Workload Characterization, Controller Design and Performance Evaluation for Cloud Capacity Autoscaling

Ahmed Ali-Eldin Hassan

PhD thesis, 2015
Department of Computing Science
Umeå University
SE-901 87 Umeå
Sweden

This work has been generously supported by the Swedish Government's strategic effort eSSENCE, the European Community's Seventh Framework Programme under grant agreement #257115 for the project OPTIMIS, the European Union Seventh Framework Programme under grant agreement 610711 for the project CACTOS, and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control. Part of the research was conducted using the resources of High Performance Computing Center North (http://www.hpc2n.umu.se/).

UMINF:15.09
ISSN: 0348-0542
ISBN: 978-91-7601-330-4

Electronic version available at http://umu.diva-portal.org/
Printed by Print & Media, Umeå University
Umeå, Sweden 2015

Abstract

This thesis studies cloud capacity auto-scaling, or how to provision and release resources to a service running in the cloud based on its actual demand using an automatic controller. As the performance of server systems depends on the system design, the system implementation, and the workloads the system is subjected to, we focus on these aspects with respect to designing auto-scaling algorithms. Towards this goal, we design and implement two auto-scaling algorithms for cloud infrastructures. The algorithms predict the future load for an application running in the cloud. We discuss the different approaches to designing an auto-scaler combining reactive and proactive control methods, and to be able to handle long running requests, e.g., tasks running for longer than the actuation interval, in a cloud. We compare the performance of our algorithms with state-of-the-art auto-scalers and evaluate the controllers' performance with a set of workloads. As any controller is designed with an assumption on the operating conditions and system dynamics, the performance of an auto-scaler varies with different workloads.

In order to better understand the workload dynamics and evolution, we analyze a 6-years long workload trace of the sixth most popular Internet website. In addition, we analyze a workload from one of the largest Video-on-Demand streaming services in Sweden. We discuss the popularity of objects served by the two services, the spikes in the two workloads, and the invariants in the workloads. We also introduce, a measure for the disorder in a workload, i.e., the amount of burstiness. The measure is based on Sample Entropy, an empirical statistic used in biomedical signal processing to characterize biomedical signals. The introduced measure can be used to characterize the workloads based on their burstiness profiles. We compare our introduced measure with the literature on quantifying burstiness in a server workload, and show the advantages of our introduced measure.

To better understand the tradeoffs between using different auto-scalers with different workloads, we design a framework to compare auto-scalers and give probabilistic guarantees on the performance in worst-case scenarios. Using different evaluation criteria and more than 700 workload traces, we compare six state-of-the-art auto-scalers that we believe represent the development of the field in the past 8 years. Knowing that the auto-scalers' performance depends on the workloads, we design a workload analysis and classification tool that assigns a workload to its most suitable elasticity controller out of a set of implemented controllers. The tool has two main components; an analyzer, and a classifier. The analyzer analyzes a workload and feeds the analysis results to the classifier. The classifier assigns a workload to the most suitable elasticity controller based on the workload characteristics and a set of predefined business level objectives. The tool is evaluated with a set of collected real workloads, and a set of generated synthetic workloads. Our evaluation results shows that the tool can help a cloud provider to improve the QoS provided to the customers.

Sammanfattning

Denna avhandling studerar autoskalning, det vill säga hur en regulator automatiskt kan allokera och avallokera resurser för en tjänst som körs i molnet beroende på tjänstens belastningsmönster. Prestandan för ett serversystem beror på systemets design, dess implementation och den belastning systemet utsätts för. Denna avhandling fokuserar på dessa aspekter för design av algoritmer för autoskalning. Två algoritmer för autoskaling i molninfrastrukturer designas och implementeras. Algoritmerna predikterar framtida belastningsmönster för en tjänst som körs i molnet. Olika ansatser till att designa en regulator som kombinerar reaktiva och proaktiva reglermetoder utvärderas. Vi designar även metoder för att hantera serverförfrågningar som tar lång tid att betjäna, det vill säga beräkningar som körs längre än regulatorns aktiveringsintervall.

Vi jämför våra algoritmer med existerande autoskalare från forskningsfronten och utvärderar dess prestanda med hjälp av olika belastningskurvor. Då alla regulatorer är designade med vissa antaganden om belastningönster och datorsystemets dynamik varierar en regulators prestanda för olika typer av belastningskurvor.

För att bättre förstå egenskaperna hos belastningskurvor och hur dessa utvecklas med tiden analyserar vi 6 år av loggdata från Wikipedia, den sjätte mest populära websidan på Internet. Vi analyserar även en belastningskurva från en av de största videostreaming-tjänsterna i Sverige. I dessa analyser studeras populariteten hos dataobjekt på dessa två tjänster, hur belastningstoppar ser ut samt invarianter i belastningen, exempelvis periodiska mönster som å terkommer varje dag eller vecka. Vi introducerar även ett volatilitetsmått för oordningen i en belastningkurva, vilket är baserat på metoder och mått som används för signalbehandling inom biomedicin. Det nya måttet kan användas för att karaktärisera belastningskurvor. Vi utvärderar det mot existerande mått från forskningslitteraturen och på visar fördelar.

För att bättre förstå avvägningar mellan olika autoskalare för olika belastningsmön ster designar vi ett ramverk för att jämföra autoskalare. Ramverket ger probabilistiska garantier för autoskalarnas prestanda i värsta-fall-scenarion. Med olika utvärderingskriteria och över 700 olika belastningskurvor jämför vi sex olika autoskalningsalgoritmer från litteraturen, vilka representerar forskningsområdets utveckling under de senaste åtta åren.

Då prestandan hos en autoskalare beror på belastningskurvan designar vi ett analys- och klassificeringsverktyg som matchar en belastningskurva till den mest lämpliga av en mängd tillgängliga autoskalare. Vilken autoskalare som väljs beror på belastningskurvans egenskaper och på förvalda preferenser såsom prestandakriterier. Verktyget utvärderas med både riktiga belastningskurvor och syntetiskt genererade belastningsmönster. Utvärderingen visar att verktyget i de allra flesta fall väljer den bäst lämpare autoskalaren och därmed att det kan hjälpa en molnleverantör att få mer responsiva tjänster och samtidigt minska resursanvändningen.

Acknowledgements

This is the end of a journey that started 30 years ago. I will thank people in chronological order of meeting. To my parents, thank you for giving me everything you had, and everything I have. Now that I am a father, I know what it is like to have a kid who loved to experiment with boiling water. Part of this journey was to make you proud! I love you! To my siblings, Maie, Yomna, and Hosam, I love you, you are great and you are all what anyone would want as siblings. I miss being the evil brother :). My first few experiments were done on you as subjects, and boy, they were fun!

To my wife, I met you first one week before starting my PhD. You were foolish enough to get engaged to me after having long interrogations, with you interrogating me :). You have written more of this thesis than I did with your support and love. This journey has been rough for you. I hope I will be able to pay back some of my debt. Thank you for believing in me, giving us our children, and dealing with my tantrums :). I love you and I look forward to our new adventures!

To my advisors, Erik and Johan, I still remember my interviews with you like it was yesterday. You have made this group among the best in the world. You have helped me build my career and be (hopefully) a good scientist. You are above all friends I cherish! It has been fun working with you! I am looking forward to all the great things we will do together in the future! Erik, you are awesome on so many levels, hard to sum up actually! Johan, your help when I was stuck in the beginning, and your always positive suggestions and feedback made this possible!

To my colleagues in the group, are not you the most awesome research group in the world? Thank you Lars, Peter, Petter, P-O, Mina, Ewnetu, Muyi, Amardeep, Viali, Jakub, Selome, Gonzalo, Abel, Luis, Cristian, Francisco, Tomas, Lennart, Daniel, Lei, and Christina. This is in no particular order. Each one of you made this a great place! To my department colleagues, well I can not thank you enough for making this a very special place for me! I will always remember this department as my academic birthplace. Keep up the great work. Special thanks go to my friends Emad and Mahmoud!

To my Collaborators, this thesis was a result of endless late-night work and discussions with you. Thank you Alessandro, Sara, Oleg, Maria, Tania, Amardeep, Ali, Stas, and Karl-Erik! Working with you has been a lot of fun and learning! Looking forward to continue these collaborations with less late-nights!

To my daughter, Salma, in our deen, having a daughter is a privilege. You light the days and nights. You will probably not read this until 10 years, but daddy loves you. To my son, Yusuf, my little man, you came to life with a miracle, you are a miracle who touched my life. Daddy loves you too! Both of you make my life meaningful. To Mohamed, Mariam and Abul-Rahman, I love you!

To the people who died in my country between January, 2011 (roughly when I started my PhD), until this day. Frequently, I envy you for being in a better place. I get the "why-not-me?" syndrome. May god bless your souls! I promise to live for your cause, and tell my children about you.

This part of the thesis should have been the longest part. I did my best to keep it short. Thank you!

Preface

This thesis consists of an introductory chapter, the following papers, and an appendix:

I. A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Proceedings of the 13th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 204-212, IEEE, 2012.

II. A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd workshop on Scientific Cloud Computing (ScienceCloud)*, pages 31-40, ACM, 2012.

III. A. Ali-Eldin, A. Rezaie, A. Mehta, S. Razroevy, S. Sjöstedt-de Luna, O. Seleznjev, J. Tordsson, and E. Elmroth. How will your workload look like in 6 years? analyzing wikimedia's workload, In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering (IC2E)*, pages 349-354, IEEE Computer Society, 2014.

IV. A. Ali-Eldin, O. Seleznjev, S. Sjöstedt-de Luna, J. Tordsson, and E. Elmroth. Measuring cloud workload burstiness, *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, pages 566 - 572, IEEE, 2014

V. A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Analysis and characterization of a Video-on-Demand service workload, *Proceedings of the 6th ACM Multimedia Systems Conference*, pages 189-200, ACM, 2015.

VI. A. Papadopoulos, A. Ali-Eldin, J. Tordsson, K.E. Årzén, and E.Elmroth. PEAS: A Performance Evaluation framework for Auto-Scaling strategies in cloud applications. *Submitted for Journal Publication.*

VII. A. Ali-Eldin, J. Tordsson, E. Elmroth, and M. Kihl. WAC: A Workload analysis and classification tool for automatic selection of cloud auto-scaling methods. *To be submitted.*

Introduction

1 Introduction

Cloud computing is a computing model that allows better management, higher utilization and reduced operating costs for datacenter operators while providing on-demand resource provisioning for multiple customers. Datacenters are often enormous in size and complexity. This complexity is further increased because of the multitude of (complex) hosted applications [47], making datacenter management a very complex task [11]. The management of a cloud datacenter is too complex for a human to manage and automated management systems are required. This thesis studies automated cloud elasticity management, one of the main datacenter management capabilities. Elasticity can be defined as the ability of cloud infrastructures to rapidly change *the amount of resources* allocated to an application in the cloud according to its demand. This work studies algorithms, techniques and tools that a cloud provider can use to automate dynamic resource provisioning allowing the provider to better manage the datacenter resources.

Performance of Internet-scale server systems like cloud systems depends on three main factors; design of the system, implementation of the system, and the load on the system [19]. The first two factors should be optimized during system design and implementation. The third factor should be optimized at run-time. Variations in system load can be attributed to either internal events in the application or external events. Internal events can occur in the servers' hardware, e.g., bad blocks on a solid state drive [43], the operating system, e.g., thread scheduling, or the applications running on the servers. Such events cause the occurrence of very short bottlenecks that reduce system performance considerably [56]. On the other hand, external events include, for example, changes in the characteristics of the workload volume [12] or the workload mix [51]. As an example, Internet traffic due to Michael Jackson's death resulted in disruptions in many major Internet services [46]. Nowadays, a drop in QoS is often reflected in a financial loss [15]. In the future, a decrease in the QoS might be reflected in a loss of a life, e.g., in case of self-driving cars relying on cloud systems [33].

The performance of networked systems has been studied since Kleinrock's early work on queuing theory and modeling in the 1970s [32], typically with an aim of optimizing server system design and implementation. Little focus has been given to optimizing run-time performance until 15 years ago when Chase et al. published their seminal work on controlling provisioned resources according to load variations [14]. A few years before Chase et al.'s paper, Foster and Kesselman introduced Grid computing [21]. These two papers were followed by Kephart's and Chess's paper on autonomic computing [31]. These advances coupled with advances in virtualization technologies [2,9], containers [40], server power management techniques [18] and increased network bandwidth has paved the way for the era of cloud computing [10] where elastic server infrastructures allow on-demand resource provisioning which is the focus of our dissertation.

1.1 What is Cloud Computing?

Cloud computing started as a technology hype-term to describe a lot of different systems leading to a lot of confusions on what the term refers to [8, 55]. The confusion was reduced when the US National Institute of Standards and Technology (NIST) defined and characterized the important aspects of cloud computing in a special publication [38]. NIST defined cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

NIST identifies in their cloud definition five essential characteristics for cloud services [38],

I. On-demand self service. Consumers can provision computing capabilities, e.g., server time and network storage, and control them automatically with no human interaction required.

II. Broad network access. The computing capabilities are available and accessible over the network through heterogeneous client platforms, e.g., mobile phones, tablets and laptops.

III. Resource pooling. Resources are pooled between multiple consumers to serve each customer's requirements.

IV. Rapid elasticity. Computing capabilities can be provisioned rapidly on-demand to deal with load changes. The provisioning can be in some cases automated.

V. Measured service. Usage can be monitored, controlled and reported transparently.

A key challenge to fulfill the cloud model is the scale of these systems. Current cloud datacenters are enormous in size. For example, on the 31st of December, 2014, Rackspace, a market leader in providing cloud computing services, had more than 112628 servers hosted in 9 datacenters serving more than 300000 customers [47]. A typical server has between 4 cores and 128 cores. Management of such huge and complex systems requires some automation in the management software. On-demand provisioning and resource pooling requires the middleware to be able to handle the provisioning demands for the customer and allocate the resources required by the service autonomically and transparently [31]

2 Cloud Capacity Auto-Scaling

Auto-scaling cloud resources, sometimes referred to as elasticity control or management, is an incarnation of the dynamic provisioning of server resources problem, a problem which has been studied for over a decade [13, 14]. For the past decade and

half, there has been tens – if not hundreds – of algorithms and techniques proposed in the literature tackling the same question; how to dynamically change the amount of provisioned resources for an application running on server systems according to the varying application's load [3, 7, 13, 14, 20, 22–29, 35–37, 39, 44, 45, 49–53]?

Horizontal elasticity is the property of a cloud datacenter to rapidly vary the number of VMs allocated to a service according to demand. Vertical elasticity is the property of a cloud datacenter to rapidly change configurations of virtual resources allocated to a service according to demand, e.g., adding more CPU power, memory or disk space to already running VMs. Autoscalers utilize the elasticity properties of a cloud by dynamically allocating sufficient capacity to a running service to maintain an acceptable level of QoS at reduced costs [54]. This dissertation focuses on automated horizontal auto-scaling of cloud resources based on workload dynamics, i.e., we focus on auto-scalers that change the number of VMs allocated to a service running in the cloud.

2.1 Auto-Scaling Controller Requirements

We have identified five key requirements for an auto-scaler for cloud resources. These requirements, in our opinion, are essential to make the auto-scaler useful and safe to use. The requirements are;

I. Scalability: It has been estimated in 2014 that Amazon's EC2 operates around one and half million servers running millions of virtual machines [42]. One single service can have up to a few thousand virtual machines [16]. Some services run in the cloud for a short period of time while other services can run for years , e.g., reddit has been running on EC2 entirely since 2009 [48]. Algorithms used for automated elasticity must be scalable with respect to the amount of resources running, the monitoring data analyzed and the time for which the algorithm has been running.

