to 5.13 show the variation in all the evaluated strategies based on USL.

The strategies based on availability (Figs. 5.10 to 5.13) produce higher pool sizes since the early stages of the execution, leading to higher cost. However, this reflects in the response time of the arriving requests, which is lower, providing a better quality of service. Also, we can see that there are reductions of the pool size in moments where the workload did not drop. This is due to the allocation not considering the portion of the processing capacity that is in use. Thus, estimation manager estimates a resource need and the allocation is calculated so that the estimated amount of resources is promptly available to process the arriving requests. However there was already a portion of the previous capacity (before the new allocation) in use. As those old requests are processed and returned, that processing capacity is available to the new arriving requests. Hence, the availability increases and the pool size can be reduced in the forthcoming monitoring cycle.

Strategies *USL* (Fig. 5.8) and *USL-short* (Fig. 5.9) are the most economic, but cause much more SLA violations due to underprovisioning, performing worse than the baseline (Fig. 5.7) in terms of response time and SLA violations. Apart from these two experiments, the SLA violations occurred

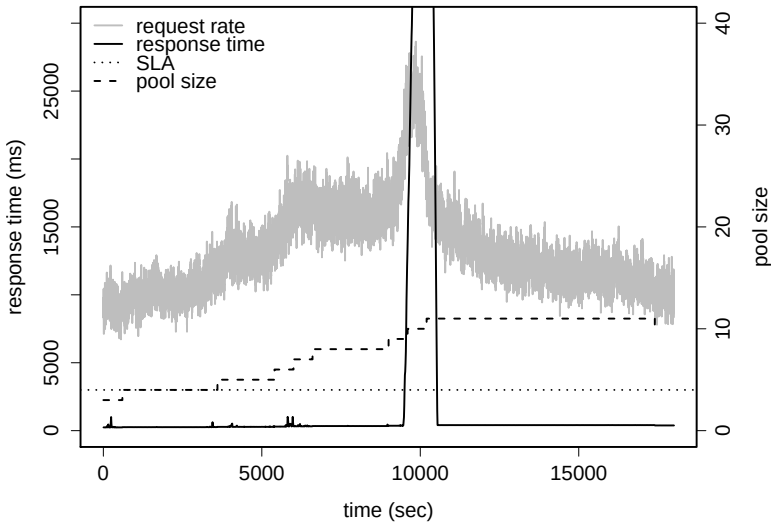

Figure 5.7: *Performance of the single-step strategy with 10 minutes of monitoring interval*

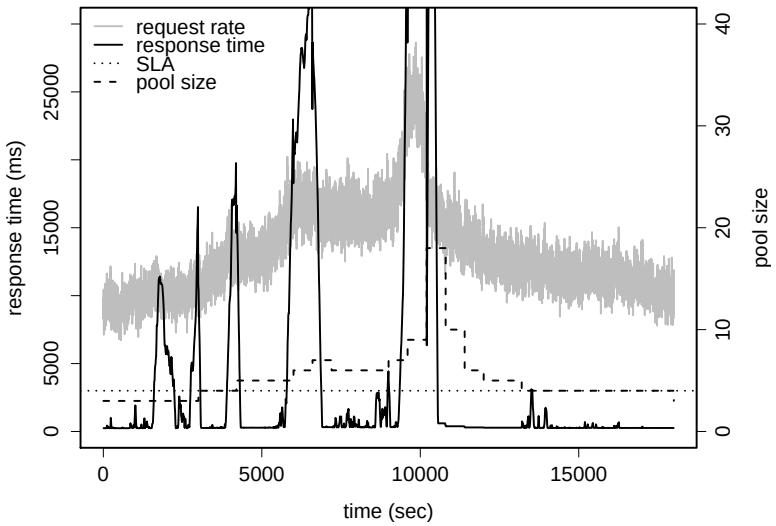

Figure 5.8: *Performance of the USL strategy with 10 minutes of monitoring interval*

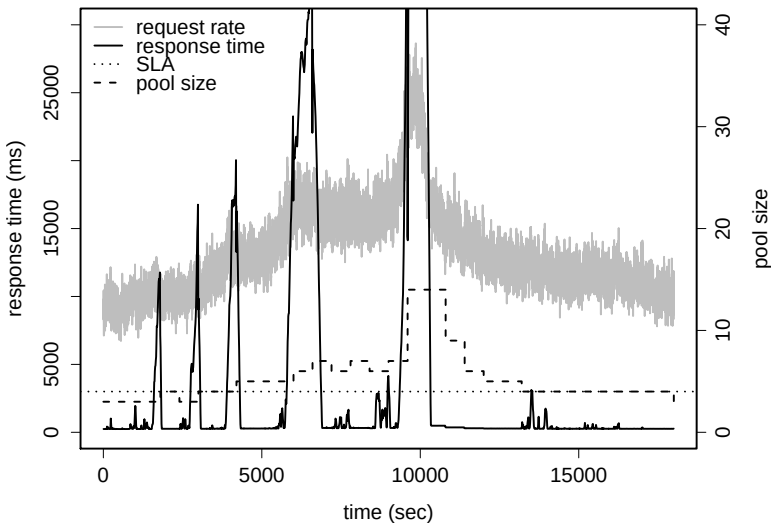

Figure 5.9: *Performance of the USL-short strategy with 10 minutes of monitoring interval*

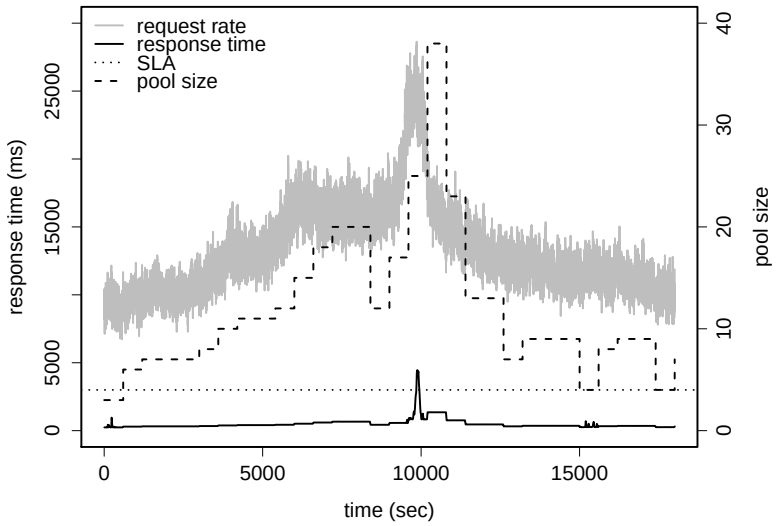

Figure 5.10: *Performance of the USL-avail strategy with 10 minutes of monitoring interval*

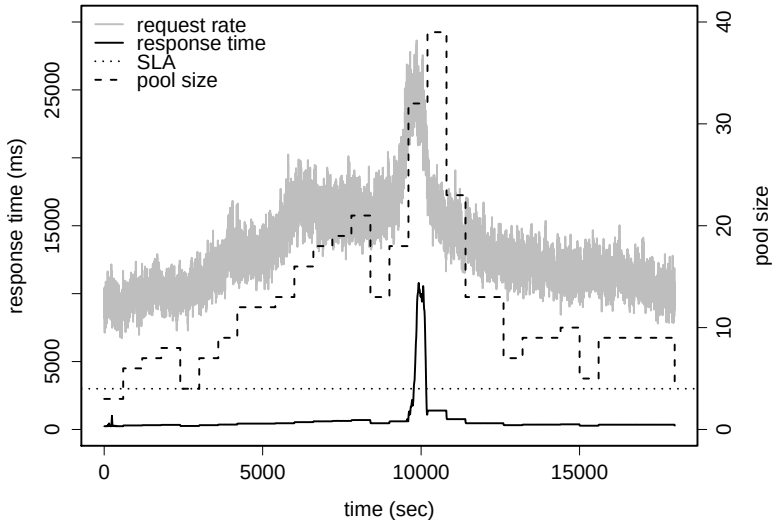

Figure 5.11: *Performance of the USL-avail-short strategy with 10 minutes of monitoring interval*

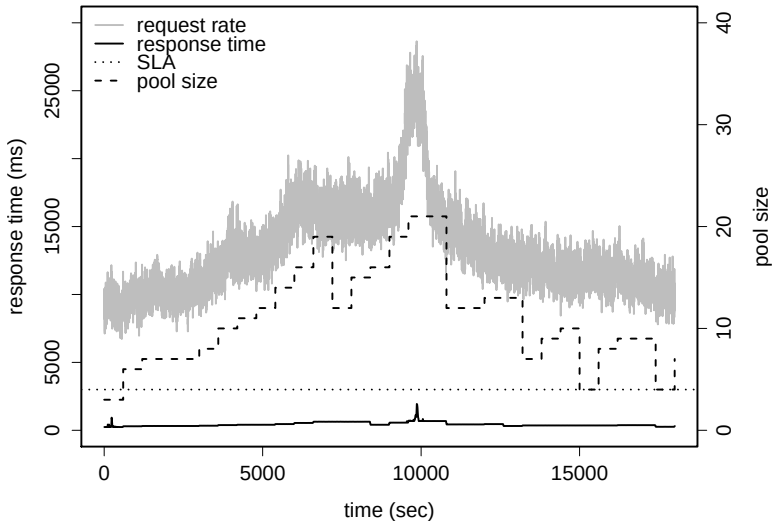

Figure 5.12: *Performance of the USL-limit strategy with 10 minutes of monitoring interval*

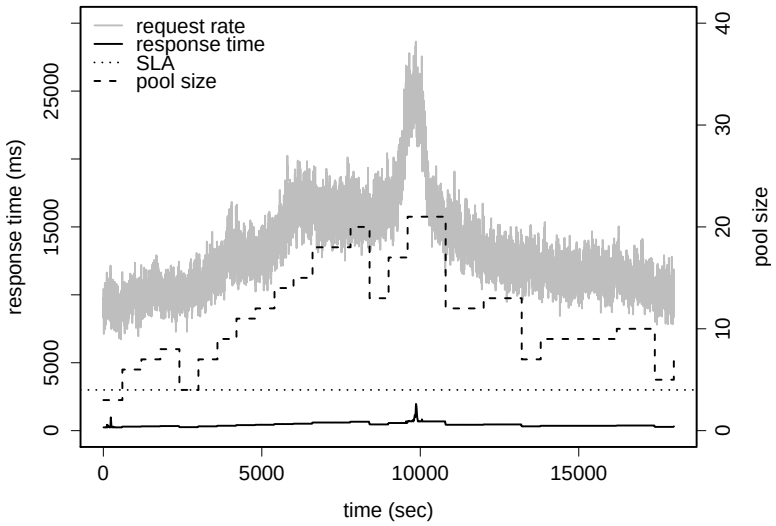

Figure 5.13: *Performance of the USL-limit-short strategy with 10 minutes of monitoring interval*

only in the load spike that happened around the 10,000 seconds mark of the experiments. The availability-based approaches were able to reduce the performance degradation during the spike, which is precisely our aim.