II. Adaptiveness: Workloads of Internet and cloud applications are dynamic [30]. An automated elasticity controller should be able to adapt to the changing workload dynamics or changes in the system models, e.g., new resources added or removed from the system. This should be done while reducing both *overprovisioning and underprovisioning* of resources.

III. Rapid: An automated elasticity algorithm should compute the required capacity rapidly enough to preserve the QoS requirements. Sub-optimal configurations that preserve the QoS requirements are better than any optimal configuration taking longer to compute than the time that can be tolerated by the users or the application to preserve the QoS required. Limited lookahead control is a very accurate technique for the estimation of the required resources but according to one study, it requires almost half an hour to come up with an accurate solution for 15 physical machines each hosting 4 Virtual Machines (VMs) [34].

IV. Robustness: Changing load dynamics might lead to a change in the controller behavior [41]. A robust controller should prevent oscillations in resource allocation and/or in performance with respect to the QoS requirements. While adaptiveness describes the ability of the controller to adapt to changing workloads, robustness describes the stability of the controller with changing workload and/or system dynamics. A controller might be able to adapt to the workload but with oscillations in resource allocation that results in application instability, e.g., adding and removing VMs to a traditional SQL database tier incurs high penalties with respect to consistency. Another controller might not be able to adapt to the changing workload, but is stable with changing workload or system dynamics, i.e., it is robust but not adaptive.

V. QoS and cost awareness: The automated elasticity algorithm should be able to vary the capacity allocated to a service according to demand while enforcing the QoS requirements. If the algorithm provisions less resources than required, then QoS may deteriorate, leading to possible losses. When the algorithm provisions extra capacity that is not needed by the service, then there is a waste of resources. In addition, the costs of the extra unneeded capacity increases the costs of running a service in the cloud.

2.2 Thesis Position Statement

The focus of our work started as an effort to design new auto-scaling algorithms for cloud applications. It became evident from the very start that while one can design a few hundred more auto-scaling algorithms using state-of-the-art techniques from different disciplines such as control theory, neural network, fuzzy logic, statistical inference, estimation theory, queuing theory, optimization and machine learning, there is a genuine lack of understanding of the dynamic provisioning problem stemming from the lack of understanding of the workloads. By lack of understanding of the problem we mean that most of the current work on auto-scaling is based on trial and error and empirical designs. While one can test a proposed algorithm under certain assumptions and conditions, and for a specific application, no one really knows how to generalize these results. Going back to the opening statement of the thesis, the performance of the system and the proposed auto-scaling algorithm depends on the system design, implementation and workloads.

Our position is that *deeper understanding of cloud workloads is needed to tackle and solve the dynamic resource provisioning problem.* Workloads can be classified into different classes based on their quantitative, and qualitative characteristics with respect to an application. For example, some workloads are bursty, e.g., the load on the Bulgarian Wikitionary project, a project hosted by the Wikimedia foundation [1], shown in Figure 1. If a traditional database system is subject to a bursty workload pattern where changing the amount of provisioned resources can come at a very high cost due to consistency and replication, then the auto-scaler should not for example cause oscillations in the resources provisioned.

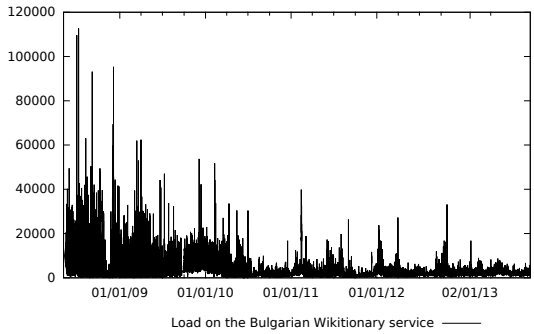


Figure 1: The Bulgarian Wikitionary project has bursty access patterns.

Our position is that *the vast space of the workloads and applications can be divided into smaller sub-spaces with each sub-space representing a category of workloads and applications with similar quantitative and qualitative characteristics.* The problem is then reduced to finding an auto-scaler for each category. The sub-spaces are multi-dimensional, with each dimension representing a feature or a characteristic of the workload or the application. Examples of workload features include periodicity, burstiness and the load-mix, i.e., the different request types [51]. Examples of application features include the amount of resources required to process one request from a certain type, the tolerance of the application to provisioning decisions, the application's ability to process delayed requests and the time required to process one request of a certain type. For example, a simple webserver application serving static content will require much less resources and processing time per request compared to an application that does optical character recognition online.

While this work does not define these categories, we show that they exist. Paper VII and Paper VIII show the varying performance of state-of-the-art auto-scaling algorithms and how one algorithm can be much better than another one for one workload or application compared to another one. Paper V introduces Sample Entropy, a measure for burstiness, one of the main features to classify workloads. Sample Entropy is used in Paper III, with other workload and application features, to classify different workloads using a k-Nearest-Neighbors classifier. The classification is used to assign each workload to the most suitable auto-scaling algorithm given some QoS constraints, e.g., on overprovisioning and underprovisioning. In papers I and II, we develop an auto-scaling algorithm capable of handling queued requests. In order to understand some of the cloud workloads, we provide deep analysis for two workloads, the workload on all Wikimedia foundation's projects including Wikipedia in Paper IV, and a Video-on-Demand workload in Paper VI.

2.3 Thesis Contributions

In support of our thesis statement, this thesis makes the following contributions towards a better understanding of workloads and elasticity controllers.

I. A comparison between all possible approaches to design a hybrid auto-scaler using a mix of reactive and proactive components for scaling. By a reactive component we mean a component that changes the future capacity using a set of predefined load thresholds, e.g., for every increase of 10 requests/second increase in the arrival rate, provision new resources. By a proactive we mean, a component capable of predicting the future load based on the patterns in the workload history. An auto-scaler was designed that predicts the future workload using the slope of the workload. The controller is further enhanced to take into account the queuing effects resulting from delayed requests.

II. Two comprehensive studies of two server system workloads. The first study is the longest server workload study we are aware of. In this study, we analyze the load on the Wikimedia foundation's servers that host among other things Wikipedia, Wikibooks and Wikitionary. The second is a study of the workload of TV4, a major Swedish Video-on-Demand (VoD) provider. These workloads are studied to provide us with a better understanding of how server systems evolve over time and thus enable us to design better auto-scalers.

III. A method to characterize and compare burstiness, a workload characteristic that affects the performance of an auto-scaler.

IV. Two comparative studies of some of the auto-scalers proposed in the literature. The first study uses scenario theory and chance constrained programming to provide some formal theoretical guarantees on the performance of different auto-scalers with different workloads. The second study compares the performance of a set of the published auto-scalers in a practical setting with different workloads.

V. A Workload Analysis and Classification tool for cloud infrastructures that acts as a meta-controller to select the most suitable auto-scaling algorithm for a workload. The tool analyzes the history of the running/new workloads and extracts some key characteristics. Workloads are then classified and assigned to the most suitable available elasticity auto-scaler based on the extracted characteristics and a set of user-defined Business-Level-Objectives (BLO), reducing the risk of bad predictions and improving the provided QoS.

3 Paper Summary and Future Work

This thesis consist of 7 papers and an appendix that support the thesis position. While most of this work was done assuming an infrastructure-as-a-Service cloud delivery model [8], the algorithms and techniques developed are suitable for all cloud delivery models. The contributions of each paper follows.

3.1 Paper I

The first paper in the thesis [7] introduces two proactive elasticity algorithms that can be used to predict future workload for an application running in the cloud. Resources are then provisioned according to the controllers' predictions. The first algorithm predicts the future load based on the workload's rate of change with respect to time. The second algorithm predicts future load based on the rate of change of the workload with respect to the average provisioned capacity. The two auto-scaling algorithms are explained and tested.

The paper also discusses the nine possible approaches to build hybrid elasticity controllers that have a reactive elasticity component and a proactive component. The reactive elasticity component is a step controller that reacts to the changes in the workload after they occur. The proactive component is a controller that with a prediction mechanism to predict future load based on the load's history. The two introduced controllers are used as the proactive component in the nine approaches discussed. Evaluation is performed using webserver traces. The performance of the resulting hybrid controllers are compared and analyzed. Best practices in designing hybrid controllers are discussed. The performance of the top performing hybrid controllers is compared to a state-of-the-art hybrid elasticity controller that uses a different proactive component. In addition, the effect of the workload size on the performance of the proposed controllers is evaluated. The proposed controller is able to reduce Service-Level-Agreement (SLA) violations by a factor of 2 to 10 compared to the state-of-the-art controller or a completely reactive controller.

3.2 Paper II

The design of the algorithms proposed in the first paper include some simplifying assumptions that ignore multiple important aspects of the cloud infrastructure and the workload's served. Aspects such as VM startup time, workload heterogeneity, and the changing request service rate of a VM are not considered in the first paper. In addition, it is assumed that delayed requests are dropped.

Paper II [4], extends on the first paper by enhancing the cloud model used for the controller design. The new model uses a G/G/N queuing model, where N is variable, to model a cloud service provider. The queuing model is used to design an enhanced hybrid elasticity controller that takes into account workload heterogeneity and the changing request service rate of a VM. The new controller is suitable for applications where buffering of delayed requests is possible. The controller also takes into account the size of the delayed requests when predicting the amount of resources required for the future load. The designed controller's performance is evaluated using webserver traces and traces from a cloud computing cluster with long running jobs. The results are compared to a controller that only has reactive components. The results show that the proposed controller reduces the cost of underprovisioning compared to the reactive controller at the cost of using more resources. The proposed controller requires a smaller buffer to keep all requests if delayed requests are not dropped.

3.3 Paper III

The third paper presents an analysis of a trace from the Wikimedia foundation spanning the period between May, 2008 till October, 2013 [17]. This is the largest web-workload analysis study we are aware of. Using descriptive statistics, time-series analysis, and polynomial splines, we study the trend and seasonality of the workload, its evolution over the years, and investigate patterns in page popularity. We show the effects that spikes on a Wikipedia page have on related Wikipedia pages. In addition, we show that the workload is highly predictable with a strong seasonality. We develop a short term prediction algorithm using splines that is able to predict the workload with a Mean Absolute Percentage Error of around 2%.

3.4 Paper IV

Sample Entropy (SampEn) is a technique that has been used in biomedical systems research for more than a decade to quantify disorder in biomedical signals. Our fourth paper introduces a modified version of the SampEn algorithm as a measure of burstiness in cloud workloads [6]. SampEn is has two main parameters. The first parameter defines how tolerant the measure is to small bursts. The second parameter defines the subset of the workload in which the algorithm searches for similarities. We show that SampEn ordering of workloads is robust against a wide selection of these two parameters. We compared the proposed algorithm to some of the state-of-the-art burstiness measures and show that the modified version of SampEn is better suited for classifying burstiness.

3.5 Paper V

Since Video-on-Demand (VoD) and video sharing services account for a large percentage of the total downstream Internet traffic and cloud applications, we analyze and model a workload trace from a VoD service provided by a major Swedish TV broadcaster [5]. The trace contains over half a million requests generated by more than 20000 unique users. We show that the user and the session arrival rates for the TV4 workload do not follow a Poisson process. The arrival rate distribution is modeled using a log-normal distribution while the inter-arrival time distribution is modeled using a stretched exponential distribution. We observe the "impatient user" behavior where users abandon streaming sessions after minutes or even seconds of starting them. Both very popular videos and non-popular videos are specially affected by impatient users. This behavior seems to be an invariant in VoD workloads and is neither affected by the average bit-rate nor by the number of videos a user watches. We discuss the spikes in the workload and their cause and show that spikes are very hard to predict in many cases.

3.6 Paper VI

In Paper VI, we introduce a method able to compare the performance of auto-scaling algorithms and provide probabilistic guarantees on their performance. The method is based on both robust control theory and stochastic control theory. The evaluation is formulated as a *chance constrained optimization problem*, which is solved using *scenario theory*. The adoption of such a technique allows one to give probabilistic guarantees on the obtainable performance of the controllers in the presence of uncertainties. Six different state-of-the-art auto-scaling algorithms have been selected from the literature for extensive test evaluation, and compared using the proposed framework. We build a discrete event simulator and parameterize it based on real experiments. Using the simulator, each auto-scaler's performance is evaluated using 796 distinct real workload traces from projects hosted on the Wikimedia foundations' servers, and their performance is compared. The evaluation is carried out using different performance metrics, highlighting the flexibility of the framework, while providing probabilistic bounds on the evaluation and the performance of the algorithms. Our results highlight the problem of generalizing the conclusions of the original published studies.

3.7 Paper VII

As it is not possible to design an autoscaler with good performance for all workloads and all scenarios as discussed in Paper VI, Paper VII proposes WAC, a Workload Analysis and Classification tool for automatic selection of a number of implemented cloud auto-scaling methods. . The tool has two main components, the analyzer and the classifier. The analyzer extracts two key features from the workload, periodicity and burstiness. Periodicity is measured using the autocorrelation of the workload, one of the standard methods for measuring periodicity. Burstiness is measured using the modified SampEn algorithm proposed in Paper IV. The classifier component uses a k-Nearest-Neighbors (kNN) algorithm to assign a workload to the most suitable elasticity controller based on the results from the analysis. The classifier requires training using training data. Four different training datasets are used for the training. The first set consists of 14 real workloads, the second set consists of 55 synthetic workloads. The third set consists of the previous two sets combined. The last set of workloads consists of a 798 workloads to different Wikimedia foundation projects. The paper then describes the training of the classifier component and the classification accuracy and results. The results show that the tool is able to assign between 87% to 98.3% of the workloads to the most suitable controller.

4 Future Work

There are several directions identified for future work starting from this thesis, some of which already started while others are planned. We are working on identifying and defining the different main workload classes in a better way and design optimal or

near-optimal auto-scaling methods for each class using stochastic and robust control theory. We are developing a statistical framework that provides probabilistic guarantees on the performance of auto-scaling algorithms based on their error models. We are working on a better statistical model for request arrival rates other than the Poisson process model. In addition, we are working on finding a bounded entropy measure for burstiness or disorder in signals that is able to provide a more solid mathematical definition than Sample Entropy.

References

[1] Page view statistics for Wikimedia projects. URL: dumps.wikimedia.org/other/pagecounts-raw/.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM SIGOPS Operating Systems Review*, pages 2–13. ACM, 2006.

[3] Ahmad Al-Shishtawy and Vladimir Vlassov. ElastMan: Autonomic elasticity manager for cloud-based key-value stores. In *Proc. 22nd Int. Symposium on High-performance Parallel and Distributed Computing*, HPDC 13, pages 115–116, 2013.

[4] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*, ScienceCloud '12, pages 31–40, New York, NY, USA, 2012. ACM.

[5] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. Analysis and characterization of a video-on-demand service workload. In *Proceedings of the 6th ACM Multimedia Systems Conference*, MMSys '15, pages 189–200, New York, NY, USA, 2015. ACM.

[6] Ahmed Ali-Eldin, Oleg Seleznjev, Sara Sjöstedt-de Luna, Johan Tordsson, and Erik Elmroth. Measuring cloud workload burstiness. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 566–572, Washington, DC, USA, 2014. IEEE Computer Society.

[7] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *IEEE Network Operations and Management Symposium*, NOMS 12, pages 204–212, April 2012.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[10] Jeff Barr, J Varia, and M Wood. Amazon ec22 beta. Amazon Web Services Blog. Accessed March, 2015, 2006. Link:`https://aws.amazon.com/blogs/aws/amazon_ec2_beta/`.

[11] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[12] Peter Bodik, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *ACM SoCC*, pages 241–252. ACM, 2010.

[13] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Quality of ServiceâIWQoS 2003*, pages 381–398. Springer, 2003.

[14] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. Managing energy and server resources in hosting centers. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 103–116. ACM, 2001.

[15] Paolo Costa, Thomas Zahn, Ant Rowstron, Greg O'Shea, and Simon Schubert. Why should we integrate services, servers, and networking in a data center? In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 111–118, New York, NY, USA, 2009. ACM.