We can also observe that letting the pool size grow indefinitely (Figs. 5.10 and 5.11) caused worse performance than when the pool size is limited (Figs. 5.12 and 5.13) to what the model estimates as a theoretical limit for optimizing the capacity of the system. This is due to the increased time spent with data synchronization among the instances when the theoretical limit is surpassed.

Experiments with monitoring interval of 30 minutes

We also executed experiments with monitoring interval of 30 minutes to validate the hypothesis that, with longer monitoring intervals, the benefit of our proposal is increased, when considering the strategy based on availability in comparison with the baseline. Confirming this hypothesis gives the possibility to increase the monitoring interval, reducing the overhead of resource management. Experiments *USL-avail* and *USL-avail-short* were not executed with this configuration because, in experiments *USL-limit* and *USL-limit-short*, the scalability limit of 21 nodes was not achieved.

The measurements obtained with 30 minutes of monitoring interval are in Table 5.7 and Figs. 5.14 to 5.18 show the respective response time and pool size variations. The correlations are similar to those observed in the experiments with monitoring interval of 10 minutes. SLA violations have correlations of 0.97, 0.96, and -0.94 with mean response time, mean queue size, and cost. Also, while *USL* caused response time to grow almost 10 times with a reduction of 28% in cost, *USL-limit-short* improved response time in 25 times with a cost increment of 54%.

Similarly to what we saw with the shorter monitoring interval, we can see a meaningful performance improvement when applying the availability based strategy. Yet, in this case there is a small benefit of using the short, recent history to estimate the workload. Comparing the pool size variation in Figs 5.16 and 5.17, we see that in the latter the pool size variation is better correlated to the workload variation.

Regarding strategies *USL* (Fig. 5.14) and *USL-short* (Fig. 5.15), the performance degraded significantly, both comparing with the baseline and with the same strategy using the shorter monitoring interval. We can see long periods of very high response times, with short spikes happening when the pool size increases. This is because of under-allocation, leading the saturation of workers. When new instances are included in the pool, they are completely available and the load balancer directs all the arriving requests to them. Initially, the new instances provide a good response time, causing the spike, but they get quickly saturated, because they are receiving a workload that was supposed to be distributed among all the servers in the pool.

Table 5.7: *Measurements of trace-driven experiments with 30 minutes of monitoring interval*

Metric	**Single-step**	**USL**	**USL-short**	**USL-limit**	**USL-limit-short**
Mean pool size	7	5	4.9	11	10.8
Max. pool size	9	9	7	19	19
Mean response time (ms)	12119	120230	83099	590	536
Std. dev. of response time	33897	149061	101332	926	820
Mean queue size	3871	47081	28448	301	299
Max. queue size	322291	210121	130671	2887	2889
Cost	113400	81901	88200	180004	174604
SLA violations %	20	63	56	3	2
Cost variation over single-step	1	0.72	0.78	1.59	1.54
Response time variation over single-step	1	9.92	6.86	0.05	0.04

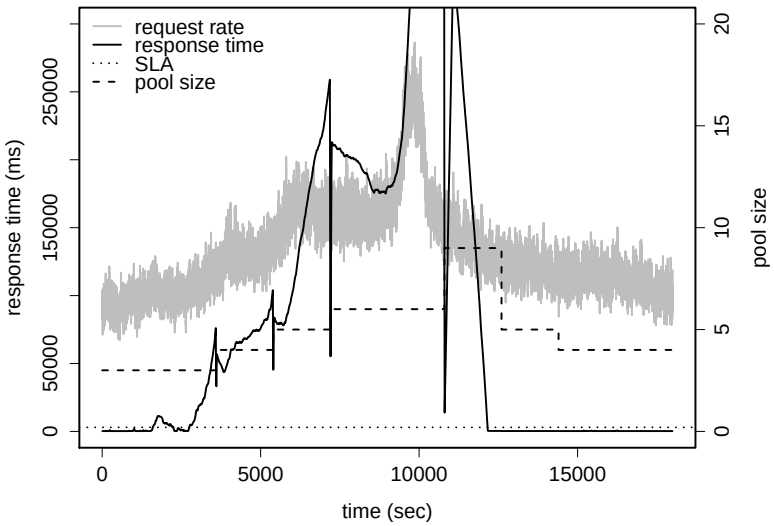

Figure 5.14: *Performance of the USL strategy with 30 minutes of monitoring interval*

When we compare the results obtained with 10 and 30 minutes of monitoring interval (Table 5.8), we see a reduction of between 10% (*USL-short*) and 20% (*single-step*) in costs, i.e., with the larger monitoring interval, the costs were lower. The variation in the response time, however, ranges from 17% to 1435% over all the strategies, i.e., with the lower monitoring interval, the response time tends to be smaller as the system adapts to the demand more rapidly. It may suggest that increasing monitoring time reduces costs, as there will be less modifications of the resource pool. But this finding may be related to the characteristic of the trace used in the experiments, in which the period of workload increase seams longer than the period of workload decrease. In any case, the effect on performance looks more dependent of the management strategy. Although the *single-step* strategy led to the highest reduction in cost, *USL-limit-short* was more effective when considering the response time variation. To illustrate, we can show that, with a monitoring interval of 10 minutes, 9% of the requests violated the SLA with *single-step* and this value more than doubled, to 20%, with monitoring interval of 30 minutes. In its turn, *USL-limit-short* did not cause any SLA violation with an interval of 10 minutes and, with 30 minutes, SLA violations happened in

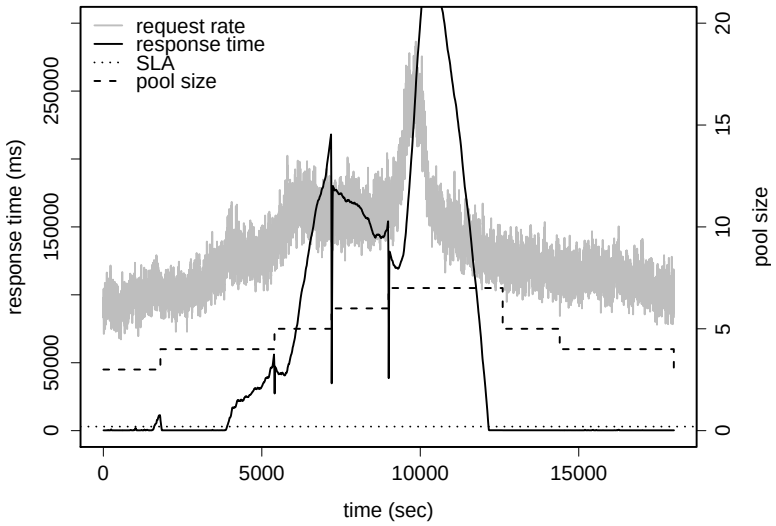

Figure 5.15: *Performance of the USL-short strategy with 30 minutes of monitoring interval*

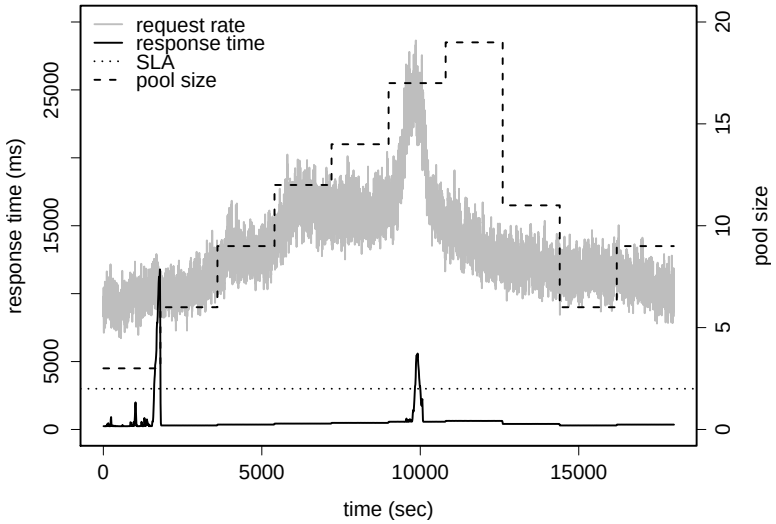

Figure 5.16: *Performance of the USL-limit strategy with 30 minutes of monitoring interval*