[16] CycleComputing. New CycleCloud HPC Cluster Is a Triple Threat, September 2011. http://blog.cyclecomputing.com/2011/09/new-cyclecloud-cluster-is-a-triple-threat-30000-cores-massive-spot-instances-grill-chef-monitoring-g.html.

[17] Ahmed Ali Eldin, Ali Rezaie, Amardeep Mehta, Stanislav Razroev, Sara SjoÌstedt-de Luna, Oleg Seleznjev, Johan Tordsson, and Erik Elmroth. How will your workload look like in 6 years? analyzing wikimedia's workload. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 349–354. IEEE, 2014.

[18] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *Power-Aware Computer Systems*, pages 179–197. Springer, 2003.

[19] Dror Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[20] Hector Fernandez, Guillaume Pierre, and Thilo Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. In *IEEE Int. Conf. on Cloud Engineering*, IC2E 14, 2014.

[21] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, 1997.

[22] Alessio Gambi, Mauro Pezze, and Giovanni Toffetti. Kriging-based self-adaptive cloud controllers. *Services Computing, IEEE Transactions on*, PP(99):1–1, 2015.

[23] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, model-driven autoscaling for cloud applications. In *Proc. 11th Int. Conf. on Autonomic Computing*, ICAC 14, pages 57–64, 2014.

[24] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. AutoScale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, November 2012.

[25] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proc. Int. Conf. on Network and Service Management*, CNSM 10, pages 9–16, Oct 2010.

[26] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *Proc. 4th ACM/SPEC Int. Conf. on Performance Engineering*, ICPE 13, pages 187–198, 2013.

[27] Waheed Iqbal, Matthew N Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Gener. Comput. Syst.*, 27(6):871–879, 2011.

[28] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1):155–162, 2012.

[29] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proc. 6th Int. Conf. on Autonomic Computing*, ICAC 09, pages 117–126, 2009.

[30] Xiaozhu Kang, Hui Zhang, Guofei Jiang, Haifeng Chen, Xiaoqiao Meng, and Kenji Yoshihira. Understanding internet video sharing site workload: A view from data center design. *Journal of Visual Communication and Image Representation*, 21(2):129–138, 2010.

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

[32] Leonard Kleinrock. *Computer applications, volume 2, queueing systems*. Wiley, 1976.

[33] Swarun Kumar, Shyamnath Gollakota, and Dina Katabi. A cloud-assisted design for autonomous driving. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 41–46. ACM, 2012.

[34] Dara Kusic, Jeffrey O Kephart, James E Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12(1):1–15, 2009.

[35] Palden Lama and Xiaobo Zhou. Efficient server provisioning with control for end-to-end response time guarantee on multitier clusters. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):78–86, Jan 2012.

[36] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, pages 1–10. ACM, 2010.

[37] A. Hasan Mahmud, Yuxiong He, and Shaolei Ren. Bats: Budget-constrained autoscaling for cloud performance optimization. *SIGMETRICS Perform. Eval. Rev.*, 42(1):563–564, June 2014.

[38] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[39] Shicong Meng, Ling Liu, and Vijayaraghavan Soundararajan. Tide: achieving self-scaling in virtualized datacenter management middleware. In *Proc. 11th Int. Middleware Conf. Industrial Track*, Middleware Industrial Track 10, pages 17–22, 2010.

[40] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[41] Manfred Morari. Robust stability of systems with integral control. *IEEE Transactions on Automatic Control*, 30(6):574–577, 1985.

[42] Timothy Prickett Morgan. A Rare Peek Into The Massive Scale of AWS. Accessed: August, 2015, http://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/.

[43] Mark Moshayedi and Patrick Wilkison. Enterprise SSDs. *Queue*, 6(4):32–39, 2008.

[44] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for Infrastructure-as-a-Service. In *Proc. 10th Int. Conf. on Autonomic Computing*, ICAC 13, pages 69–82, 2013.

[45] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[46] Linnie Rawlinson and Nick Hunt. Jackson dies, almost takes internet with him. *CNN, Jun*, 2009. Acessed, March, 2015.

[47] Rackspace Investor Relations. Rackspace Reports Strong Fourth Quarter 2014 Results, Press Release. `http://phx.corporate-ir.net/phoenix.zhtml?c=221673&p=irol-newsArticle&ID=2017418`, February 2015.

[48] Amazon AWS. AWS Case Study: reddit. Accessed: May, 2013, http://aws.amazon.com/solutions/case-studies/reddit/.

[49] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. 2011 IEEE 4th Int. Conf. on Cloud Computing*, CLOUD 11, pages 500–507, 2011.

[50] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proc. 2nd ACM Symposium on Cloud Computing*, SOCC 11, pages 5:1–5:14, 2011.

[51] Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. 7th Int. Conf. on Autonomic Computing*, ICAC 10, pages 21–30, 2010.

[52] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, and Pawan Goyal. Dynamic provisioning of multi-tier internet applications. In *Proc. 2nd Int. Conf. on Autonomic Computing*, ICAC 05, pages 217–228, 2005.

[53] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1:1–1:39, 2008.

[54] 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.

[55] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[56] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien-An Lai, Chien-An Cho, Yuji Nomura, and Calton Pu. Lightning in the cloud: A study of transient bottlenecks on n-tier web application performance. In *Conference on Timely Results in Operating Systems (TRIOS)*, Broomfield, CO, October 2014. USENIX Association.

Paper I

An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures

A. Ali-Eldin, J. Tordsson, and E. Elmroth

In Proceedings of the 13th IEEE/IFIP Network Operations and Management Symposium (NOMS), pages 204-212, IEEE, 2012.

An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures*

Ahmed Ali-Eldin†, Johan Tordsson†, and Erik Elmroth†

Abstract

Cloud elasticity is the ability of the cloud infrastructure to rapidly change the amount of resources allocated to a service in order to meet the actual varying demands on the service while enforcing SLAs. In this paper, we focus on horizontal elasticity, the ability of the infrastructure to add or remove virtual machines allocated to a service deployed in the cloud. We model a cloud service using queuing theory. Using that model we build two adaptive proactive controllers that estimate the future load on a service. We explore the different possible scenarios for deploying a proactive elasticity controller coupled with a reactive elasticity controller in the cloud. Using simulation with workload traces from the FIFA world-cup web servers, we show that a hybrid controller that incorporates a reactive controller for scale up coupled with our proactive controllers for scale down decisions reduces SLA violations by a factor of 2 to 10 compared to a regression based controller or a completely reactive controller.

1 Introduction

With the advent of large scale data centers that host outsourced IT services, cloud computing [21] is becoming one of the key technologies in the IT industry. A cloud is an elastic execution environment of resources involving multiple stakeholders and providing a metered service at a specified level of quality [16]. One of the major benefits of using cloud computing compared to using an internal infrastructure is the ability of the cloud to provide its customers with elastic resources that can be provisioned on demand within seconds or minutes. These resources can be used to

*The paper has been re-typeset to match the thesis style. The order of the subfigures in Figure 4 was changed during the re-typesetting for readability. Reproduced with permission of IEEE.

†Department of Computing Science, Umeå University, Sweden, email: {ahmeda, tordsson, elmroth}@cs.umu.se

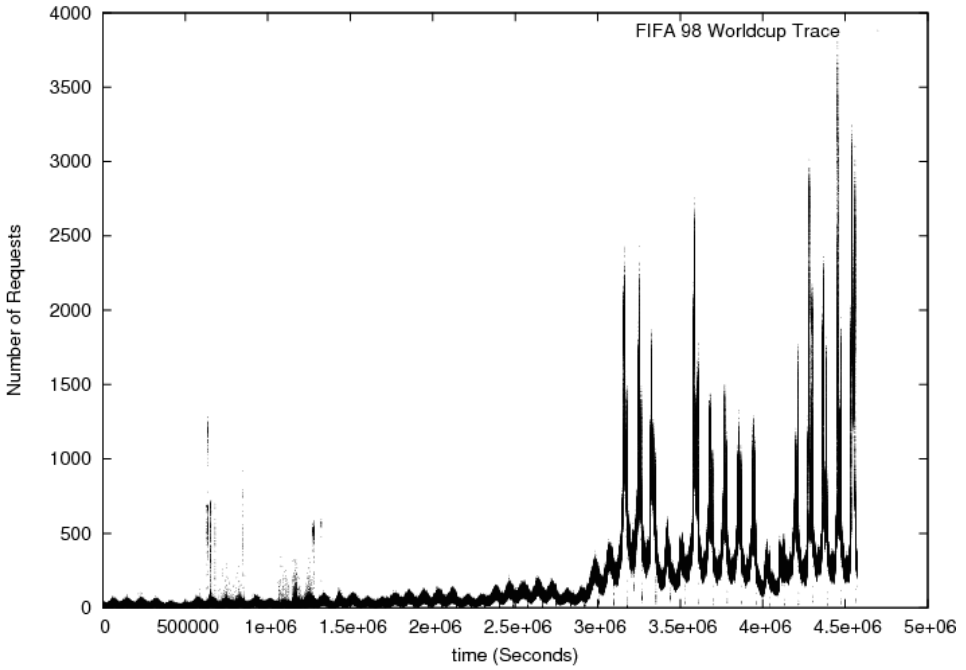


Figure 1: Flash crowds illustrating the rapid change in demand for the FIFA world cup website.

handle flash crowds. A flash crowd, also known as a slashdot effect, is a surge in traffic to a particular service that causes the service to be virtually unreachable [4]. Flash crowds are very common in today's networked world. Figure 1 shows the traces of the FIFA 1998 world cup website. Flash crowds occur frequently before and after matches. In this work, we try to automate and optimize the management of flash crowds in the cloud by developing an autonomous elasticity controller.

Autonomous elastic cloud infrastructures provision resources according to the *current* actual demand on the infrastructure while enforcing service level agreements (SLAs). Elasticity is the ability of the cloud to rapidly vary the allocated resource capacity to a service according to the current load in order to meet the quality of service (QoS) requirements specified in the SLA agreements. Horizontal elasticity is the ability of the cloud to rapidly increase or decrease the number of virtual machines (VMs) allocated to a service according to the current load. Vertical elasticity is the ability of the cloud to change the hardware configuration of VM(s) already running to increase or decrease the total amount of resources allocated to a service running in the cloud.

Building elastic cloud infrastructures that scale up and down with the actual demand of the service is a problem far from being solved [16]. Scale up should be fast enough in order to prevent breaking any SLAs while it should be as close as possible to the actual required load. Scale down should not be premature, i.e., scale down should occur when it is anticipated that the service does not need these resources in the near future. If scale down is done prematurely, resources are allocated and deallocated in a way that causes oscillations in the system. These resource oscillations introduce problems to load balancers and add some extra costs due to the frequent release and allocation of resources [14]. In this paper we develop two adaptive horizontal elasticity controllers that control scale up and scale down decisions and prevent resource oscillations.

This paper is organized as follows; in Section 2, we describe the design of our controllers. In Section 3 we describe our simulation framework, our experiments and discuss our results. In Section 4 we describe some approaches to building elasticity controllers in the literature. We conclude in Section 5.

2 Building an Elastic Cloud Controller

In designing our controllers, we view the cloud as a control system. Control systems are either closed loop or open loop systems [23]. In an open loop control system, the control action does not depend on the system output making open loop control generally more suited for simple applications where no feedback is required and no system monitoring is performed. Contrarily, a closed loop control system is more suited for sophisticated application as the control action depends on the system output and on monitoring some system parameters. The general closed-loop control problem can be stated as follows: The controller output $\mu(t)$ tries to force the system output $C(t)$ to be equal to the reference input $R(t)$ at any time t irrespective of the disturbance ΔD. This general statement defines the main targets of any closed loop control system irrespective of the controller design.

In this work, we model a service deployed in the cloud as a closed loop control system. Thus, the horizontal elasticity problem can be stated as follows: The elasticity controller output $\mu(t)$ should add or remove VMs to ensure that the number of service requests $C(t)$ is equal to the total number of requests received $R(t) + \Delta D(t)$ at any time unit t with an error tolerance specified in the SLA irrespective of the change in the demand ΔD while maintaining the number of VMs to a minimum. The model is simplified by assuming that servers start up and shut down instantaneously.

We design and build two adaptive proactive controllers to control the QoS of a service as shown in Figure 2. We add an estimator to adapt the output of the controller with any change in the system load and the system model.

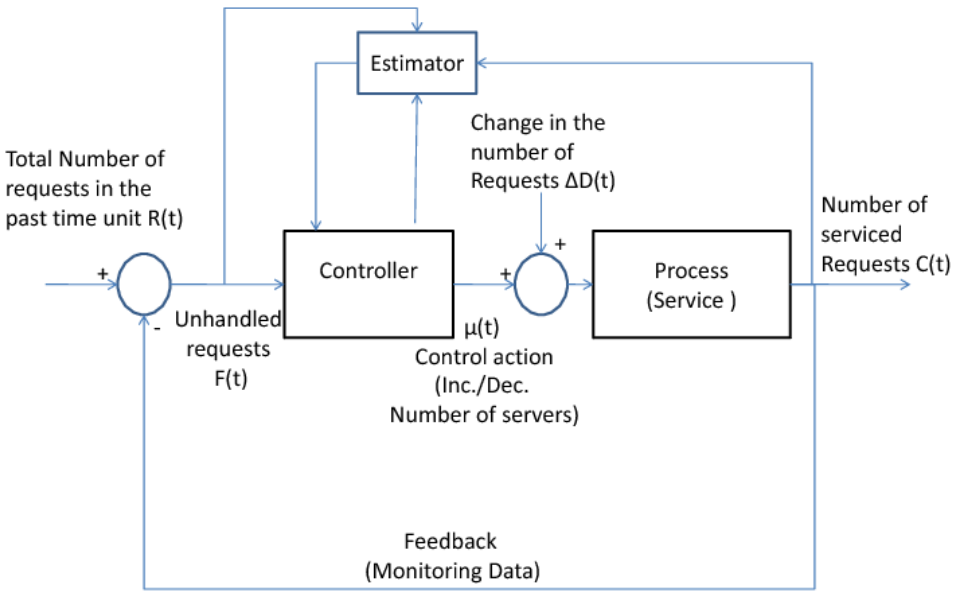


Figure 2: Adaptive Proactive Controller Model.

2.1 Modeling the state of the service

Figure 3 shows a queuing model representing the cloud infrastructure. The infrastructure is modeled as a $G/G/N$ stable queue in which the number of servers N required is variable [19]. In the model, the total number of requests per second R_{total} is divided into two inputs to the infrastructure, the first input $R(t)$ represents the total amount of requests the infrastructure is capable of serving during time unit t. The second input, ΔD represents the change in the number of requests from the past time unit. Since the system is stable, the output of the queue is the total service capacity required per unit time and is equal to R_{total}. P represents the increase or decrease in the number of requests to the current service capacity $R(t)$.

The goal of a cloud provider is to provide all customers with enough resources to meet the QoS requirements specified in the SLA agreements while reducing over provisioning to a minimum. The cloud provider monitors a set of parameters stated in the SLA agreements. These parameters represent the controlled variables for the elasticity controller. Our controllers are parameter independent and can be configured

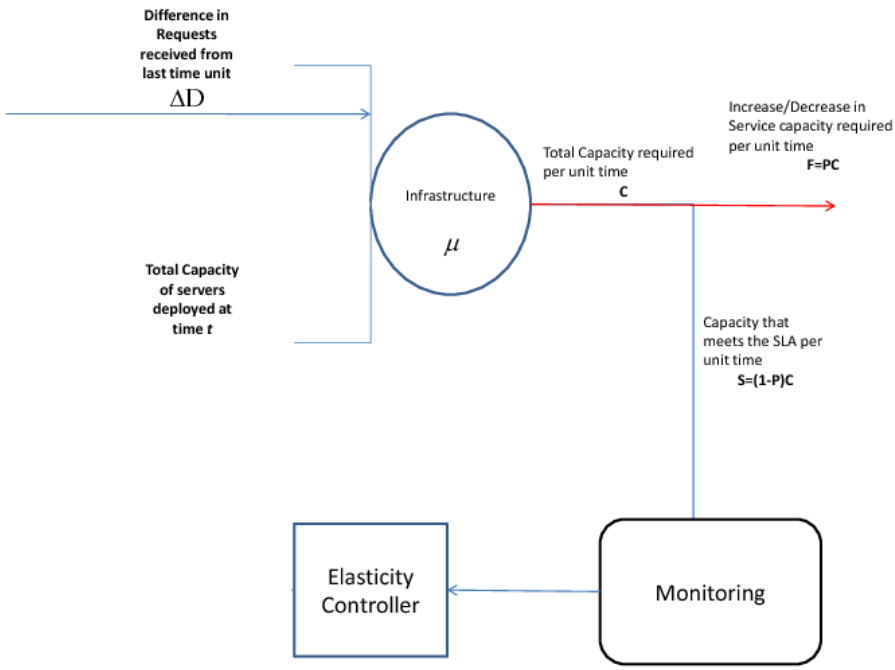


Figure 3: Queuing Model for a service deployed in the cloud.

to use any performance metric as the controlled parameter. For the evaluation of our controllers, we choose the number of concurrent requests received for the past time unit to be the monitored parameter because this metric shows both the amounts of over provisioned and under provisioned resources which is an indicator to the costs incurred due to the elasticity engine. Most of the previous work on elasticity considers response time to be the controlled parameter. Response time is software and hardware dependent and is not well suited for comparing the quality of different elasticity approaches [6].

2.2 Estimating future usage

From Figure 3, the total future service capacity required per unit time, $C(t+1)$, is $C(t+1) = \Delta D(t) + R(t)$, where $R(t)$ is the current service capacity and $\Delta D(t)$ is the change in the current service capacity required in order to meet the SLA agreement while maintaining the number of VMs to minimum. A successful proactive elasticity engine is able to estimate the change in future demand $\Delta D(t)$ and add or remove VMs

based on this proactive estimation. $\Delta D(t)$ can be estimated by

$$\Delta D(t) = P(t)C(t) \tag{1}$$

where $P(t)$ is the gain parameter of the controller. $P(t)$ is positive if there is an increase in the number of requests, negative if there is a decrease in the number of requests, or zero if the number of requests is equal to the current service capacity.

We define $\widehat{C}$ to be the infrastructure's average periodical service rate over the past T_d time units. $\widehat{C}$ is calculated for the whole infrastructure and not for a single VM. Thus, $\widehat{C} = \frac{\sum_i^{T_d} n_i t_i}{T_d}$, where T_d is a system parameter specifying the period used for estimating the average periodical service rate and t_i is the time for which the number of requests received per unit time for the whole infrastructure stay constant at n_i requests per unit time before the demand changes. Thus, $\sum_i^{T_d} t_i = T_d$. We also define $\overline{n}$, the average service rate over time as $\overline{n} = \frac{\sum_t n(t)}{T}$.

From equation 1 and since the system is stable ,

$$F = \widehat{C}\, P, \tag{2}$$

where F, the estimated increase or decrease of the load, is calculated using the gain parameter of the controller P every time unit. The gain parameter represents the estimated rate of adding or removing VMs. We design two different controllers with two different gain parameters.

For the first controller P_{C1}, the gain parameter P_1 is chosen to be the periodical rate of change of the system load,

$$P_1 = \frac{\Delta D_{T_d}}{T_D}. \tag{3}$$

As the workload is a non-linear function in time, the periodical rate of change of the load is the derivative of the workload function during a certain period of time. Thus, the gain parameter represents the load function changes over time.

For the second controller P_{C2}, the gain parameter P_2 is the ratio between the change in the load and the average system service rate over time,

$$P_2 = \frac{\Delta D_{T_d}}{\overline{n}}. \tag{4}$$

This value represents the load change with respect to the average capacity. By substituting this value for P in Equation 1, the predicted load change is the ratio between the current service rate and the average service rate multiplied by the change in the demand over the past estimation period.

2.3 Determining suitable estimation intervals

The interval between two estimations, T_d, represents the period for which the estimation is valid, is a crucial parameter affecting the controller performance. It is used for calculating $\widehat{C}$ for both controllers and for P_1 in the case of the first controller. T_d controls the controllers' reactivity. If T_d is set to one time unit, the estimations for the system parameters are done every time unit and considers only the system load during past time unit. At the other extreme, if T_d is set to ∞, the controller does not perform any predictions at all. As the workload observed in data centers is dynamic [11], setting an adaptive value for T_d that changes with the load dynamics is one of our goals.

We define K to be the tolerance level of a service i.e. the number of requests the service does not serve on time before making a new estimation, in other words,

$$T_d = \frac{K}{\widehat{C}}. \tag{5}$$

K is defined in the SLA agreement with the service owner. If K is specified to be zero, T_d should always be kept lower than the maximum response time to enforce that no requests are served slower by the system.

2.4 An elasticity engine for scale-up and scale-down decisions

The main goal of any elasticity controller is to enforce the SLAs specified in the SLA agreement. For today's dynamical network loads [4], it is very hard to anticipate when a flash crowd is about to start. If the controller is not able to estimate the flash crowd on time, many SLAs are likely to be broken before the system can adapt to the increased load.

Previous work on elasticity considers building hybrid controllers that combines reactive and proactive controllers [27] and [13]. We extend on this previous work and consider all possible ways of combining reactive and proactive controllers for scaling of resources in order to meet the SLAs. We define an elasticity engine to be an elasticity controller that considers both scale-up and scale-down of resources. There are nine approaches in total to build an elasticity engine using a reactive and a proactive controller. These approaches are listed in Table 1. Some of these combinations are intuitively not good, but for the sake of completeness we evaluate the results of all of these approaches. In order to facilitate our discussion, we use the following naming convention to name an elasticity engine; an elasticity engine consists of two controllers, a scale up (U) and a scale down (D) controller. A controller can be either reactive (R) or proactive (P). P_{C1} and P_{C2} are a special case from proactive controllers e.g. URP-DRP elasticity engine has a reactive and proactive

Table 1: *Overview of the nine different ways to build a hybrid controller.*

Engine Name	Scale up mechanism	Scale down mechanism
UR-DR	Reactive	Reactive
UR-DP	Reactive	Proactive
UR-DRP	Reactive	Reactive and Proactive
URP-DRP	Reactive and Proactive	Reactive and Proactive
URP-DR	Reactive and Proactive	Reactive
URP-DP	Reactive and Proactive	Proactive
UP-DP	Proactive	Proactive
UP-DR	Proactive	Reactive
UP-DRP	Proactive	Reactive and Proactive

controller for scale up and scale down while a UR-DP$_{C1}$ is an engine having a reactive scale up controller and P$_{C1}$ for scale down.

3 Experimental Evaluation

In order to validate the controllers, we designed and built a discrete event simulator that models a service deployed in the cloud. The simulator is built using Python. We used the complete traces from the FIFA 1998 world cup as input to our model [5]. The workload contains 1.3 billion Web requests recorded in the period between April 30, 1998 and July 26, 1998. We have calculated the aggregate number of requests per second from these traces. They are by far the most used traces in the literature. As these traces are quite old, we multiply the number of requests received per unit time by a constant in order to scale up these traces to the orders of magnitude of today's workloads. Although there are newer traces available such as the Wikipedia trace [26], but they do not have the number of peaks seen in the FIFA traces. We assume perfect load balancing and quantify the performance of the elasticity engines only.

3.1 Nine Approaches to build an elasticity engine

In this experiment we evaluate the nine approaches to a hybrid controller and quantify their relative performance using P$_{C1}$ and P$_{C2}$. We use the aggregate number of requests per unit time from the world cup traces multiplied by a constant equal to 50 as input to our simulator. This is almost the same factor by which the number of Internet users increased since 1997 [3]. To map the number of service requests to the number of servers, we assume that each server can serve up to 500 requests per unit time. This number is an average between the number of requests that can be handled by a Nehalem Server running the MediaWiki application [17] and a Compaq ProLiant

DL580 server running a database application [8]. We assume SLAs that specify the maximum number of requests not handled per unit time to be fewer than 5% of the maximum capacity of one server.

The reactive controller is reacting to the current load while the proactive controller is basing its decision on the history of the load. Whenever a reactive controller is coupled with a proactive controller and the two controllers give contradicting decisions, the decision of the reactive controller is chosen. For the UR-DR controller, scale down is only done if the number of unused servers is greater than two servers in order to reduce oscillations.

To compare all the different approaches, we monitor and sum the number of servers the controllers fail to provision on time to handle the increase in the workload, S^-. This number can be viewed as the number of extra servers to be added to avoid breaking all SLAs, or as the quality of estimation. $\overline{S^-}$ is the average number of requests the controller fails to provision per unit time. Similarly, we monitor the number of extra servers deployed by the infrastructure at any unit time. The summation of this number indicates the provisioned unused server capacity, S^+. $\overline{S^+}$ is the averaged value over time. These two aggregate metrics are used to compare the different approaches.

Table 2 shows the aggregate results when P_{C1} and P_{C2} are used for the proactive parts of the hybrid engine. The two right-most columns in the table show the values of S^- and S^+ as percentages of the total number of servers required by the workload respectively. We compare the different hybridization approaches with a UR-DR elasticity engine [24].

The results shown in the two tables indicate that using an UR-DP_{C2} engine reduces S^- by a factor of 9.1 compared to UR-DR elasticity engine, thus reducing SLA violations by the same ratio. This comes at the cost of using 14.33% extra servers compared to 1.4% in the case of a UR-DR engine. Similar results are obtained using a URP_{C2}-DP_{C2} engine. These results are obtained because the proactive scale down controller does not directly release resources when the load decreases instantaneously but rather makes sure that this decrease is not instantaneous. Using a reactive controller for scale down on the other hand reacts to any load drop by releasing resources. It is also observed that the second best results are obtained using an UR-DP_{C1} elasticity engine. This setup reduces S^- by a factor of 4, from 1.63% to 0.41% compared to a UR-DR engine at the expense of increasing the number of extra servers used from 1.4% to 9.44%.

A careful look at the table shows that elasticity engines with reactive components for both scale up and scale down show similar results even when a proactive component is added. We attribute this to the premature release of resources due to the reactivity component used for the scale down controller. The premature release of resources causes the controller output to oscillate with the workload. The worst performance is seen when a proactive controller is used for scale up with a reactivity component in

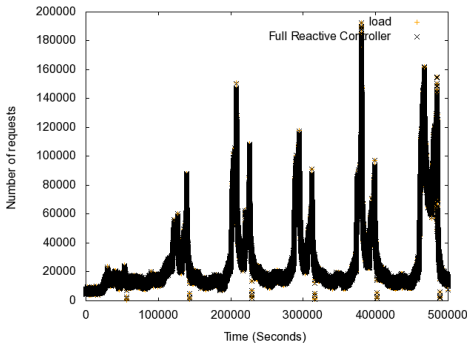
Table 2: S^- and S^+ for P_{C1} and P_{C2}

Name	S^-	$\overline{S^-}$	S^+	$\overline{S^+}$	S^-%	S^+%
UR-DR	-1407732	-0.3	120641	0.026	-1.63%	1.4%
P_{C1} results						
UR-DP_{C1}	-354814	-0.077	8159220	1.78	-0.41%	9.44%
UR-DRP_{C1}	-1412289	-0.3	1202806	0.26	-1.63%	1.4%
URP_{C1}-DRP_{C1}	-1411678	-0.3	1203170	0.26	-1.63%	1.4%
URP_{C1}-DR	-1407036	-0.3	1206391	0.26	-1.62%	1.4%
URP_{C1}-DP_{C1}	-354127	-0.077	8160627	1.78	-0.41%	9.4%
UP_{C1}-DP_{C1}	-4147953	-0.9	1827431	0.399	-4.8%	2.1%
UP_{C1}-DR	-8474040	-1.85	408447	0.399	-9.8%	2.1%
UP_{C1}-DRP_{C1}	-11408704	-2.49	190427.0	0.041	-10%	0.27%
P_{C2} results						
UR-DP_{C2}	-159029	-0.0347	12386346.0	2.7	-0.18%	14.33%
UR-DRP_{C2}	-1418949.0	-0.31	1176239.0	0.257	-1.64%	1.36%
URP_{C2}-DRP_{C2}	-1419269.0	-0.31	1175393.0	0.257	-1.64%	1.35%
URP_{C2}-DR	-1407732.0	-0.31	1206407.0	0.263	-1.63%	1.4%
URP_{C2}-DP_{C2}	-159029	-0.0347	12386346.0	2.707	-0.18%	14.33%
UP_{C2}-DP_{C2}	-4350841.0	-0.951	2216866.0	0.485	-5.03%	2.6%
UP_{C2}-DR	-11245521	-2.458	396697	0.0867	-13%	0.46%
UP_{C2}-DRP_{C2}	-11408704	2.49	190427	0.0416	-13.2%	0.22%

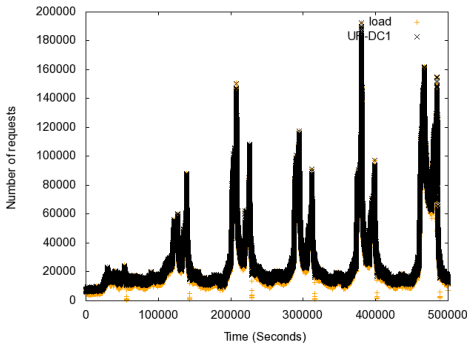
the scale down controller. This engine is not able to react to sudden workload surges. In addition it releases resources prematurely.

Figures 4(a), 4(b) and 4(e) shows the performance of a UR-DR, UR-DP_{C1} and a UR-DP_{C2} elasticity engines over part of the trace from 06:14:32, the 21^{st} of June,1998 to 01:07:51 27^{th} of June,1998. Figures 4(c), 4(d) and 4(f) shows an in depth view of the period between 15:50:00 the 23^{rd} of June, 1998 till 16:07:00 on the same day (between time unit 208349 and 209349 on the left hand side figures).

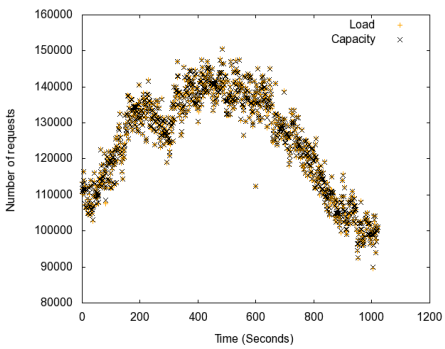
The UR-DR elasticity engine releases resources prematurely as seen in Figure 4(c). These resources are then reallocated when there is an increase in demand causing resource allocation and deallocation to oscillate. The engine is always following the demand but is never ahead. On the other hand, figures 4(d) and 4(f) show different behavior where the elasticity engine tries not to deallocate resources prematurely in order to prevent oscillations and to be ahead of the demand. It is clear in Figure 4(f) that the elasticity engine estimates the future load dynamics and forms an *envelope* over the load. An envelope is defined as the smooth curve that takes the general shape of the load's amplitude and passes through its peaks [12]. This delay in the



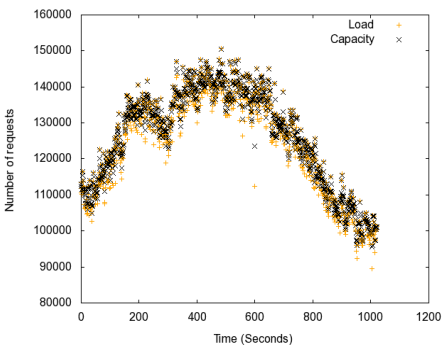
(a) UR-DR performance in a period of 6 days.