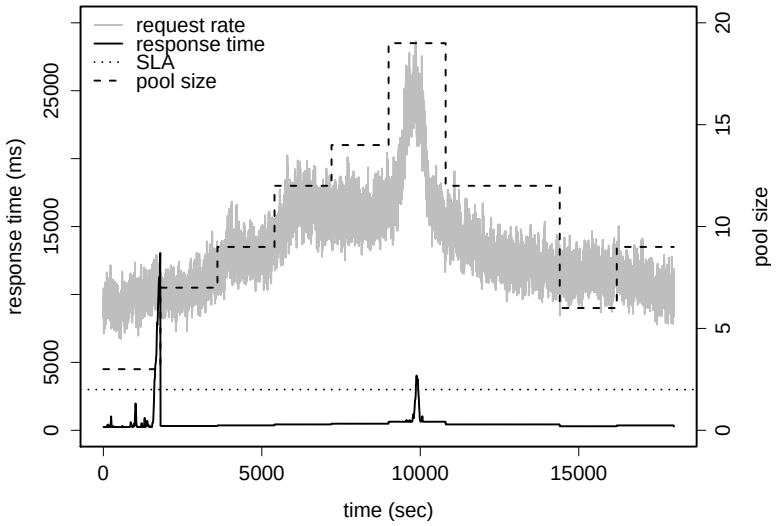

Figure 5.17: *Performance of the USL-limit-short strategy with 30 minutes of monitoring interval*

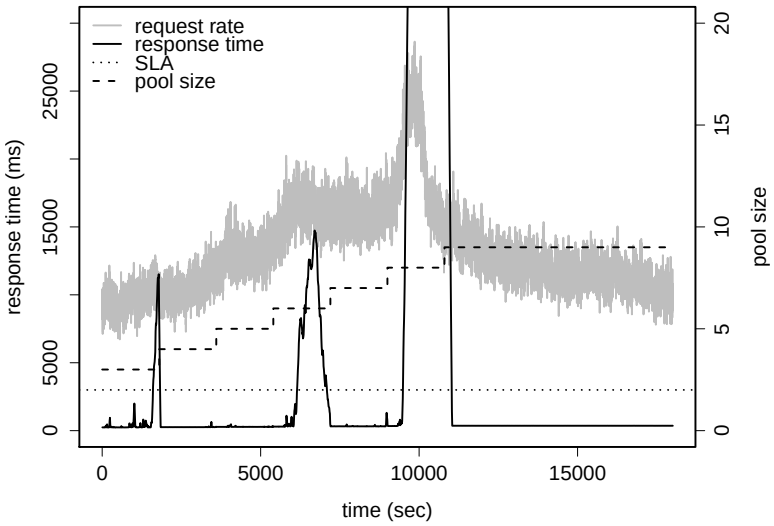

Figure 5.18: *Performance of the single-step strategy with 30 minutes of monitoring interval*

only 2% of the requests. Hence, the results give evidence of the applicability of long monitoring intervals, reducing costs and the overhead incurred by the management system, with limited effects on performance maintenance.

Table 5.8: *Percentage of the metrics obtained in the trace-driven experiments with 30 minutes of monitoring interval in relation with those obtained with 10 minutes*

Strategy	**Cost**	**Resp. time**
single-step	80%	266%
USL	85%	1220%
USL-short	90%	1535%
USL-limit	89%	120%
USL-limit-short	85%	117%

5.4.4 Synthetic workload experiments

The trace we used in the previous section includes a smooth increment in the arrival rate, followed by a period of stagnation, a short spike (sudden increase and decrease in a short time interval), and ends with a smooth decrease. Although the importance of assessing our proposal in such scenario, it is also relevant to evaluate our proposed strategies under condition of drastic variations in the workload. Thus, we also executed experiments with synthetic workloads that exercise more extreme conditions and present the results in this section.

Here, we describe the results of experiments with a workload in which there are two nearby load pikes and a second set of experiments in which, after a sudden load increase, the workload forms a plateau, remaining high for a period. Those experiments were executed with a reduced time scale for which the monitoring interval was set to 20 seconds so that we could also evaluated the proposed strategy in a highly dynamic environment. The estimated parameters to feed the monitor and estimation manager are shown in Table 5.9. Due to the short time intervals, we only ran experiments using the complete monitoring interval to estimate the workload. Additionally, we ran a sequence of experiments to verify the impact of different monitoring intervals in the different strategies.

Table 5.9: *Setup estimated parameters for synthetic workload experiments*

Parameter	Value
mean response time	5.002s
maximum throughput with one worker	0.769
contention factor	0.0398
coherency factor	0.0031

Experiments with two load spikes

Regarding the workload with two spikes, the frequency of requests in this workload follows a pattern that stays 60s in each of the following levels of requests/s: 1, 4, 1, 3, 5, and 1. In Fig. 5.19 we can see the variation of the response times in experiments based on utilization and on USL considering all the pool or just the available portion, with their respective percentage of SLA violations (limit of 13 seconds). The pool size variations are shown in Fig. 5.20. Measurements are summarized in Table 5.10.

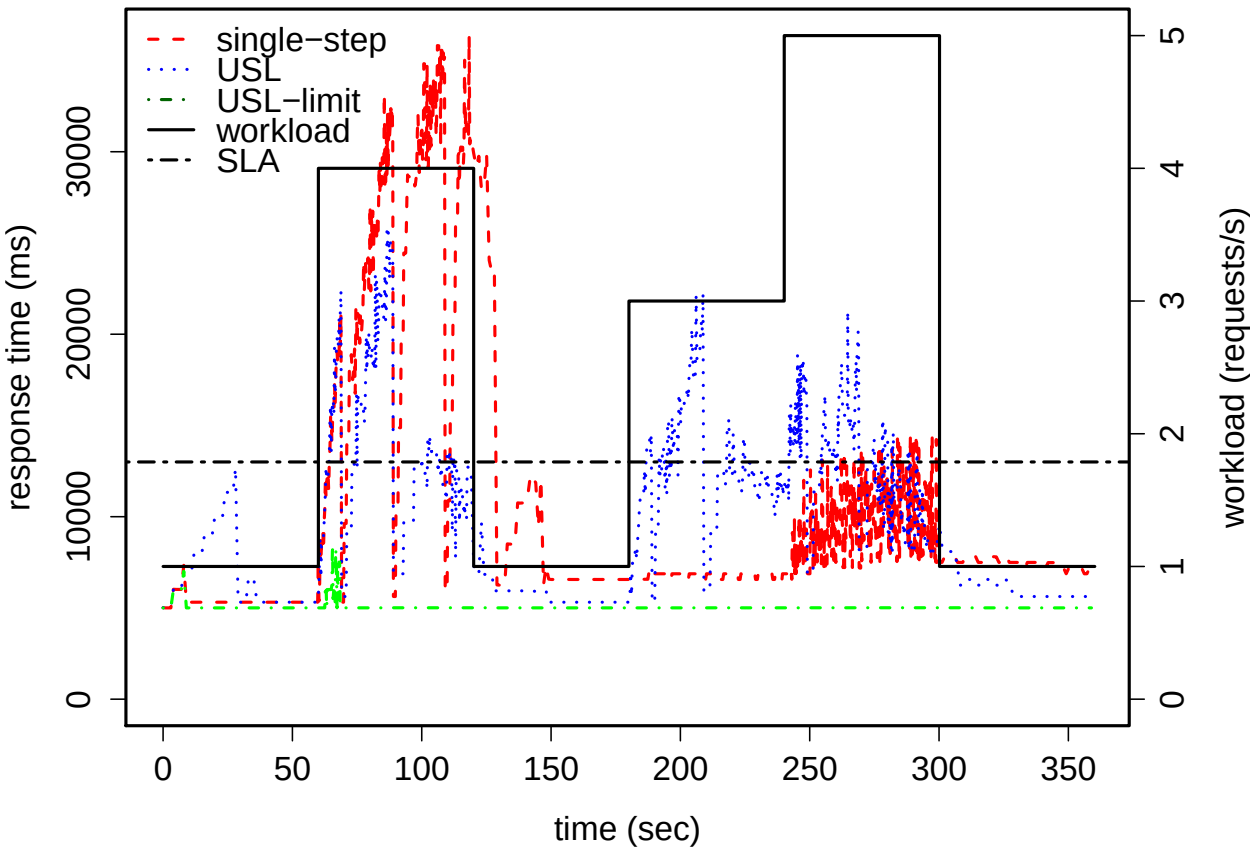

Figure 5.19: *Response time variations in experiments with a synthetic workload with two spikes*

With this workload, *USL* provides the worst results. When the first spike ends, the *USL* strategy causes a quick pool size shrinkage and, as remaining workers are busy, the performance is affected when the second spike comes. Due to its smooth variation of the pool size, the *single-step* strategy benefits from a yet large pool when the second spike starts, providing better performance, in spite of higher cost. Yet, *USL-limit* provides the best performance with no disturbance at the second load spike as it estimates larger pool needs than *USL*, resulting in the highest cost.

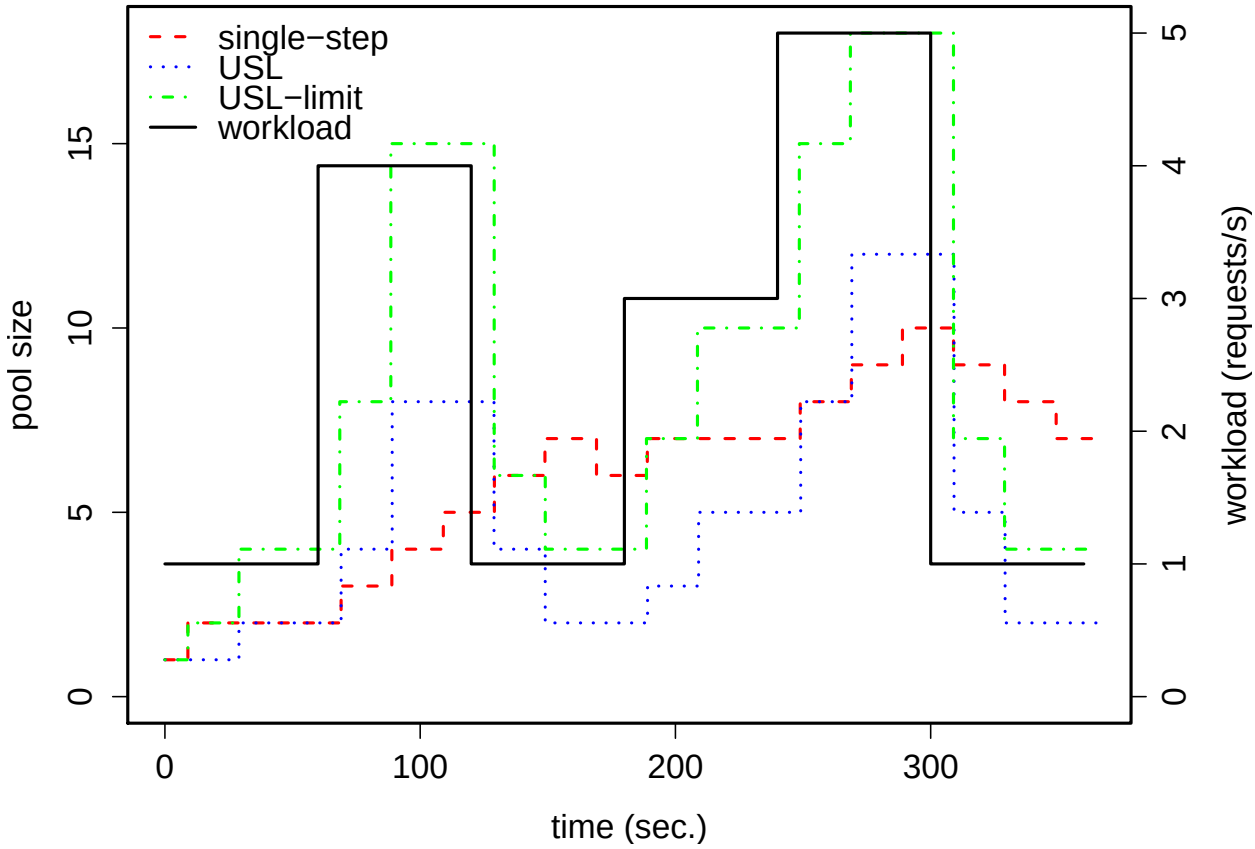

Figure 5.20: *Pool size variation in experiments with a synthetic workload with two spikes*

Experiments with load plateau

In the following experiments, we chose a workload that grows and remains at a high level long enough to stabilize the request queue in all the experiments, as shown in Fig. 5.21. The workload starts at 1 request/s, after 60 seconds changes to 4 requests/s and after 240 seconds at that level, changes back to 1 request/s. Our intention is to observe the time that the system takes to stabilize its performance in this scenario and the related effects. While the *single-step* strategy took 194 seconds to stabilize, using *USL*, we needed 94 seconds and, with *USL-limit*, only 16 seconds.

Figures 5.22 and 5.23 show the variations of response time and pool size respectively, and measurements are listed in Table 5.11. In this case, *USL* was better than *single-step* both in terms of cost and performance, different from what happened in the other experiments. *USL-limit* is, again, the most effective in terms of performance although it is also the most expensive.

Table 5.10: *Measurements of experiments with a synthetic workload with two spikes*

Metric	USL	single-step	USL-limit
Mean pool size	4.76	5.93	8.49
Max. pool size	12	10	18
Mean response time (s)	11.38	12.14	5.05
Std. dev. of response time	4.44	8.26	0.27
Mean queue size	28.16	29.87	12.92
Max. queue size	80	104	27
Cost	1735.57	2171.38	3091.43
SLA violations %	33	25	0

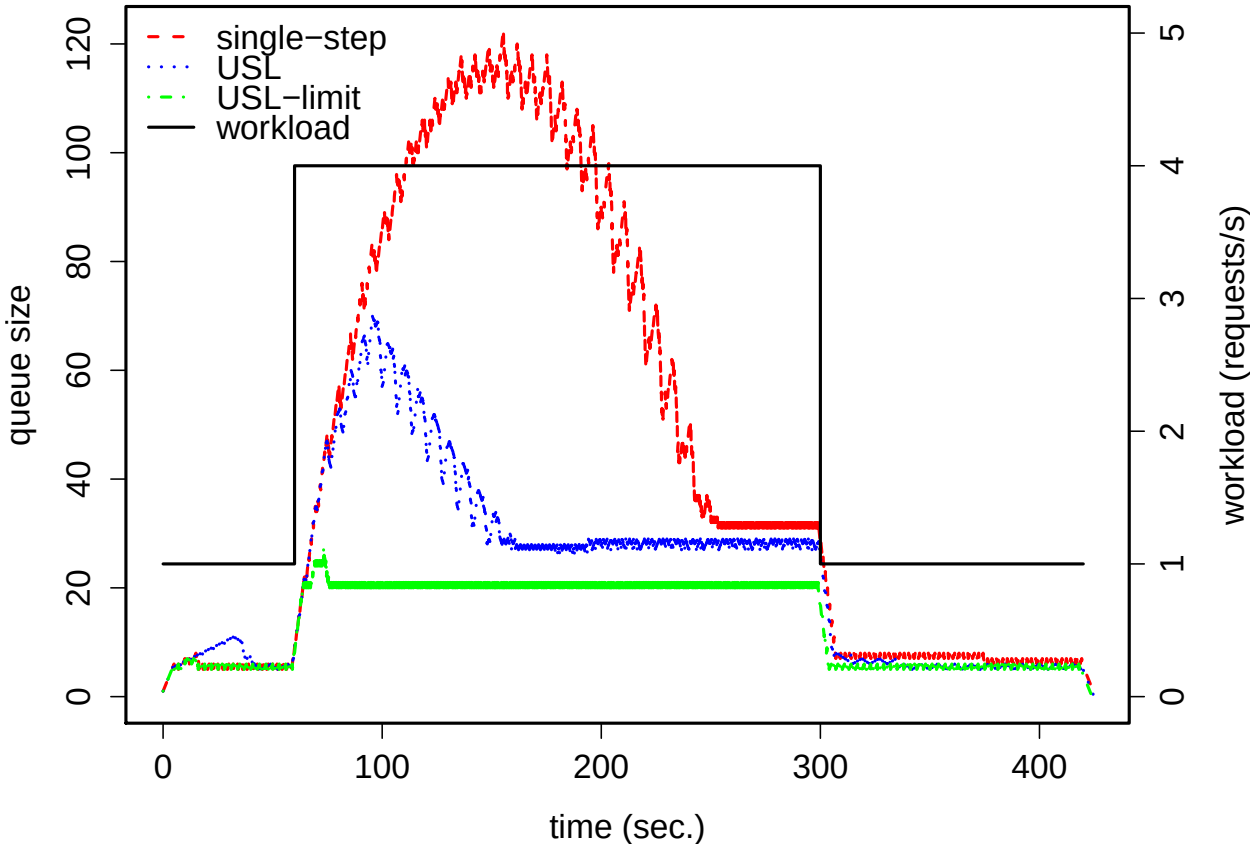

Figure 5.21: *Queue size variation in experiments with a synthetic workload with a plateau*

Varying the monitoring interval

We ran experiments to evaluate how the monitoring interval affects the performance of the different approaches regarding SLA maintenance. For that, we used different monitoring intervals, varying in the range from one to 30 seconds, using a workload that starts at 0.5 requests/s, after 30s grows to 4 requests/s, and after 60 more seconds, reduces to 1 request/s, and ends. Table 5.12 shows the respective percentage of SLA violation observed with each strategy, for each monitoring interval.

We can see a very quick degradation of the performance obtained with the utilization-based strategy. The efficacy of a pool-update strategy based on verification at fixed time intervals depends on when, in the period between two consecutive verifications, the workload changes. If the workload increases just after the monitor iteration, the system will run overloaded for a longer period, resulting in more SLA violations. Conversely, if the workload increases just before the monitor iteration, the

Table 5.11: *Measurements of experiments with a synthetic workload with a plateau*

Metric	USL	single-step	USL-limit
Mean pool size	5.2	6.9	9.8
Max. pool size	8	11	15
Mean response time (s)	8.45	16.44	5.04
Std. dev. of response time	3.6	9.83	0.23
Mean queue size	22.7	44.2	14
Max. queue size	70	122	27
Cost	2200.9	2929.1	4174.6
SLA violations %	9	52	0