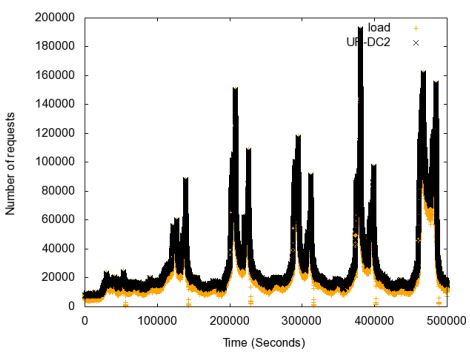
(b) UR-DP$_{C1}$ performance in a period of 6 days.



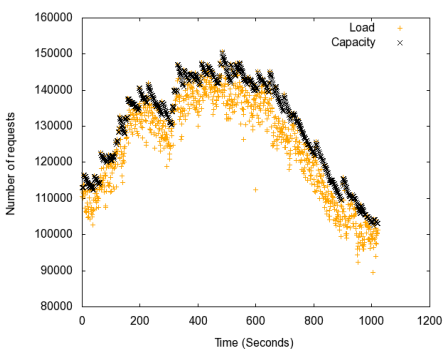
(c) UR-DR: Zooming on a period of 17 minutes.



(d) UR-DP$_{C1}$: Zooming on a period of 17 minutes.



(e) UR-DP$_{C2}$ performance in a period of 6 days.



(f) UR-DP$_{C2}$: Zooming on a period of 17 minutes.

Figure 4: Performance of UR-DR, UR-DP$_{C1}$ and, UR-DP$_{C2}$ elasticity engines with time: The Figures show how the different engines detect future load. It can be observed that the UR-DR engine causes the capacity to oscillate with the load while UR-DP$_{C1}$ and UR-DP$_{C2}$ predict the envelope of the workload.

deallocation comes at the cost of using more resources. These extra resources improve the performance of the service considerably as it will be always ahead of the load. We argue that this additional cost is well justified considering the gain in service performance.

3.1.1 Three classes of SLAs

An infrastructure provider can have multiple controller types for different customers and different SLA agreements. The results shown in table 2 suggest having three classes of customers namely, gold, silver and bronze. A gold customer pays more in order to get the best service at the cost of some extra over-provisioning and uses a UR-DP$_{C2}$ elasticity engine. A silver customer uses the UR-DP$_{C1}$ elasticity engine to get good availability while a bronze customer uses the UR-DR and gets a reduced, but acceptable, QoS but with very little over-provisioning. These three different elasticity engines with different degrees of over provisioning and qualities of estimation give cloud providers convenient tools to handle customers of different importance classes and thus increase their profit and decrease their penalties. Current cloud providers usually have a general SLA agreement for all their customers. RackSpace [2] for example guarantees 100% availability with a penalty equal to 5% of the fees for each 30 minutes of network or data center downtime for the cloud servers. It guarantees 99.9% availability for the cloud files. The network is considered not available in case of [2]: (i) The Rackspace Cloud network is down, or (ii) the Cloud FilesÂ service returns a server error response to a valid user request during two or more consecutive 90 second intervals, or (iii) the Content Delivery Network fails to deliver an average download time for a 1-byte reference document of 0.3 seconds or less, as measured by The Rackspace Cloud's third party measuring service. For an SLA similar to the RackSpace SLA or Amazon S3 [1], using one of our controllers significantly reduces penalties paid due to server errors, allowing the provider to increase profit.

3.2 Comparison with regression based controllers

In this experiment we compare our controllers with the controller designed by Iqbala et al. [13] who design a hybrid elasticity engine with a reactive controller for scale-up decisions and a predictive controller for scale-down decisions. When the capacity is less than the load, a scale up decision is taken and new VMs are added to the service. For scale down, their predictive component uses second order regression. The regression model is recomputed for the full history every time a new measurement data is available. If the current load is less than the provisioned capacity for k time units, a scale down decision is taken using the regression model. If the predicted number of servers is greater than the current number of servers, the result is ignored. Following our naming convention, we denote their engine UR-DRegression. As

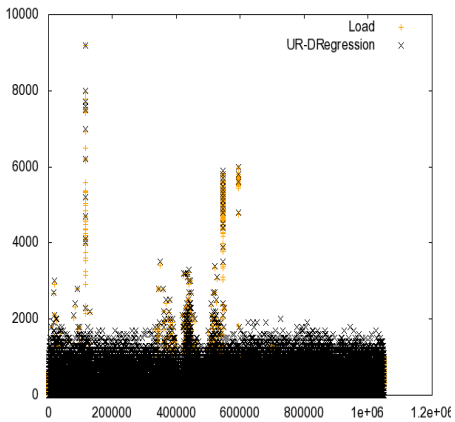
Table 3: *Comparison between the UR-DRegression, UR-DP$_{C1}$, UR-DP$_{C2}$, and UR-DR elasticity engines*

Name	S^-	S^+	S^-	S^+
UR-DRegression	-74791.7	1568047.8	-2.24%	47%
UR-DP$_{C1}$	-50307.2	1076236.3	-1.51%	32.24%
UR-DP$_{C2}$	-35818.6	1326841.7	-1.07%	39.75%
UR-DR	-99801.8	653082.9	-2.99%	19.57%

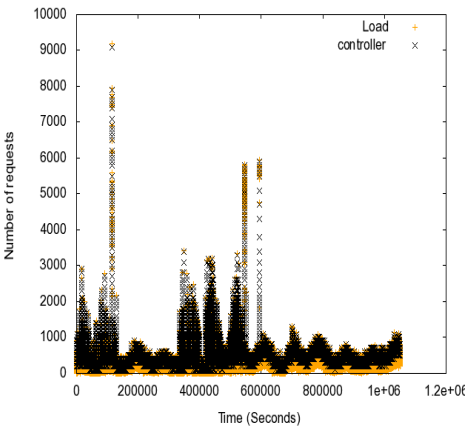
regression is recomputed every time a new measurement data is available on the full history, simulation using the whole world cup traces would be time consuming. Instead, in this experiment we used part of the trace from 09:47:41 on the 13^{th} of May, 1998 to 17:02:49 on the 25^{th} of May, 1998. We multiply the number of concurrent requests by 10 and assume that the servers can handle up to 100 requests. We assume that the SLA requires that a maximum of 5% of the capacity of a single server is not serviced per unit time.

Table 3 shows the aggregated results for four elasticity engines; UR-DRegression, UR-DP$_{C1}$, UR-DP$_{C2}$ and UR-DR. Although all the proactive approaches reduce the value of S^- compared to a UR-DR engine, P$_{C2}$ still shows superior results. The number of unused server that get provisioned by the regression controller S^+ is 50% more than for P$_{C1}$ and 15% more than P$_{C2}$ although both P$_{C1}$ and P$_{C2}$ reduces S^- more. The $UR-DR$ controller has a higher SLA violation rate (3%) while maintaining a much lower over-provisioning rate (19.57%). As we evaluate the performance of the controller on a different part of the workload and we multiply the workload by a different factor, the percentages of the capacity the controller fail to provision on time and the unused provisioned capacity changed from the previous experiment.

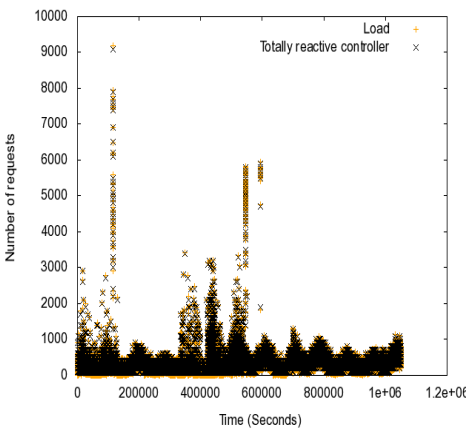
Figures 5(a), 5(b) and 5(c) show the load compared to the controller outputs for the UR-DR, UR-DP$_{C2}$, and UR-DRegression approaches. The amount of unused capacity using a regression based controller is much higher than the unused capacity for the other controllers. The controller output for the UR-DRegression engine completely over-estimates the load causing prediction oscillations between the crests and the troughs. One of the key advantages of P$_{C1}$ and P$_{C2}$ is that they depend on simple calculations. They are both scalable with time compared to the regression controller. The highest observed estimation time for the UR-DRegression is 6.5184 seconds with an average of 0.97695 seconds compared to 0.000512 seconds with an average of 5.797×10^{-6} in case of P$_{C1}$ and P$_{C2}$.



(a) UR-DRegression elasticity engine.



(b) UR-DP$_{C2}$ elasticity engine.



(c) UR-DR elasticity engine

Figure 5: Performance Comparison of UR-DR, UR-DP$_{C2}$ and UR-DRegression elasticity engines. The UR-DRegression controller over-provisions many servers to cope with the changing workload dynamics.

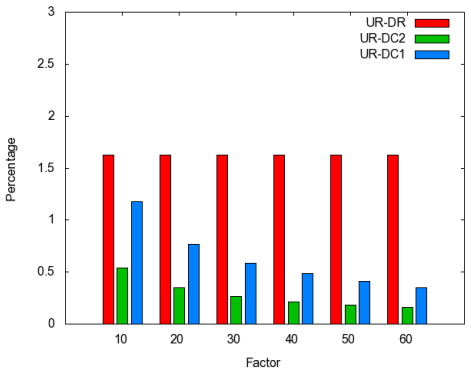
3.3 Performance impact of the workload size

In this experiment we investigate the effect of changing the load and server power on the performance of our proposed elasticity engines. We constructed six new traces using the world cup workload traces by multiplying the number of requests per second in the original trace by a factor of 10, 20, 30, 40, 50, and 60. We ran experiments with the new workloads using the UR-DR, UR-DP$_{C1}$ and UR-DP$_{C2}$ elasticity engines. For each simulation run, we assume that the number of requests that can be handled by any server is 10 times the factor by which we multiplied the traces, e.g., for an experiment run using a workload trace where the number of requests is multiplied by 20, we assume that the server capacity is up to 200 requests per second. We also assume that for each experiment the SLA specifies the maximum unhandled number of requests to be 5% of the maximum capacity of a single server.

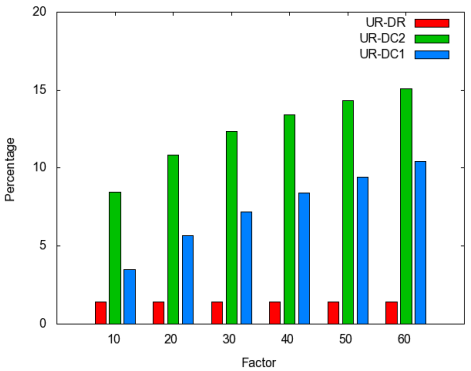
Figure 6(a) shows the percentage of servers the engines failed to provision on time to handle the increase in demand for each workload size (S^-) while Figure 6(b) shows the percentages of extra servers provisioned for each workload size. It is clear that the UR-DR engine exhibits the same performance with changing workloads. For the UR-DP$_{C1}$ and the UR-DP$_{C2}$ engines on the other hand, the performance depends on the workload and the server capacity. As the factor by which we multiply the workload increases, the percentage of servers the two engines failed to provision on time decreases. Inversely, the percentage of extra servers provisioned increases. These results indicate that the quality of estimation changes with any change in the workload. We attribute the improvement in the quality of estimation when the load increases using the UR-DP$_{C1}$ and UR-DP$_{C2}$ engines to the ability of both estimators to predict the envelope of the workload, thus decreasing the number of prematurely deallocated resources. Although the number of requests increases in the different workloads, the number of times the controllers deallocate resources prematurely also increases, but at a slower rate than the load. We have performed similar experiments with the Wikipedia traces [26] and obtained similar results [10]. Due to lack of space we omit those results.

3.4 Discussion

Although our proactive controllers P$_{C1}$ and P$_{C2}$ are designed using the change in the load as the controller parameter, they can be generalized to be used with any hardware parameter such as CPU load, memory consumption, network load and disk load or any server level parameter such as response time. When P$_{C1}$ or P$_{C2}$ controller is used with hardware measured parameter, e.g., CPU load, $C(t)$ becomes the total CPU capacity needed by the system to handle the CPU load per unit time. ΔD is the change in the load. $\widehat{C}$ becomes the average periodical measurement of the CPU load



(a) The effect of changing load size on the percentage of S^- to the total number of servers.



(b) The effect of changing load size on the percentage of S^+ to the total number of servers.

Figure 6: The effect of changing the workload size and the server capacity on the UR-DR, UR-DP$_{C1}$ and UR-DP$_{C2}$ elasticity engines.

and $\overline{n}$ the average measurement of the CPU load over time. The definition of the two controllers remains the same.

Both the UR-DP$_{C1}$ and UR-DP$_{C2}$ engines can be integrated in the model proposed by Lim et al. [20] to control a storage cloud. In storage clouds, adding resources does not have an instantaneous effect on the performance since data must be copied to the new allocated resources before the effect of the control action takes place. For such a scenario, P$_{C1}$ and P$_{C2}$ are very well suited since they predict the envelope of the demand. The engines can also replace the elasticity controllers designed by Urgaonkar et al. [27] or Iqbala et al. [13] for a multi-tier service deployed in the cloud.

4 Related Work

The problem of dynamic provisioning of resources in computing clusters has been studied for the past decade. Cloud elasticity can be viewed as a generalization of that problem. Our model is similar to the model introduced in [9]. In that work, the authors tried to estimate the availability of a machine in a distributed storage system in order to replicate its data.

Toffetti et al. [25] use Kriging surrogate models to approximate the performance profile of virtualized, multi-tier Web applications. The performance profile is specific to an application. The Kriging surrogate model needs offline training. A change in the workload dynamics results in a change in the service model. Adaptivity of the

service model of an application is vital to cope with the changing load dynamics in todays Internet [4].

Lim et al. [20] design an integral elasticity controller with proportional thresholding. They use a dynamic target range for the set point. The integral gain is calculated offline making this technique suitable for a system where no sudden changes to the system dynamics occur as the robustness of an integral controller is affected by changing the system dynamics [22].

Urgaonkar et al. [27] propose a hybrid control mechanism that incorporates both a proactive controller and a reactive controller. The proactive controller maintains the history of the session arrival rate seen. Provisioning is done before each hour based on the worst load seen in the past. No short term predictions can be done. The reactive controller acts on short time scales to increase the resources allocated to a service in case the predicted value is less than the actual load that arrived. No scale down mechanism is available.

In [18], the resource-provisioning problem is posed as one of sequential optimization under uncertainty and solved using limited look-ahead control. Although the solution shows very good theoretical results, it exhibits an exponential increase in computation time as the number of servers and VMs increase. It takes 30 minutes to compute the required elasticity decision for a system with 60 VMs and 15 physical servers. Similarly, Nilabja et al. use limited lookahead control along with model predictive control for automating elasticity decisions. Improving the scalability of their approach is left as a future direction to extend their work.

Chacin and Navaro [7] propose an elastic utility driven overlay network that dynamically allocate instances to a service using an overlay network. The instances of each services construct an overlay while the non-allocated instances construct another overlay. The overlays change the number of instances allocated to a service based on a combination of an application provided utility function to express the service's QoS, with an epidemic protocol for state information dissemination and simple local decisions on each instance.

There are also some studies discussing vertical elasticity [15]. Jung et al. [14] design a middleware for generating cost sensitive adaptation actions such as elasticity and migration actions. Vertical elasticity is enforced using adaptation action in fixed steps predefined in the system. To allocate more VMs to an application a migration action is issued from a pool of dormant VMs to the pool of the VMs of the target host followed by an increase adaptation action that allocates resources on the migrated VM for the target application. These decisions are made using a combination of predictive models and graph search techniques reducing scalability. The authors leave the scalability of their approach for future work.

5 Conclusions and Future Work

In this paper, we consider the problem of autonomic dynamic provisioning for a cloud infrastructure. We introduce two adaptive hybrid controllers P_{C1} and P_{C2}, that use both reactive and proactive control to dynamically change the number of VMs allocated to a service running in the cloud based on the current and the predicted future demand. Our controllers detect the workload envelope and hence do not deallocate resources prematurely. We discuss the different ways of designing a hybrid elasticity controller that incorporates both reactive and proactive components. Our simulation results show that using a reactive controller for scale up and one of our proactive controllers for scale down improves the SLA violations rate two to ten times compared to a totally reactive elasticity engine. We compare our controllers to a regression based elasticity controller using a different workload and demonstrate that our engines over-allocate between 32% and 15% less resources compared to a regression based engine. The regression based elasticity engine SLA violation rate is 1.48 to 2.1 times the SLA violation rate for our engines. We also investigate the effect of the workload size on the performance of our controllers. For increasing loads, our simulation results show a sublinear increase in the number of SLAs violated using our controllers compared to a linear increase in the number of SLAs violations for a reactive controller. In the future, we plan to integrate vertical elasticity control in our elasticity engine and modify the controllers to consider the delay required for VM start up and shut down.

6 Acknowledgments

This work is supported by the OPTIMIS project (http://www.optimis-project.eu/) and the Swedish government's strategic research project eSSENCE. It has been partly funded by the European Commission's IST activity of the 7th Framework Program under contract number 257115 . This research was conducted using the resources of High Performance Computing Center North (http://www.hpc2n.umu.se/).

References

[1] Amazon S3 service level agreement, October 2007.

[2] Rackspace hosting: Service level agreement, June 2009.

[3] Internet growth statistics, July 2011.

[4] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long. Managing flash crowds on the Internet. 2003.

[5] M. Arlitt and T. Jin. "1998 world cup web site access logs", August 1998.

[6] P. Bodık. *Automating Datacenter Operations Using Machine Learning*. PhD thesis, University of California, 2010.

[7] P. Chacin and L. Navarro. Utility driven elastic services. In *Distributed Applications and Interoperable Systems*, pages 122–135. Springer, 2011.

[8] P. Dhawan. Performance comparison: Exposing existing code as a web service, October 2001.

[9] A. Duminuco, E. Biersack, and T. En-Najjary. Proactive replication in distributed storage systems using machine availability estimation. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 27:1–27:12, New York, NY, USA, 2007. ACM.

[10] A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan. Optimis: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66 – 77, 2012.

[11] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[12] W. M. Hartmann. *Signals, sound, and sensation*. Amer Inst of Physics, 1997.

[13] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871 – 879, 2011.

[14] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 9:1–9:20. Springer-Verlag New York, Inc., 2009.

[15] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 117–126, New York, NY, USA, 2009. ACM.

[16] D. Kossmann and T. Kraska. Data management in the cloud: Promises, state-of-the-art, and open questions. *Datenbank-Spektrum*, 10:121–129, 2010. 10.1007/s13222-010-0033-3.

[17] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz. Napsac: design and implementation of a power-proportional web cluster. *ACM SIGCOMM Computer Communication Review*, 41(1):102–108, 2011.

[18] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12(1):1–15, 2009.

[19] H. Li and T. Yang. Queues with a variable number of servers. *European Journal of Operational Research*, 124(3):615 – 628, 2000.

[20] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[21] P. Mell and T. Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6), 2009.

[22] M. Morari. Robust stability of systems with integral control. *Automatic Control, IEEE Transactions on*, 30(6):574–577, 1985.

[23] K. Ogata. *Modern control engineering*. Prentice Hall, 2009.

[24] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 50–57. IEEE, 2009.

[25] G. Toffetti, A. Gambi, M. Pezzé, and C. Pautasso. Engineering autonomic controllers for virtualized web applications. *Web Engineering*, pages 66–80, 2010.

[26] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. `http://www.globule.org/publi/WWADH_comnet2009.html`.

[27] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1, 2008.

Paper II

Efficient Provisioning of Bursty Scientific Workloads on the Cloud Using Adaptive Elasticity Control

A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth

In Proceedings of the 3rd workshop on Scientific Cloud Computing (ScienceCloud), pages 31-40, ACM, 2012.

Efficient Provisioning of Bursty Scientific Workloads on the Cloud Using Adaptive Elasticity Control*

Ahmed Ali-Eldin†, Maria Kihl‡, Johan Tordsson†, and Erik Elmroth†

Abstract

Elasticity is the ability of a cloud infrastructure to dynamically change the amount of resources allocated to a running service as load changes. We build an autonomous elasticity controller that changes the number of virtual machines allocated to a service based on both monitored load changes and predictions of future load. The cloud infrastructure is modeled as a $G/G/N$ queue. This model is used to construct a hybrid reactive-adaptive controller that quickly reacts to sudden load changes, prevents premature release of resources, takes into account the heterogeneity of the workload, and avoids oscillations. Using simulations with Web and cluster workload traces, we show that our proposed controller lowers the number of delayed requests by a factor of 70 for the Web traces and 3 for the cluster traces when compared to a reactive controller. Our controller also decreases the average number of queued requests by a factor of 3 for both traces, and reduces oscillations by a factor of 7 for the Web traces and 3 for the cluster traces. This comes at the expense of between 20% and 30% over-provisioning, as compared to a few percent for the reactive controller.

1 Introduction

Elasticity of the cloud infrastructure is the ability of the infrastructure to allocate resources to a service based on the running load as fast as possible. An *elasticity controller* aims to allocate enough resources to a running service while at the same

*The paper has been re-typeset to match the thesis style. Produced with permission of ACM.

†Department of Computing Science, Umeå University, Sweden, email: {ahmeda, tordsson, elmroth}@cs.umu.se

‡Department of Electrical and Information Technology, Lund University, Sweden, email: {Maria.Kihl}@eit.lth.se

time avoiding costly over-provisioning. The problem for an elasticity controller is thus to decide when, and how much, to scale up or down. Scaling can be done either horizontally, by increasing or decreasing the number of Virtual Machines (VMs) allocated, or vertically, by changing the hardware configuration for CPU, memory, etc. of already running VMs. The resources allocated to a service can vary between a handful of VMs to tens of thousands of VMs depending on the load requirements. Most Infrastructure as a Service (IaaS) providers does not host a single service but rather quite a few scalable services and applications. Given the scale of the current and future cloud datacenters and services, these are impossible to manage manually, making autonomic management a key issue for clouds.

Recently, the scientific computing community started discussing the potential use of cloud computing infrastructures to run scientific experiments such as medical NLP processing [6] and workflows for astronomical data released by the Kepler project [25]. Most of the applications are embarrassingly parallel [11]. There are some limitations to the wide adoption of the cloud paradigm for scientific computing as identified by Truong et al. [23] such as the lack of cost evaluation tools, cluster machine images and, as addressed in this paper, autonomic elasticity control.

There are many approaches to solve the elasticity problem [5, 7, 10, 17, 18, 20, 24, 27, 28], each with its own strengths and weaknesses. Desired properties of an elasticity controller include the following:

- Fast: The time required by the controller to make a decision is a key factor for successful control, for example, limited look-ahead control is shown to have superior accuracy but requires 30 minutes to control 60 VMs on 15 physical servers [14].

- Scalable: The controller should be scalable with respect to the number of VMs allocated to a service and with respect to the time of running the algorithm. There are many techniques that can be used for estimation of the load and elasticity control which are not scalable with either time or scale e.g., regression based control is not scalable with respect to the algorithm execution time [1].

- Adaptive: Scientific workloads and Internet traffic are very dynamic in nature [2, 15]. Elasticity controllers should have a proactive component that predicts the future load to be able to provision resources a priori. Most prediction techniques such as neural networks build a model for the load in order to predict the future. Another desired property of an adaptive controller is the ability to change the model whenever the load dynamics change.

- Robust and reliable: The changing load dynamics might lead to a change in the controller behavior [9, 19]. A controller should be robust against changing load dynamics. A robust controller should prevent oscillations in resource

allocation i.e., the controller should not release resources prematurely. A reactive controller (step controller) is a controller that only allocates new VMs to a service when the load increases and deallocates the VMs once the load decreases beyond a certain level. This type of controller thus reduces the number of VMs provisioned and minimizes the provisioning costs, at the expense of oscillations.

Our previous work [1] studies different ways to combine reactive and proactive control approaches for horizontal elasticity. The two simple hybrid controllers proposed combine reactive scaling up with proactive scale-down. These controllers act on the monitored and predicted service load, but ignore multiple important aspects of infrastructure performance and service workload. In this paper, our previous work is extended by an enhanced system model and controller design. The new controller takes into account the VM startup time, workload heterogeneity, and the changing request service rate of a VM. It thus controls the allocated capacity instead of only the service load. The controller design is further improved by adding a buffer to the controller to store any delayed requests for future processing. This buffer model characterizes many scientific workloads where jobs are usually queued for future processing. The proposed controller can be used by both the cloud service provider and the cloud user to reduce the cost of operations and the cost of running a service or an experiment in the cloud. The controller can be used also to control the elasticity of a privately run cloud or cluster.

The performance of the controller is tested using two sets of traces, a Web workload from the FIFA world cup [3] and a recently published workload from a Google cluster composed of around 11 thousand machines [26]. The Web trace is selected as it is a well known and rather bursty workload and thus challenging for an elasticity controller. The cluster traces, consisting mostly of MapReduce jobs, are chosen to evaluate the behavior of our approach on traces more similar to scientific workloads.

The rest of this paper is organized as follows. Section 2 describes the system model and the design of the proposed controller. In Section 3, the simulation framework and the experiments are described and the results are discussed. Section 4 discusses some of the different approaches available in the literature for building elasticity controllers. Section 5 contains the conclusions.

2 Controller design

2.1 System model

In this work, the cloud infrastructure is modeled as a closed loop control system and queueing models are used to design a feedback elasticity controller. The cloud infrastructure is modeled as a $G/G/N$ stable queue in which the number of servers

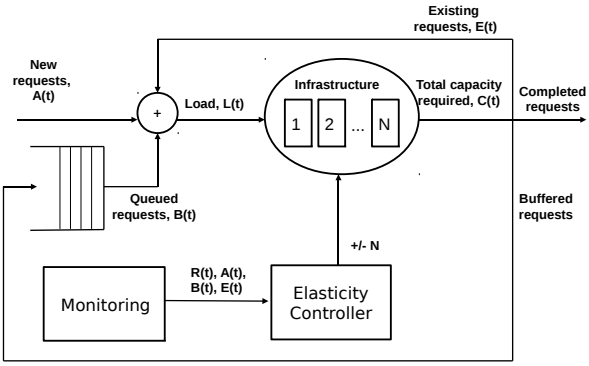


Figure 1: Queueing model and elasticity control for a cloud service.

N required is variable [16] as shown in Figure 1. This is a generalization of the work by Khazaei et al. [13] where a cloud is modeled as an $M/G/m$ queue with a constant number of servers, m. We assume that the system serves generic requests that can be anything from a Web query to Pubmed [12] to a job in a workflow to process astronomical data [25].

The number of VMs allocated to a service at any time unit, N, changes according to the controller output. When the load increases, VMs are added and when it decreases VMs are removed. We assume that it takes one time unit for a VM to boot and get intialized. In practice, it also takes time to shut-down a VM, but for most applications, no more requests are sent to a VM after a shutdown command is issued. It is thus assumed that the effect of VM shut down on capacity is instantaneous.

In our model, requests not served are buffered and delayed as shown in Figure 1. We make no assumptions about a finite buffer size, but the designed controller uses the number of buffer requests as one criteria for adding VMs. The buffer length is also used as a performance metric in the evaluation section. A buffered request is delayed and thus the larger the number of buffered requests, the slower the request response time. Assuming the queue is stable, the average service rate over time is equal to the average arrival rate over time. Whenever the system is at risk of instability due to increase in the demand, the elasticity controller increases N to enforce system stability. The elasticity control problem can be stated as follows: the elasticity controller should add or remove VMs to ensure system stability, i.e., over a long period of time, the number of serviced requests (the service capacity) is equal

Table 1: *Overview of used notation.*

Variable	Description
N	Number of VMs deployed
$L(t)$	Total service load at time t
$R(t)$	Total service capacity available at time t
$C(t)$	Service capacity required at time t
$A(t)$	Arriving (new) requests at time t
$D(t)$	Increase/decrease in required capacity at time t
$B(t)$	Size of buffer at time t
$E(t)$	Amount of already processing requests at time t
K	Number of queued requests before starting a new VM
r	Number of time units needed to start all buffered requests
T_d	Estimation interval (time between two estimations)
$\overline{L_{T_d}}$	Average load over the last estimation interval
$\overline{L_t}$	Average load over all time
$\tilde{D}$	Predicted value for request change rates over next T_d time units
P	Estimated ratio between $\tilde{D}$ and average load
M_{Avg}	The average of the median request service rates per unit time over the the T_d

to the total number of received requests received with an error tolerance (number of buffered requests). This should be achieved irrespective of the change in the request arrival rate and while maintaining the number of VMs to a minimum.

2.2 Estimating future usage

The optimal total service capacity, $C(t)$, required for time t is:

$$C(t) = C(t-1) + D(t), \tag{1}$$

where $C(t-1)$ is the capacity required in the last time step and $D(t)$ is the increase or decrease in capacity needed in order to meet SLAs while maintaining the number of VMs to a minimum. The controller is activated each T_d time units. When

evoked at time t, it estimates the change in workload for the next T_d time units, $D(t+1), D(t+2), \ldots, D(t+T_d)$. VM allocations are adjusted at times $t+1, t+2, \ldots, t+T_d$ according to these predictions, followed by a new prediction for $t+T_d \ldots t+2T_d$. We define $A(t)$ as the arrival rate of new requests to the service. A suitable initial service configuration could be $C(0) = A(0)$.

We define the total workload of the service, $L(t)$, as the sum of the arriving requests, the existing, already processing requests, $E(t)$, and any buffered requests to be served. No assumptions are thus made about the time needed to serve a request, which can vary from seconds to hours. We use $B(t)$ to denote the number of requests in the buffer at time t. If enough VMs are allocated to initialize all buffered requests in the next time unit, these machines may become idle and be released shortly after, causing oscillations in resource allocations. We thus define r, a system parameter specifying over how many time units the currently buffered load should be started. Now, the total workload at time t can be written as:

$$L(t) = A(t) + E(t) + \frac{B(t)}{r}. \tag{2}$$

The capacity change required can be written as

$$D(t) = L(t) - R(t) \tag{3}$$

where $R(t)$ denotes the currently allocated capacity. Assuming that $A(t)$ remains constant for the next time unit, the estimated change in the current service capacity required, $\tilde{D}$ for the future T_d time units can be estimated by

$$\tilde{D} = P\overline{L_{T_d}} \tag{4}$$

where P represents the estimated rate of adding or removing VMs. We define $\overline{L_{T_d}}$ to be the average periodical service load over the past T_d time units,

$$\overline{L_{T_d}} = \frac{\sum_{i=0}^{T_d} L(t-i)}{T_d}. \tag{5}$$

Similarly, we define $\overline{L_t}$, as the average load over all time as follows:

$$\overline{L_t} = \frac{\sum_{i=0}^{t} L(i)}{t}. \tag{6}$$

Now, P represents the estimated ratio of the average change in the load to the average load over the next T_d time units and $\overline{L_{T_d}}$ is the estimated average capacity required to keep the buffer size stable for the next T_d time units. P is positive if there is an increase in total workload (new service requests, buffered requests, and requests that

need to be processed longer); negative if this sum decreases (with the buffer empty); and zero if the system is at a steady state and the buffer is empty.