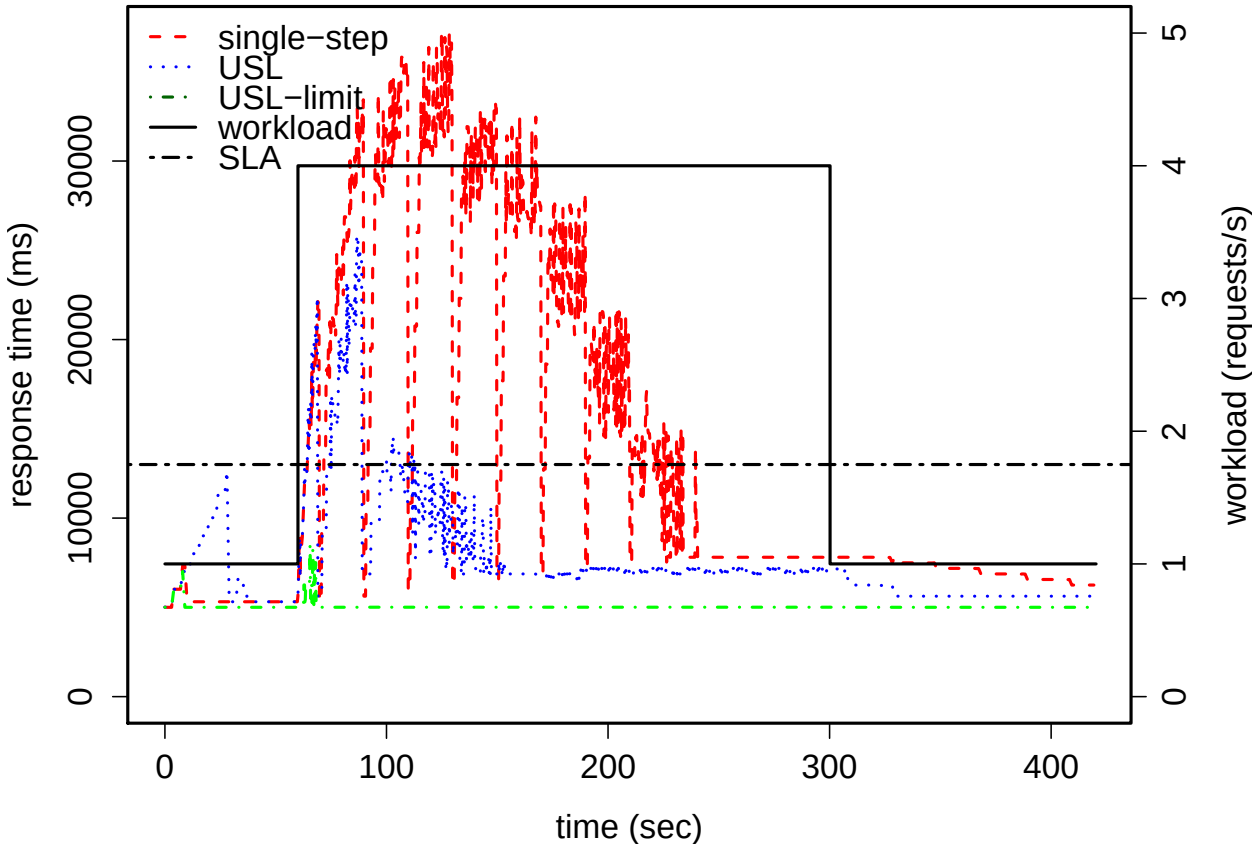

Figure 5.22: *Response time variations in experiments with a synthetic workload with a plateau*

Table 5.12: *Monitoring interval vs. SLA violations*

Monitoring interval	USL	Single-step	USL-limit
1	11	0	0
2	2	5	0
3	1	17	0
4	4	66	1
5	6	72	1
7	13	70	1
10	13	80	8
13	22	83	11
15	66	85	3
20	27	85	21
30	71	88	21

pool will be the quickly adjusted, leading to better SLA maintenance. We see this effect in the experiments using *USL* at 15 and 30 seconds of monitoring interval, and *USL-limit* at 15 seconds. In case of a drastic variation of the workload, the utilization-based strategy is less sensitive to this issue because it will take a possibly long sequence of monitoring cycles until stabilization. Whenever the workload changes, inside a monitoring cycle, there will be SLA violations for some cycles, until the system stabilizes again.

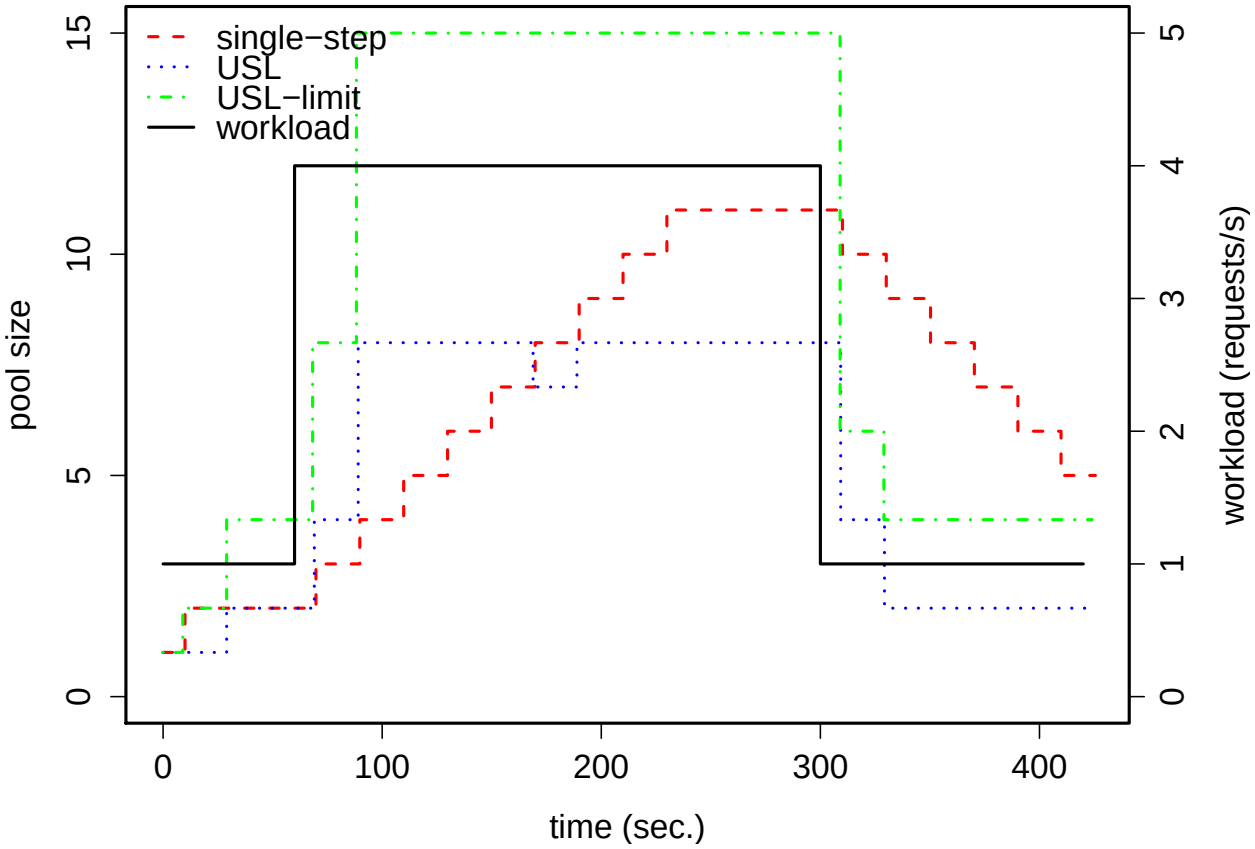

Figure 5.23: *Pool size variation in experiments with a synthetic workload with a plateau*

5.5 Final considerations

In this chapter, we presented our proposal for automated resource management in elastic systems. The strategy requires modeling the system scalability. We showed how we inverted the USL model to apply in our strategy and the procedure to estimate the parameter models. The model is used by a controller to estimate what is the resource need to handle the received workload.

We experimentally evaluated six variations of our proposal, comparing with a baseline strategy based on utilization. We observed that, on the one hand, if reducing costs is paramount, the application of our strategy using all the pool size to estimate the number of VMs to request or release is suggested, specially if extreme variations of the workload are not frequent. On the other hand, if reducing the recurrence of SLA violations is more important, using our strategy considering only the available capacity of the current pool is used in estimates.

Additionally, limiting the maximum pool size that the workers can use is recommended. After a limit, that is estimated by Equation 2.6, the additional nodes lead to performance degradation, beyond the fact that larger pool sizes imply in higher costs. In the next chapter, we discuss the findings and limitations of this thesis.

Chapter 6

Discussion

In this thesis, we implemented a framework for automated scalability evaluation of distributed systems, proposed a technique for assessment of system scalability, and a strategy to model the scalability of a system, applying it to the automation of dynamic resource management. In this chapter we discuss our main findings, based on the observations we made on the empirical evaluations. Our results suggest that the application of the model-based strategy using the current resource availability provides, in general, the best option among the observed. It improves the performance in a higher extent than it increases costs.

6.1 Findings

We, first, discuss the findings related to the Scalability Explorer. Later, we discuss our proposal for dynamic resource allocation based on scalability modeling.

6.1.1 The Scalability Explorer

We started this thesis research by reviewing works related to scalability evaluation of distributed systems. They are summarized in Section 3.1. We found evaluations being manually performed and tools that are restrictive in many aspects such as with respect to which kinds of system can be evaluated, how to vary the workload along the experiments, when to stop an experiment etc. Based on these findings, we incrementally designed and implemented a software framework, described in Chapter 4. Our aim, with that, was to offer a tool that provides flexibility to the user, allowing to adapt the scalability evaluation to contextual specificities. With these activities, we addressed **RQ1** (*Which elements and procedures are required for scalability evaluation of distributed systems?*).

The most important finding at this point, however, was the lack of precise criteria to evaluate scalability, which is related to **RQ2** – *How to produce a meaningful assessment of the scalability of a distributed system?* Most of the tools and frameworks summarized in Section 3.1 fall in one of those categories: those that only support the experiment execution but do not provide any automated analysis; those that calculate scalability ratios, or generate graph of speedup, response rate, etc; and those that find the maximum capacity of the system.

To address this limitation, we proposed a tool-based strategy for assessment of scalability. It consists of verifying whether a system is scalable by testing whether the performance obtained at different scales are equivalent (Section 4.4). Our objective was to eliminate subjectivity in scalability analysis. For instance, Jogalekar and Woodside consider a "perfect scalability" the case when their metric returns 1, which happens in cased of linear scaling. However, they "arbitrarily use the threshold of 0.8" to consider a system scalable [JW00]. The metric is calculated as the ratio of the productivity delivered by two different scales. Thus, considering that a system presents sub-linear scalability, the farther apart the scales are from each other, the lower the scalability ratio should be. Chen and Sun [CS06] use their isospeed-e metric to analyze tree parallel algorithms, running experiments with five different node sets. They obtained outputs between 0.22 and 0.61, where

the ideal is also 1, in case of linearity, and simply identified the most scalable algorithm among the evaluated. With the ANOVA test that we are using, we aim at eliminating the arbitrariness, the vagueness, and the assumption of linearity from the evaluation of scalability. The application of this strategy, however, requires identifying at which pattern the system scales. In the example illustrated in Section 4.4, we used the computational complexity of the algorithm as a basis to verify a scalability pattern that provides equivalent performance at different scales. Another possibility is to estimate the empirical scalability model of the system, as we did in our proposal for automated resource management.