We define P to be the ratio between $D(t)$ and the average system load over time,

$$P = \frac{D(t)}{\overline{L_t}}. \tag{7}$$

This value represents the change in the load with respect to the average capacity. By substituting Equations 7 in Equation 4,

$$\tilde{D} = \frac{\overline{L_{T_d}}}{\overline{L_t}} D(t). \tag{8}$$

This formulation is a proportional controller [21] where $D(t)$ is the error signal and $\overline{L_{T_d}}/\overline{L_t}$, the normalized service capacity, is the gain parameter of the controller. By substituting Equation 5 in Equation 8, we obtain

$$\tilde{D} = \frac{\sum_{i=0}^{T_d} L(t-i)}{\overline{L_t}} \frac{D(t)}{T_d}. \tag{9}$$

If T_d is optimal, i.e., estimations occur when the rate of change of the load changes, then $D(t)/T_d$ is the slope of the changing load multiplied by the ratio between the instantaneous load and the overtime average load.

2.3 Determining suitable estimation intervals

The interval between two estimations, T_d, is a crucial parameter affecting the controller performance. It is used to calculate P and $\tilde{D}$ and T_d also controls the reactivity of the controller. If T_d is set to one, the controller performs predictions every time unit. At the other extreme, if T_d is set to ∞, the controller performs no predictions at all. As the workloads observed in datacenters are dynamic [2], setting an adaptive value for T_d that changes according to the load dynamics is important.

We define the maximum number of buffered requests, K, as the tolerance level of a service i.e., the maximum number of requests queued before making a new estimation or adding a new VM, thus:

$$T_d = \begin{cases} K/|\tilde{D}| & \text{if K¿0 and —}\tilde{D}| \neq 0 \\ 1 & \text{if K=0 or } \tilde{D} = 0 \end{cases} \tag{10}$$

The value of K can be used to model SLAs with availability guarantees. A low value for K provides quicker reaction to load changes, but will also result in oscillations as resources will be provisioned and released based on the last few time units only. Conversely, K is large, the system reacts slowly to changing load dynamics. Similarly, r affects the rate with which buffered requests should be started, and thus impose similar tradeoffs between oscillations and quickly reacting to load increases.

2.4 Hybrid elasticity control

The main goal of an elasticity controller is to allocate enough resources to enforce the SLAs while decreasing total resource usage. It is very hard to anticipate whether an observed increase in load will continue to rise to a large peak [2] as there are no perfect estimators or controllers. Using pure estimation for scale-up decisions is dangerous as it can lead to system instability and oscillations if the load dynamics change suddenly while the controller model is based on the previous load.

Algorithm 1: Hybrid elasticity controller with both proactive and reactive components.

Data: r, K

Result: Perform resource (de)allocation to keep the system stable

$Proactive\_Aggregator \leftarrow 0$;
$T_d \leftarrow 1$;
for *each time step t* **do**
  Update $R(t), A(t), B(t)$, and $E(t)$ from monitoring data;
  Calculate $D(t)$ using Equation 3;
  if *Time from last estimation* $\geq T_d$ **then**
    Calculate $\overline{L_{T_d}}$ from Equation 5;
    Calculate $\overline{L_t}$ from Equation 6;
    Calculate P from Equation 7;
    Calculate $\tilde{D}$ from Equation 8;
    Update M_{Avg};
    Calculate T_d from Equation 10;
  $N_{Reactive} \leftarrow \lceil D(t)/M_{Avg} \rceil$;
  $Proactive\_Aggregator += \tilde{D}/M_{Avg}$;
  $N_{Proactive} \leftarrow \lfloor Proactive\_Aggregator \rfloor$;
  $Proactive\_Aggregator -= N_{Proactive}$;
  if $N_{Reactive} > K$ **then**
    if $N_{Proactive} > 0$ **then**
      Deploy $N_{Proactive} + N_{Reactive}$ servers
    else
      Deploy $N_{Reactive}$ servers
  else
    (Un)deploy $N_{Proactive}$ servers

Our design is a hybrid controller where a reactive component is coupled with the proposed proactive component for scale up and a proactive only component for scale down. The details of the implementation are given in Algorithm 1. The controller starts by receiving monitoring data from the monitoring subsystem in Line 4. If T_d time units passed since the last estimation, the system model is updated by reestimating $\tilde{D}$ and T_d as shown from Line 6 to Line 12. The unit of $\tilde{D}$ is requests per time unit.

The actual calculation of the number of VMs to be added or removed by the different controllers is done between lines 13 and 16. In some applications, $\tilde{D}$ is divided by M_{Avg} in Line 14 to find the number of servers required. The rate of the proactive controller can be steered by multiplying $\tilde{D}$ by a factor, e.g., to adding double the estimated VMs for some critical applications. The reactive controller is coupled with the proactive controller to reach a unified decision as shown from Line 17 to Line 23. For scale up decisions, when the decisions of both the reactive component and the proactive component are to scale up, the decisions are added. For example, if the reactive component decides that two more VMs are required while the proactive component decides that three VMs are needed, five VMs are added. The reactive component is reacting for the current load while the proactive component is estimating the future load based on the past load. If the reactive component decides that a scale up is needed while the proactive decides that a scale down is needed then the decision of the reactive component alone is performed because the reactive component's decision is based on the current load while the proactive component's decision may be based on a skewed model that needs to be changed.

For the reactive and proactive components to calculate the number of VMs required for a given load, the controller needs to know the service capacity of a VM i.e., the number of requests serviced per VM every time unit. This number is variable as the performance of a VM is not constant.

In addition, there are different request types and each request takes different time to service. As a solution, we calculate M_{Avg}, the average of the median request service rates per VM per unit time over the past estimation period T_d. M_{Avg} is used by the reactive and proactive components to calculate the number of VMs required per unit time to service all the requests while meeting the different SLAs. The median is chosen as it is a simple and efficient statistical measure which is robust to outliers and skewed distributions. No assumptions are made about the service rate distribution for a VM. M_{Avg} is a configurable parameter which can be changed based on deployment requirements.

3 Experimental Evaluation

To validate the controller, a three-phase discrete-event simulator [4] was built using python that models a service deployed in the cloud. Two different workloads are used for the evaluation, the complete traces from the FIFA 1998 world cup [3] and a set of Google cluster traces [26].

In all evaluations, the controller time step, i.e., the time it takes to start a VM is selected to be 1 minute, which is a reasonable assumption [22]. The effects of the controller's decision to increase the resources provisioned does thus not appear until after one minute has elapsed and the new VMs are running. The granularity of the monitoring data used by the controller is also 1 minute.

Algorithm 2: Reactive elasticity controller.

Data: r, K
Result: Perform resource (de)allocation to keep the system stable

for *each time step t* **do**
  Update $R(t), A(t), B(t)$, and $E(t)$ from monitoring data;
  Calculate M_{Avg};
  Calculate $D(t)$ using Equation 3;
  $N_{Reactive} \leftarrow \lceil D(t)/M_{Avg} \rceil$;
  if $N_{Reactive} > 0$ *and* $D(t) > K$ **then**
    Deploy $N_{Reactive}$ servers
  if $N_{Reactive} < -2$ **then**
    Undeploy $N_{Reactive}$ servers

The controller's performance is compared to a completely reactive controller similar to the one proposed by Chieu et al. [8]. The design of the reactive controller is shown in Algorithm 2. The calculation of the median service rate is done every minute as this is the time between two load estimations. In order to reduce oscillations in the reactive controller, scale down is not done until the capacity change $D(t)$ is less than the current provisioned capacity $C(t)$ by $2M_{Avg}$, i.e., scale down is only done when there are more than two extra VMs provisioned. In the experiments, we refer to our controller in Algorithm 1 as C_{Hybrid} and the reactive controller in Algorithm 2 as C_{Reactive}.

With a few exceptions, most of the work available on elasticity control compares performance with static provisioning. However, we chose to compare our controller with a reactive controller to highlight the tradeoffs between over-provisioning, SLA violations, and oscillations.

Table 2: Web workload performance overview.

C_{Hybrid} results								C_{Reactive} results					
F	λ	OP	$\overline{OP}$	UP	$\overline{UP}$	V	$\overline{N}$	OP	$\overline{OP}$	UP	$\overline{UP}$	V	$\overline{N}$
1	100	41905	0.548	3883	0.05	2.5	3	35600	0.47	267402	3.49	5.98	2.95
10	100	535436	6.99	8315	0.1	19.7	26.09	206697	2.7	8835898	115.45	135.97	23.28
20	100	1075447	14.05	98678	1.29	38.9	51.76	380059	4.966	19611571	256.26	297.29	46.14
30	100	1617452	21.14	148896	1.9	58.1	77.46	555503	7.25	30944637	404.35	466	69.15
40	100	2155408	28.16	197660	2.58	77.3	103.11	732157	9.567	42265699	552.28	634.57	92.14
20	200	654596	8.55	35380	0.46	19.3	27.57	225979	2.95	5187614	67.78	87.63	22.86
30	300	761956	9.96	30951.0	0.4	19.3	28.94	235436	3.07	3783052	49.4	69	22.71
40	400	857608	11.2	30512	0.4	19.3	30.16	241854	3.16	3180898	41.56	61.04	22.7

3.1 Performance Metrics

Different metrics can be used to quantify the controller performance. We define OP to be the number of over-provisioned VMs by the controller per time unit aggregated over the whole trace. $\overline{OP}$ is the average number of over-provisioned VMs by the controller per minute. Similarly, UP and $\overline{UP}$ are the aggregate and the average number of under-provisioned VMs by the controller per minute. We also define V, the average number of servers required to service the buffered load per minute,

$$V = \Sigma \frac{\text{Buffered Load}}{\text{Median Service rate of a VM}}. \tag{11}$$

V represents the way the buffers get loaded. It does not represent the optimal number of servers required to service the load but rather represents average required number of VMs due to the queue build up. We use $\overline{N}$ to denote the average number of VMs deployed over time.

3.2 Web workload performance evaluation

The Web workload contains 1.3 billion Web requests recorded at servers for the 1998 FIFA world cup in the period between April 30, 1998 and July 26, 1998. The aggregate number of requests per second were calculated from these traces. In the simulation, the requests are grouped by time of arrival. The focus is not on individual requests but rather on the macro-system performance. For the experiments, the average service rate of a VM is drawn from a Poisson distribution with an average equal to λ requests per second. It is assumed that the time required to process one request is 1 second. The tolerance level K is chosen to be 5, i.e., 5 requests may be buffered before the controller reacts.

Assuming perfect load balancing, the performance of the controller is the only factor affecting performance in the simulation. We set r to 60 seconds, i.e., queued requests are emptied over one minute.

The controller is configured for the worst case scenario by using the maximum load recorded during the past minute as the input request arrival rate to the controller. This assumption can be relaxed by monitoring the average or median load for the past minute instead. Using the median or the average will result in provisioning less resources for most workloads.

As the Web traces are quite old, we have multiplied the number of requests by a factor F in order to stress test the controller performance under different load dynamics. For different experiments, λ is also changed. Table 2 shows the performance of the two controllers when F takes the values of 1, 10, 20, 30, and 40 while λ takes the values of 100, 200, 300, and 400. Due to the size of the trace, the aggregate metrics UP and OP are quite large.

For the over-provisioning metrics, OP and $\overline{OP}$, it is clear that C_{Hybrid} has a higher over-provisioning rate compared to C_{Reactive}. This is intuitive because C_{Hybrid} provisions resources ahead in time to be used in the future and delays the release of resources in order to decrease oscillations. When λ changes such that the ratio between λ and F is constant at 10, OP and $\overline{OP}$ are reduced for both controllers compared to when only F increases and the rate of increase of both values is lower. Notably, these values are quite small if we compare them to static provisioning. If capacity would be statically provisioned for the workload, 42 servers would be needed when $F = 1$ and $\lambda = 100$, whereas using the proactive controller or even the reactive controller, the average number of provisioned servers is around 3, reducing resource use by around 92.5% compared to static provisioning but at the cost of an increase in the number of delayed requests.

Looking at the aggregate and the average under-provisioning metrics, UP and $\overline{UP}$, C_{Hybrid} is superior to C_{Reactive}. Since C_{Hybrid} scales up and down proactively, it prevents oscillations. It proactively allocates VMs to and does not release resources prematurely, thus decreasing the amount of buffered and delayed requests. In fact, C_{Reactive} shows a very high rate of under-provisioning due to the fact that all delayed requests are buffered. The average number of under-provisioned servers of C_{Reactive} is 70 times that of $C_{\text{Proactive}}$ when $F = 1$ and $\lambda = 100$. Since C_{Reactive} is always lagging the load and releases resources prematurely causing oscillations, the performance of the controller is quite bad. In real life, the buffer is not infinite and thus requests are dropped. In comparison, using the proposed controller, C_{Hybrid}, the request drop rate is quite low. Again, we note that for a ratio between λ and F of 10, under-provisioning (UP and $\overline{UP}$) is reduced for both controllers compared to when only F increases. Actually, for C_{Hybrid}, UP and $\overline{UP}$, are almost constant while for C_{Reactive}, they decrease significantly with the increase of λ and F while having a constant internal ratio. We attribute this to the higher service capacity of the VMs allowing the system to cope with higher system dynamics. For future work, we plan to investigate the effect of the VM capacity on the performance of an elasticity

Table 3: *Number of VMs added and removed for the Web workload with* $F = 1$ *and* $\lambda = 100$.

	X_R	X_P
$C_{\text{Proactive}}$	1141	1152
C_{Reactive}	15029	N/A

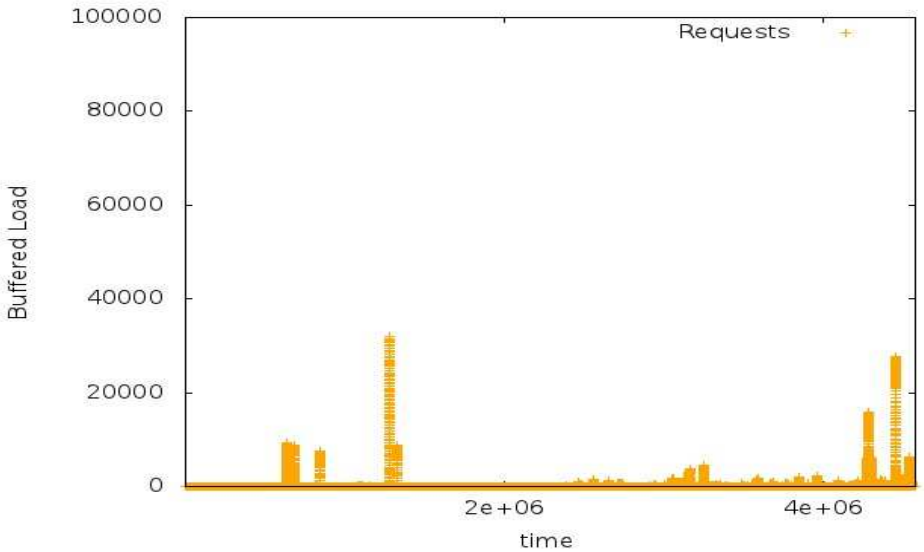
controller with different system dynamics and if the property we have just noted is general for any arrival process.

The V columns in Table 2 show the average number of VMs required to empty the buffer in one minute or the average number of minutes required by a single VM to empty the buffer. This is illustrated by figures 2(a) and 2(b) that show the average buffered requests per VM at any second for C_{Hybrid} and C_{Reactive} respectively when $N = 100$ and $K = 1$. In Figure 2(a) there are three major peaks when the buffered load per VM is above 1000 requests resulting in a relatively small V and $\overline{UP}$ in the table. These three peaks are a result of sudden large load increases. On the other hand, Figure 2(b) shows more peaks with buffered load more than 50000 requests per VM, resulting in a relatively high V and $\overline{UP}$. The average required buffer size for C_{Reactive} per VM in order to service all requests is almost 3 times the buffer size required by $C_{\text{Proactive}}$. Thus, for a limited buffer size, C_{Reactive} drops many more requests than C_{Hybrid}.