6.1.2 Dynamic resource management

Regarding **RQ2**, we also suggested shifting the attention of scalability evaluation from "Is the system scalable?" to "How does this system scale?", and modeling the system scalability curve (Section 5.1). Scalability models are not a novelty. But, to the best of our knowledge, the application we gave to the models is new, which is to estimate resource demand at runtime. Then, we can apply the model to improve system elasticity, which is our answer to **RQ3**, i.e., *How that assessment can be used to improve a system's QoS?*

Our strategy, presented in Chapter 5, targets dynamic resource management in clouds with special attention to drastic variations in the workload, which are not well handled by traditional strategies to provide elasticity. The two synthetic workloads we used in the experimental validation have that characteristic, focusing the validation on the scenario we give most attention. However, we also validated the proposal against a trace in which, most of the time, the workload exhibits smooth variations or remains relatively constant. This is valuable to verify the general applicability of the proposed strategy. The application of a strategy is less beneficial if it is only accurate in a specific condition – specially when this condition is unpredictable.

Generally, we found similar results in the experiments: the strategy that uses all the available resources in the estimations is more economic than the baseline, but delivers worse performance (with one exception in the case of the synthetic plateau); and availability-based strategies perform better than the baseline, but are costlier. We also observed that the performance gain/loss is much higher than the cost increase/decrease. Based on these relations, we consider that the availability-based approach is the most appropriate among the observed, in terms of cost/benefit. However, the final choice for the strategy to apply in a given service would, rely on the weight its stakeholders give to costs and performance or SLA assurance. One possibility would be to estimate the cost per SLA violation and make a decision based on that. For instance, suppose a loss of US$0.01 per SLA violation. Then, based on the data we obtained with the trace-driven experiments with 10 minutes of monitoring interval, we have that changing from the *single-step* (9% of SLA violations) to the *USL-limit* strategy (with no SLA violation), would increase revenues in over US$5,300.00, considering the 5,956,710 requests performed in the experiment. Additionally, considering the current cost of running the service in `t2.medium` instances at AWS, which is of US$0.047 per hour[1], this same change increases the cost from US$1.86 to US$2.64 for the five hours of execution.

Regarding the extension of the time slice to use to estimate the arrival rate, our rationale was that the shorter, most recent interval would provide a more precise estimate of the current workload. The difference is meaningful when applying it with the pool-size-based strategy. But, in combination with the availability-based strategy, there was only a small benefit in the 30 minutes monitoring interval. With monitoring interval of 10 minutes, the cost even increased a little. We experimented only the utilization of 20% of the monitoring interval as an option for the short-history estimate. Nevertheless, we suggest to consider the possibility of experimenting different time slices, specially for longer monitoring intervals.

[1] https://aws.amazon.com/ec2/pricing/on-demand/ accessed on 2107-03-29

6.2 Threats to validity

In this section, we consider the limitations regarding the empirical validations of the application of ANOVA test to verify scalability and the strategy for automated resource management of elastic systems.

6.2.1 ANOVA test for scalability evaluation

There are cases in which a system cannot maintain the same response time when it uses more nodes. Particularly, in this thesis, this is the case when data synchronization among the nodes increase the response time. Then, the application of the Analysis of Variance (ANOVA) is not suitable, since the response time is expected to be different at different scales. An alternative, though, is to make the analysis less strict, simply comparing the mean response time with an SLA. Then, we can attest that, even though the response time is changing, the system performance behaves adequately. It is important to notice that, in any case, there are limits after which a given system no longer scales [Law98, Gun07].

Additionally, the ANOVA test assumes the normal distribution of the population – in our case, the response times. Nevertheless, there is an alternative hypothesis test, the Kruskal-Wallis [KW52], with the same goal which is non-parametric, i.e., there is no assumption regarding probability distributions, and we plan to include an implementation of this test in the Scalability Explorer.

6.2.2 Dynamic resource management

The limitations of the validation of the resource management strategies are related to the facts, first, that we did not consider the time spent with setting up a new server when it is requested. In the cluster where we ran the experiments, the resources were preallocated. In elastic clouds, however, the resources might be dynamically requested and set up to be included in a service infrastructure. This procedure can be optimized with some practices, but not completely eliminated. Notwithstanding, we consider that the absence of node setup time is more beneficial to the utilization-based approach as it tends to cause more changes in the pool size. Thus, it should not favor our proposal in the comparison.

Another limitation is that we used constant simulated processing times, reducing the variability of the response times. This is an issue when we work with performance and scalability models. The model comprises a generalization of service behavior. More variance increases the frequency with witch the behavior, in some requests, deviates from the model. This is inherent to modeling, in general. Besides, there is no such model that fits all the software systems. One must consider whether USL is adequate to model a given system or whether there is a better choice.

We calculated the execution cost of the experiments in seconds. The granularity of the leasing times used by commercial cloud providers is currently much coarser. Thus, the application of the proposed runtime algorithm (Fig. 5.2) may actually incur in higher cost. An alternative for cost reduction is to mark the virtual machines to be released when the renewal of the payment is approaching and, on new requests, first check for marked VMs to unmark.

Additionally, we made the choice for pool size updates in steps of one, which is preferable due to better granularity, in the utilization-based strategy. But it is possible to define rules with higher pool size variations. They tend to increase the cost, because of the reduced granularity, but also should reduce the time for stabilization in case of drastic variations of the workload, due to higher leaps. Yet, the pool size variations are always of the same extent, and with higher step sizes, there are less possible pool sizes, reducing the flexibility of the strategy.

Chapter 7

Conclusion

In the beginning of the Internet popularization the anticipation of demands could be a fact of life or death to a business. Setting up a new server and deploying a system on it, integrated to the existing infrastructure, could take months. Hence, estimating future demands was a very relevant task. The best scenario, in case of demand reaching levels beyond infrastructure capacity, was that the IT infrastructure was limiting traffic and, consequently, incomes. But system overload can affect general user experience. Overloaded servers tend to deliver poor QoS; the requests may take a long time to complete, or even be dropped. But the simple acquisition of hardware in a volume that would prevent an overload also has caveats. The high acquisition and maintenance cost, if not accompanied by increasing revenues can lead to insolvency, i. e., the enterprise is not capable of paying its obligations.

This scenario started to change with the advent of cloud computing. It promoted a paradigm change. One of the fundamental shifts was that, instead of buying or contracting long-term leasing of servers, the companies could rent virtual resources in the short-term, in a pay-per-user basis. That is possible due to standardizations and automations in the process of setting up common infrastructural resources, and the advances in hardware virtualization. Than, cloud providers publish APIs through which their customers can request, receive, and release resources.

But this new infrastructure is of no benefit if the systems cannot take advantage of it. First, systems need to be scalable – their architecture must allow them to improve performance or capacity by attaching more resources. Second, systems should be elastic – an automated procedure is put in charge of dynamic resource management, eliminating the need for human monitoring and actuation.

The assessment of the scalability of such systems is mainly performed via either analytic modeling or experimentation. The former is considered quick and cheap. The models are constructions based on a theoretic ground that enables the extraction of information regarding performance and scalability. But building detailed models can incur in space-state-explosion, making the model too complex for practical analysis. The alternative, which is the strategy used in this thesis, is to run the system, expose it to a workload, make measurements of metrics of interest, and create a model from those data, based on statistic inference, abstracting of the system internals. The drawback of this strategy is related to the time and cost of setting up the system and running the workload.

The procedure for scalability experiments is commonly manual, but there are efforts toward automation. Generally, though, the tools and frameworks are limited to a small scope of possibilities in many aspects. With this scenario in view, our first contribution was the Scalability Explorer, a framework for the definition and execution of automatic scalability evaluation of distributed systems. It contains a set of hotspots in which the user can set the desired behavior for specific elements of common scalability experiments, such as load generation, resource management, and parameter scaling.

During the initial phase of our research, we identified limitations in how scalability is measured and evaluated. The more commonly applied techniques comprise the analysis of graphs or tables with selected metrics. Some researchers proposed metrics that calculate scalability ratios. In all those cases, if the system does not scale linearly, it is up to the user to decide margins of tolerance

to consider the system scalable.

We proposed an alternative method to evaluate scalability. The application of ANOVA is used to verify whether the mean value of several populations are equivalent. The application of this test relaxes the need for linearity and still eliminates arbitrary decisions regarding scalability assessment. It requires, though, that the tester find the relation between the workload and the performance. In the demonstration of this strategy, we used the computational complexity to find this relation. Another possibility to find those relations is applying scalability modeling.

Also, we observed limitations in the automation of resource management in elastic systems, similarly to what we identified in scalability evaluation. The common approach for elasticity automation is to define rules based on utilization metrics to trigger the requesting or releasing of resources. The application of utilization metrics is limiting because a high utilization may imply that the system needs more resources, but cannot quantify the need. Demand can grow indefinitely, but utilization will not exceed 100%. Thus, a fixed amount of resources is defined in a rule, and it may need to be triggered more than once until the system achieves a stable configuration. In addition, there is no rule or procedure to identify at which load level changes should happen.

Aiming at overcoming these limitations, we proposed the application of scalability modeling to estimate resource demand. We inverted the function defined by the Universal Scalability Law [Gun93, Gun07, Gun08] and defined an algorithm to make scaling decisions based on the model. In this manner, we were able to estimate the number of VMs required to support a given workload. In an empirical validation, we observed that this strategy can deliver better performance than the application of rules based on utilization.

7.1 Future work

The next step in this research is to work on the dynamic runtime refinement of the model to tackle deviations in the estimates. This is of interest specially for systems with low contention and coherency. Initial experiments at small scale could produce a model that makes poor predictions at workload levels far from those used in the estimates. Thus, we plan to work on a feedback loop that observes deviations of measured performance from estimates and uses the new measurements to update the model at runtime. Another possibility, regarding this issue, is to evaluate the applicability of different scalability models.

Other direction we consider is the elimination of the fixed monitoring interval. The intention is to make the estimation manager act when a need for update in the pool size is detected. This possibility can improve the SLA maintenance in case of variations of the workload in a small time slot. However the benefits may be limited by the time needed to set up new instances.

Another possibility of future work is on workload prediction. We can investigate the integration of our strategy for resource management with existing approaches to predict future load levels or investigate new strategies.

Regarding the Scalability Explorer, we are considering to define annotations to increase the flexibility of scalability test implementation. Additionally, we are planning updates to simplify the inclusion of other metrics in the `ReportData`. Currently, it is possible to add data manually in a `ReportData`. However, we plan to add a framework hotspot specifically for this task. Then, we can provide implementations to add data collected by common monitoring tool, such as sar and ganglia.

During the time we developed this research, some hot topics arose or achieved maturity, in computer science. Big Data and the Internet of Things (IoT) are two such topics. The former relies on huge infrastructure and computation power to store and process an enormous amounts of data, finding in cloud computing an appropriate base. The latter refers to a massive system of Internet enabled devices. It connects the physical world to the Internet, making possible the remote interaction of humans with remove devices, and among devices themselves. IoT strongly relies on cloud computing and big data technologies. Additionally, we can see a massification of Internet access through mobile devices. Hence, many companies target those devices as the main mean to

interact with their solutions. Due to the limited processing and storage capacity of mobile devices, the systems are designed with thin clients for user interaction and cloud based back-ends for storage and processing. Thus, we can see the magnitude of cloud computing for IT strategies, and highlight the importance of applying an adequate strategy for resource management.

Bibliography

[ACP12] M. Andreolini, M. Colajanni, and M. Pietri. A scalable architecture for real-time monitoring of large information systems. In *IEEE Second Symposium on Network Cloud Computing and Applications*, 2012. 21

[AHA13] A.M.D. Aljohani, D.R.W. Holton, and I. Awan. Modeling and Performance Analysis of Scalable Web Servers Deployed on the Cloud. *2013 Eighth International Conference on Broadband and Wireless Computing, Communication and Applications*, pages 238–242, October 2013. 23

[All90] OA Allen. *Probability, statistics, and queueing theory with computer science applications*. Academic Press, second edition, 1990. 11

[Alp16] Alphabet. Annual report. https://abc.xyz/investor/, 2016. Accessed: 2016-11-30. 1

[Ama16a] Amazon. 2015 annual report. http://phx.corporate-ir.net/phoenix.zhtml?c=97664&p=irol-reportsannual, 2016. Accessed: 2016-11-28. 1

[Ama16b] Amazon. Q3 2016 financial results. http://phx.corporate-ir.net/phoenix.zhtml?c=97664&p=irol-reportsother, 2016. Accessed: 2016-12-26. 7

[Amd67] GM Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring joint computer conference*, pages 483—-485, 1967. 8, 12, 13, 17

[ASZ+10] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, 2010. 7

[Bar11] Angela Bartels. Data center evolution: 1960 to 2000. https://blog.rackspace.com/datacenter-evolution-1960-to-2000, 2011. Accessed: 2016-12-01. 2

[BDIS04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004. 11

[BJHG04] DA Bacigalupo, SA Jarvis, L He, and Nudd. GR. An investigation into the application of different performance prediction techniques to e-commerce applications. In *Proc. of the 18 th International Parallel and Distributed Processing Symposium (IPDPS)*, volume 00, 2004. 11, 12

[BKKL09] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather tomorrow ? Towards a Benchmark for the Cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, 2009. 8

[BMKM12] Felipe Besson, Paulo Moura, Fabio Kon, and Dejan Milogicic. Rehearsal: a framework for automated testing of web service choreographies. In *Proceedings of the 3rd Brazilian Conference on Software: Theory and Practice*, 2012. 4

[BMKM15] Felipe Besson, Paulo Moura, Fabio Kon, and Dejan Milojicic. Bringing test-driven development to web service choreographies. *Journal of Systems and Software*, 99:135–154, 2015. 4

[BPcV12] Lee Badger, Robert Patt-corner, and Jeff Voas. Cloud Computing Synopsis and Recommendations of the National Institute of Standards and Technology. *Nist Special Publication*, 800(146):81, 2012. 6, 9

[BYV+09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009. 6

[CB02] G. Casella and R.L. Berger. *Statistical Inference*. The Wadsworth & Brooks/Cole Statistics/Probability series. Wadsworth & Brooks/Cole Advanced Books & Software, 2nd edition, 2002. 4, 30, 37

[CBNEB12] B.F. Cornea, J. Bourgeois, T.T. Nguyen, and D. El-Baz. Scalable performance predictions of distributed peer-to-peer applications. In *IEEE 14th International Conference on High Performance Computing and Communication, IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 193 –201, june 2012. xi, 17, 18

[CDKB13] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Sistemas Distribuídos - 5ed: Conceitos e Projeto*. Bookman Editora, 2013. 5

[CEM10] C Chapman, W Emmerich, and Fg Marquez. Elastic service management in computational clouds. *CloudMan 2010*, pages 1–8, 2010. 23

[CMA13] Matheus Cunha, Nabor Mendonc, and Sampario Américo. Cloud Crawler : Um Ambiente Programável para Avaliar o Desempenho de Aplicações em Nuvens de Infraestrutura. In *31st Brazilian Symposium on Computer Networks and Distributed Systems*, 2013. 22

[CS06] Yong Chen and Xian-he Sun. STAS: A Scalability Testing and Analysis System. *2006 IEEE International Conference on Cluster Computing*, pages 1–10, 2006. 12, 17, 21, 23, 63

[CW00] Marc Courtois and Murray Woodside. Using regression splines for software performance analysis. In *Proceedings of the second international workshop on Software and performance - WOSP '00*, pages 105–114, 2000. 12

[dASLTV08] E.C. de Almeida, G. Sunye, Y. Le Traon, and P. Valduriez. A framework for testing peer-to-peer systems. In *19th International Symposium on Software Reliability Engineering*, pages 167 –176, nov. 2008. 22

[DPC10] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th international conference on World wide web - WWW '10*, New York, New York, USA, 2010. 11, 24

[DPC11] Jiang Dejun, Guillaume Pierre, and CH Chi. Resource Provisioning of Web Applications in Heterogeneous Clouds. In *USENIX Conference on Web Application Development*, 2011. 12, 24

[DRD+16] Rodrigo Da Rosa Righi, Vinicius Facco Rodrigues, Cristiano André Da Costa, Guilherme Galante, Luis Carlos Erpen De Bona, and Tiago Ferreto. Autoelastic: Automatic resource elasticity for high performance applications in the cloud. *IEEE Transactions on Cloud Computing*, 4(1):6–19, 2016. 10

[DRW07] Leticia Duboc, David Rosenblum, and Tony Wicks. A framework for characterization and analysis of software system scalability. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 375, 2007. 8

[Fac16] Facebook. Reports third quarter 2016 results. https://investor.fb.com/investor-news/2016/default.aspx, 2016. Accessed: 2016-11-25. 1

[GD12] Guilherme Galante and Luis Carlos E De Bona. A survey on cloud computing elasticity. In *Proceedings - 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing, UCC 2012*, pages 263–270, 2012. 9

[GGK93] Ananth Y Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel & Distributed Technology*, 1(3):12–21, 1993. 8, 12

[GGW10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: PRedictive Elastic reSource Scaling for cloud systems. In *Proceedings of the 2010 International Conference on Network and Service Management, CNSM 2010*, pages 9–16, 2010. 24

[GPBT11] Jerry Gao, Pushkala Pattabhiraman, Xiaoying Bai, and W. T. Tsai. SaaS performance and scalability evaluation in clouds. *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, pages 61–71, December 2011. 17

[GTL+12] Xavier Guerin, Wei Tan, Yanbin Liu, Seetharami Seelam, and Parijat Dube. Evaluation of Multi-core Scalability Bottlenecks in Enterprise Java Workloads. *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 308–317, aug 2012. 8

[Gun93] NJ Gunther. A Simple Capacity Model of Massively Parallel Transaction Systems. In *CMG-CONFERENCE*, 1993. 13, 68

[Gun07] NJ Gunther. *Guerrilla Capacity Planning*. Springer, Berlin Heidelberg, 2007. 8, 14, 15, 44, 65, 68

[Gun08] NJ Gunther. A General Theory of Computational Scalability Based on Rational Functions. *arXiv preprint arXiv:0808.1431*, pages 1–14, 2008. 11, 13, 68

[Gus88] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988. 12

[GWV05] Zao-Bin Gan, Deng-Wen Wei, and Vijay Varadharajan. Evaluating the Performance and Scalability of Web Application Systems. *Third International Conference on Information Technology and Applications (ICITA'05)*, 1:111–114, 2005. 12

[HDPT04] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing system*. John Wiley & Sons, 2004. 41

[HDV08] Ahmed Harbaoui, Bruno Dillenseger, and Jean-Marc Vincent. Performance characterization of black boxes with self-controlled load injection for simulation-based sizing. In *French Conference on Operating Systems (CFSE)*, 2008. 11, 24

[Hil90] Mark D Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, pages 18–21, 1990. 8

[HLM+10] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey. Early observations on the performance of Windows Azure. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, page 367, 2010. xi, 17, 19

[HPJ14] Thomas Heyman, Davy Preuveneers, and Wouter Joosen. Scalar: Systematic Scalability Analysis with the Universal Scalability Law. *2014 International Conference on Future Internet of Things and Cloud*, pages 497–504, 2014. 22, 23

[HSDV10] Ahmed Harbaoui, Nabila Salmi, Bruno Dillenseger, and Jean-Marc Vincent. Introducing Queuing Network-Based Performance Awareness in Autonomic Systems. *Sixth International Conference on Autonomic and Autonomous Systems*, pages 7–12, March 2010. 24

[HWSK10] Jens Happe, Dennis Westermann, Kai Sachs, and Lucia Kapová. Statistical Inference of Software Performance Models for Parametric Performance Completions. In *Research into PracticeâĂŞReality and Gaps*, pages 20—-35. Springer, 2010. 12

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and ModelingNo Title*. John Wiley \& Sons, 1991. 6, 11, 16

[JAP14] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Autonomic resource provisioning for cloud-based software. *SEAMS*, pages 95–104, 2014. 23

[Jay13] K R Jayaram. Elastic Remote Methods. In *Middleware*, pages 143–162, 2013. 23

[Jos07] Nicolai M. Josuttis. *SOA in Practice - The Art of Distributed Systems Design*. O'Reilly, 2007. 10

[JSM+12] Deepal Jayasinghe, Galen Swint, Simon Malkowski, Jack Li, Qingyang Wang, Junhee Park, and Calton Pu. Expertus: A Generator Approach to Automate Performance Testing in IaaS Clouds. *2012 IEEE Fifth International Conference on Cloud Computing*, pages 115–122, jun 2012. 22

[JW98] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 7, pages 524 –531 vol.7, jan 1998. 21

[JW00] Prasad Jogalekar and Murray Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):589–603, 2000. 8, 12, 17, 21, 63

[KB03] Samuel Kounev and Alejandro Buchmann. Performance modeling and evaluation of large-scale J2EE applications. *Int. CMG Conference*, 2003. 11

[KBW12] Markus Klems, David Bermbach, and René Weinert. A runtime quality measurement framework for cloud database service systems. In *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology*, pages 38–46. IEEE, September 2012. 8, 17, 22, 23

[KBXL10] Zhu Kenan, Yin Baolin, Chen Xi, and Cao Li. A highly scalable evaluation system with service-based architecture. In *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*, volume 1, pages 230 –233, march 2010. 21

[KM12a] M. Kamra and R. Manna. Performance of cloud-based scalability and load with an automation testing tool in virtual world. In *IEEE Eighth World Congress on Services (SERVICES)*, pages 57 –64, june 2012. 21

[KM12b] Madhvi Kamra and Ratnamala Manna. Performance of Cloud-Based Scalability and Load with an Automation Testing Tool in Virtual World. *2012 IEEE Eighth World Congress on Services*, pages 57–64, jun 2012. 8

[KMN04] a. Kamra, V. Misra, and E.M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. *Twelfth IEEE International Workshop on Quality of Service(IWQOS)*, pages 47–56, 2004. 7, 11

[KR12] James Kurose and Keith Ross. *Computer networking: A top–down approach - 6th edition*. Pearson, 2012. 5

[KW52] William H. Kruskal and W. Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952. 65

[Law98] Darren R. Law. Scalable means more than more: a unifying definition of simulation scalability. In *Winter Simulation Conference Proceedings*, pages 781 –788, dec 1998. 8, 65

[LBCP09] H.C. Lim, Shivnath Babu, J.S. Chase, and S.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, 2009. 23

[LLCK09] Jae Yoo Lee, Jung Woo Lee, Du Wan Cheun, and Soo Dong Kim. A Quality Model for Evaluating Software-as-a-Service in Cloud Computing, 2009. 12

[LLGK13] Leonardo Leite, Nelson Lago, Marco Gerosa, and Fabio Kon. Um middleware para encenação automatizada de coreografias de serviços web em ambientes de computação em nuvem. In *31st Brazilian Symposium on Computer Networks and Distributed Systems*, 2013. 31

[LLGZ13] Yifeng Luo, Siqiang Luo, Jihong Guan, and Shuigeng Zhou. A RAMCloud storage system based on HDFS: Architecture, implementation and evaluation. *Journal of Systems and Software*, 86(3):744 – 750, 2013. 17, 21

[LMC+14] Leonardo Leite, Carlos Eduardo Moreira, Daniel Cordeiro, Marco Gerosa, and Fabio Kon. Deploying large-scale service compositions on the cloud with the choreos enactment engine. In *IEEE 13th International Symposium on Network Computing and Applications (NCA)*, 2014. 31

[LW09] Huan Liu and Sewook Wee. Web server farm in the cloud: Performance evaluation and dynamic architecture. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 369–380, Berlin, Heidelberg, 2009. Springer-Verlag. xi, 17

[MA01] Daniel A. Menasce and Virgilio Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. 1, 21

[MCCS08] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Burstiness in Multi-Tier Applications: Symptoms, Causes, and New Models. In *ACM/IFIP/USENIX Middleware*, 2008. 11, 12

[MDLX05] J. Mizher, M.H. Dunham, Lin Lu, and Yongqiao Xiao. Scalability of oat. In *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on*, page 48, 2005. 21

[MK13] Paulo Moura and Fabio Kon. Automated Scalability Testing of Software as a Service. In *AST - 8th International Workshop on Automation of Software Test*, San Francisco, United States, 2013. 4

[MKVvS15] Paulo Moura, Fabio Kon, Spyros Voulgaris, and Maarten van Steen. Using performance forecasting to accelerate elasticity. In *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC)*, 2015. 4

[MKVvS16] Paulo Moura, Fabio Kon, Spyros Voulgaris, and Maarten van Steen. Dynamic resource allocation using performance forecasting. In *International Conference on High Performance Computing & Simulation (HPCS)*, 2016. 4

[MPS+15] Edwin L. C. Mamani, Lourenco A. Pereira, Marcos J. Santana, Regina H. C. Santana, Pedro Northon Nobile, and Francisco Jose Monaco. Transient performance evaluation of cloud computing applications and dynamic resource control in large-scale distributed systems. *2015 International Conference on High Performance Computing & Simulation (HPCS)*, pages 246–253, 2015. 23

[MVV13] Gabriele Mencagli, Marco Vanneschi, and Emanuele Vespa. Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments. *Proceedings of the 2013 International Conference on High Performance Computing and Simulation, HPCS 2013*, pages 11–18, 2013. 24

[Nah04] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004. 5

[Net16] Netflix. Q316 letter to shareholders. https://ir.netflix.com, 2016. Accessed: 2016-11-28. 1

[Pri08] Dan Pritchett. Base: an Acid Alternative. *ACM Queue*, 6(3):48–55, 2008. 10

[RVG+11] J Rehr, F Vila, J Gardner, L Svec, and M Prange. Scientific computing in the cloud. *Computing in Science Engineering*, PP(99):1, 2011. 17

[SAP11] N. Snellman, A. Ashraf, and I. Porres. Towards automatic performance and scalability testing of rich internet applications in the cloud. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 161 –169, 30 2011-sept. 2 2011. 21, 23

[Sch06] Theo Schlossnagle. *Scalable internet architectures*. Sams, Indianapolis, IN, USA, first edition, 2006. 9

[SDH+10] Nabila Salmi, Bruno Dillenseger, Ahmed Harbaoui, Jean-marc Vincent, and Orange Labs. Model-based Performance Anticipation in Multi-tier Autonomic Systems : Methodology and Experiments. *International Journal on Advances in Networks and Services*, 3(3):346–360, 2010. 24

[SEB12] Khaled Salah, Khalid Elbadawi, and Raouf Boutaba. Estimating service response time for elastic cloud applications. *2012 1st IEEE International Conference on Cloud Networking, CLOUDNET 2012 - Proceedings*, pages 12–16, January 2012. 23

[Smi16] David Mitchell Smith. Cloud computing primer for 2016. Technical report, Gartner, 2016. 7

[SN90] Xian-he Sun and Lionel M Ni. Another view on parallel speedup. In *Supercomputing '90 Proceedings of the 1990 ACM/IEEE conference on Supercomputing1*, pages 324–333, 1990. 13

[SR94] Xian-He Sun and Diane Rover. Scalability of parallel algorithm-machine combinations. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):599–613, jun 1994. 12

[SVRG13] Monika Simjanoska, Goran Velkoski, Sasko Ristov, and Marjan Gusev. Does the Performance Scale the Same as the Cost in the Cloud, 2013. 8, 12

[Szt10] János Sztrik. *Basic Queueing Theory*. 2010. 11

[TDP+13] Alain Tchana, Bruno Dillenseger, Noel De Palma, Xavier Etchevers, Jean-Marc Vincent, Nabila Salmi, and Ahmed Harbaoui. Self-scalable Benchmarking as a Service with Automatic Saturation Detection. In *Middleware 2013*, pages 389–404, 2013. 22, 23, 44

[THBG12] Wei-Tek Tsai, Yu Huang, Xiaoying Bai, and Jerry Gao. Scalable Architectures for SaaS. *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 112–117, apr 2012. 8

[TvS07] Andrew Tanenbaum and Maarten van Steen. *Sistemas distribuídos: princípios e paradigmas - 2ed.* Pearson, 2007. 5

[Twi16] Twitter. Q3 2016 letter to shareholders. https://investor.twitterinc.com, 2016. Accessed: 2016-11-25. 1

[UPS+05] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):291, June 2005. 11

[VNM+12] Nedeljko Vasić, Dejan Novaković, Svetozar Miucin, Dejan Kostić, and Ricardo Bianchini. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012. 23

[VRMB11] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011. 9

[W3C04] W3C. Web services architecture, February 2004. http://www.w3.org/TR/ws-arch/. 10

[WA02] Elaine J Weyuker and Alberto Avritzer. A metric to predict software scalability. In *Proceedings of the Eighth IEEE International Software Metrics Symposium (METRICS)*, volume 2002-Janua, pages 152–158, 2002. 12

[WG06] CB Weinstock and JB Goodenough. On system scalability. Technical Report March, Carnegie Mellon Software Engineering Institute, 2006. 9

[WWW+94] Jim Waldo, Jim Waldo, Geoff Wyant, Geoff Wyant, Ann Wollrath, Ann Wollrath, Sam Kendall, and Sam Kendall. A note on distributed computing. Technical report, IEEE Micro, 1994. 10

[XP09] Kaiqi Xiong and Harry Perros. Service Performance and Analysis in Cloud Computing. *2009 Congress on Services - I*, pages 693–700, July 2009. 11