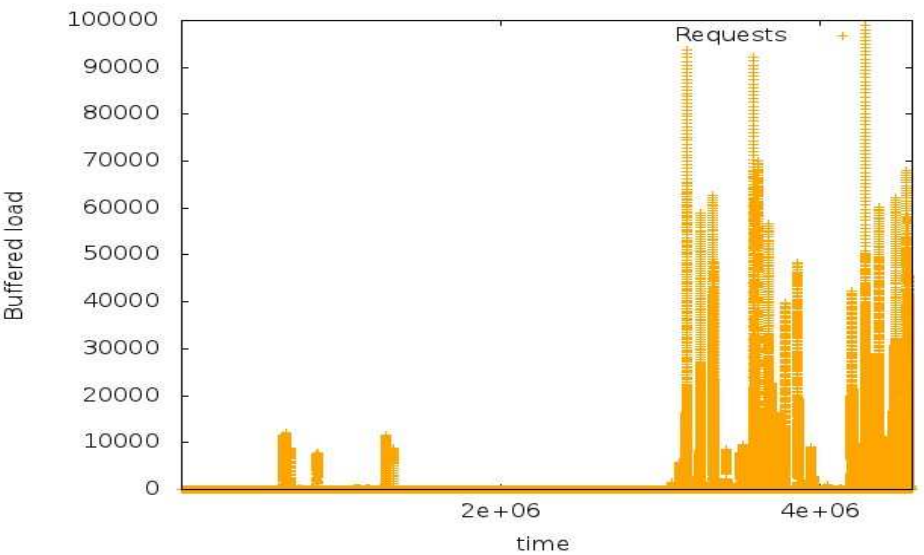
Figures 3(a) and 3(b) show the load and the provisioned capacity using both controllers for the full trace when $\lambda = 100$ and $F = 1$. These plots show the macro behavior of the controllers. The total number of buffered requests is the reason for having very high load peaks for C_{Reactive}. The largest spike seen in Figure 3(a) was around the fifth of May at 11:15. For ten minutes, the load suddenly increases tenfold and then starts oscillating causing some instability in $C_{\text{Proactive}}$. Notable, as the buffered load is emptied over r time units, the capacity does not increase with the same rate as the numbered of buffered requests increase.

To study the micro behavior of the controllers, figures 4(a) and 4(b) show the load and controller output for one and half hour from 21:21:12 on 25 June, 1998 to 22:44:31 on the same day. These figures show that $C_{\text{Proactive}}$ tracks the envelope of the load by keeping the provisioned capacity slightly higher than the load while C_{Reactive} oscillates the provisioned capacity with the increase or decrease of the load. Note that as the capacity of a single VM is variable, the capacity in Figure 4(a), which is measured in number of requests, appears to be oscillating. What actually happens is that the number of provisioned VMs drops gradually from 13 to 6 with no oscillations.

Table 3 shows X_R, the total number of servers added and removed by the reactive component of a controller, and X_P, the total number of servers added and removed by the proactive component, using C_{Hybrid} and C_{Reactive} for a simulation run with

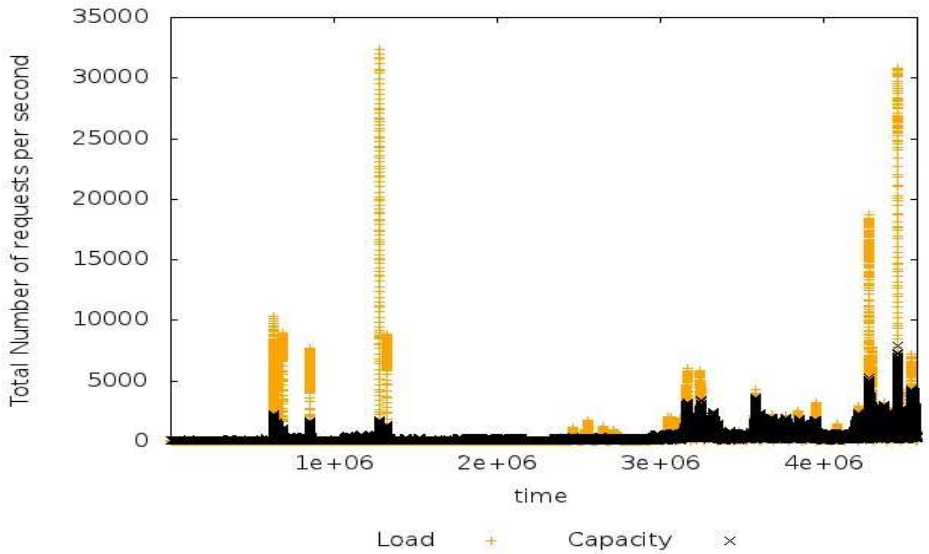


(a) $C_{Proactive}$

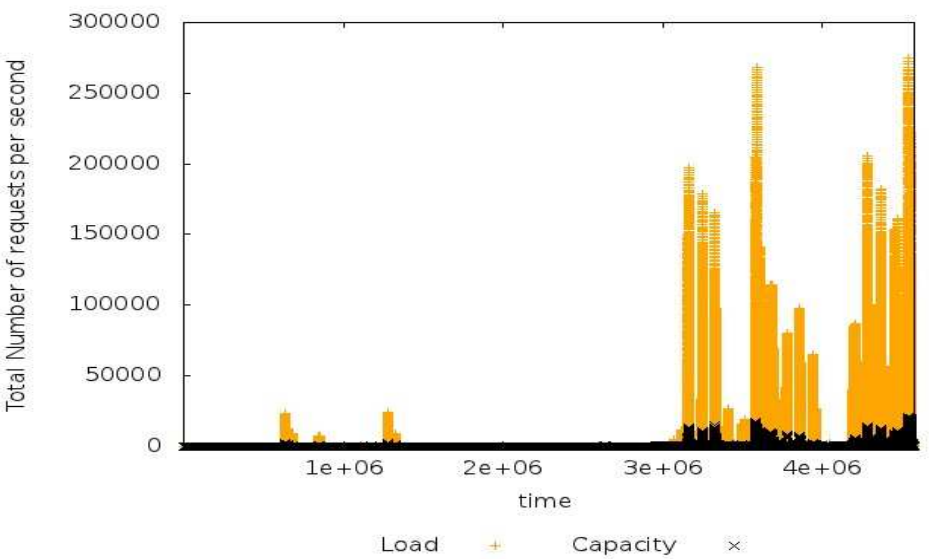


(b) C_{Reactive}

Figure 2: Number of buffered requests over time for the Web workload.

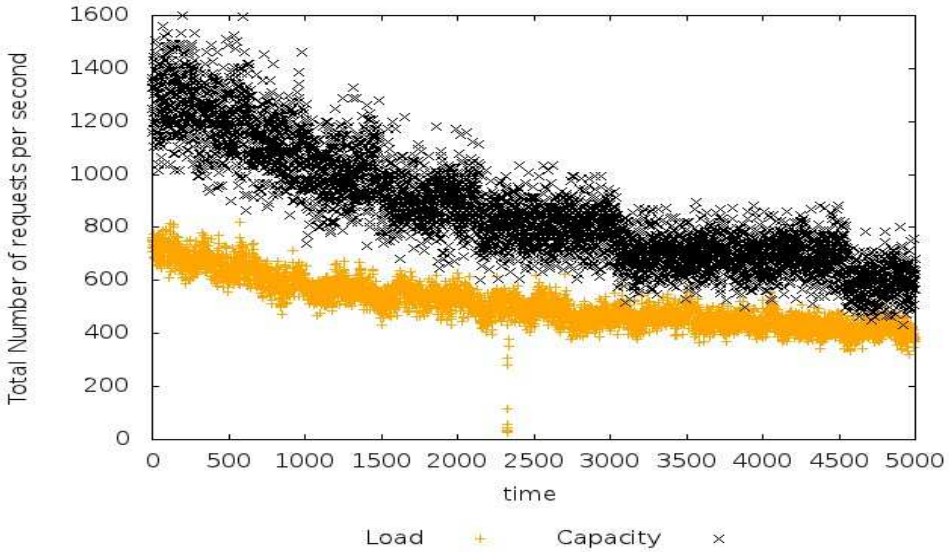


(a) C_{Hybrid}

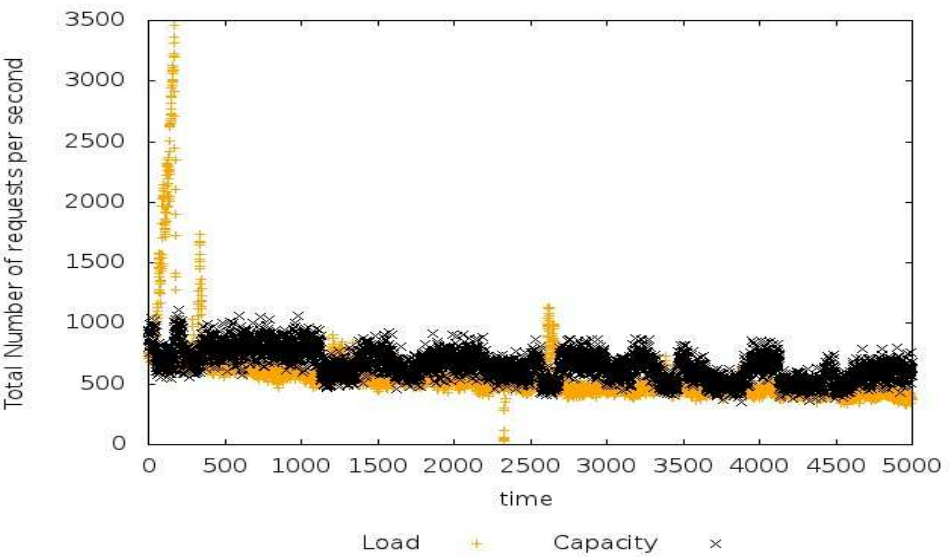


(b) C_{Reactive}

Figure 3: Load and the provisioned capacity over time for the Web workload.



(a) C_{Hybrid}



(b) C_{Reactive}

Figure 4: Load and the provisioned capacity for 1.5 hours of the Web workload.

Table 4: *Properties of the cluster workload.*

	execution time	queue time	total time
Median	347.4 s	3.6 s	441.6 s
Average	3961.8 s	220.76 s	4182.5 s
90*th* percentile	3803 s	3325 s	4409 s

$F = 1$ and $\lambda = 100$. The total number of server added or removed by $C_{\text{Proactive}}$ is 2293 servers almost one seventh of the total number of server added or removed by the reactive controller. These results illustrate how the reactive controller increases resource oscillations.

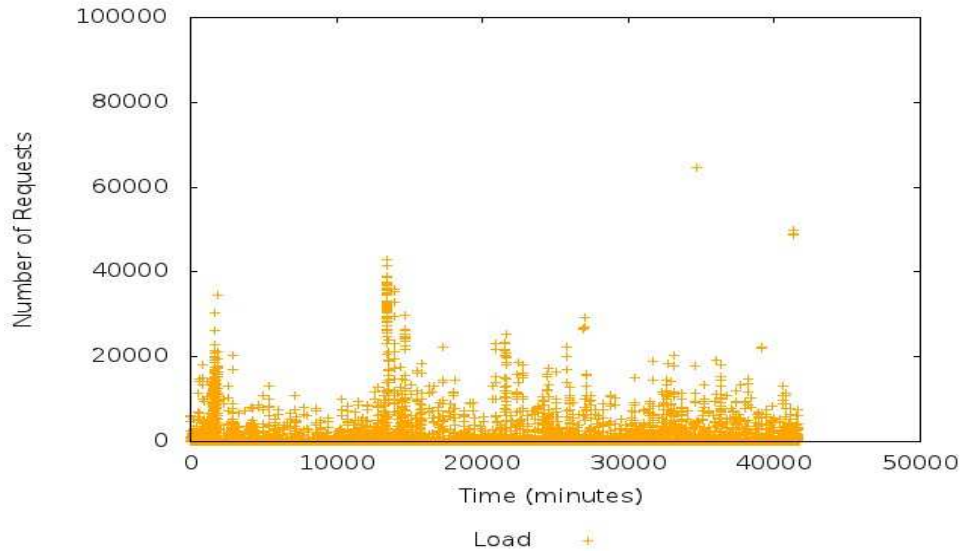
3.3 Cluster workload performance evaluation

Recently, Google published a new sample dataset of resource usage information from a Google production cluster [26]. The traces are from an cluster with 11000 servers. As this cluster is used to run a mixture of MapReduce and other computationally intensive jobs, the traces are representative for scientific workloads.

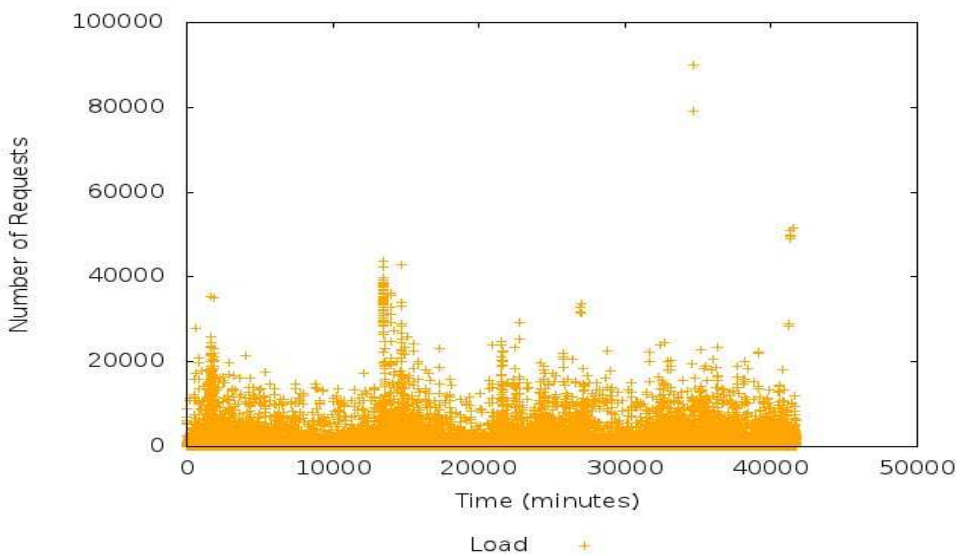
In this experiment, there is no risk of causing oscillations if $r = 1$ since most of the requests take more than 1 minute to serve. The median job length in the workload is 347.5 seconds or almost 6 minutes. Table 4 summarizes the statistical properties of the tasks in the workload. Due to lack of space we do not comment more on the properties of the workload. K is set to 5 jobs also for this experiment. We define that a job is delayed if it remains in the queue for more than 1 minute. The median number of tasks that can be processed on a single machine in the trace is 170, while the minimum is 120. To be more conservative, we set the number of tasks assigned to a server to 100.

Table 5 shows the performance of the proactive and reactive controllers. The amount of under-provisioning using the C_{Reactive} is almost three times that of $C_{\text{Proactive}}$. This comes at the cost of over-provisioning on average 164 VMs compared to around 1.4 VMs for the reactive controller. However, the amount of over-provisioning is still low, around 25%, as $C_{\text{Proactive}}$ used 847 VMs on average, as compared to 687 VMs for the reactive controller.

While $\overline{OP}$ and $\overline{UP}$ may be crucial for a workload like the Web trace, they are less important for a workload of jobs like the cluster trace where a job can wait in the queue for minutes. More importantly for this workload type is V, the average number of buffered tasks. $C_{\text{Proactive}}$ keeps the average number of buffered tasks below K. On the contrary, the reactive controller's average buffer length is double the allowed buffer size K and three times that of the proactive controller. This is illustrated in figures 5(a) and 5(b) that show the number of buffered requests over time.

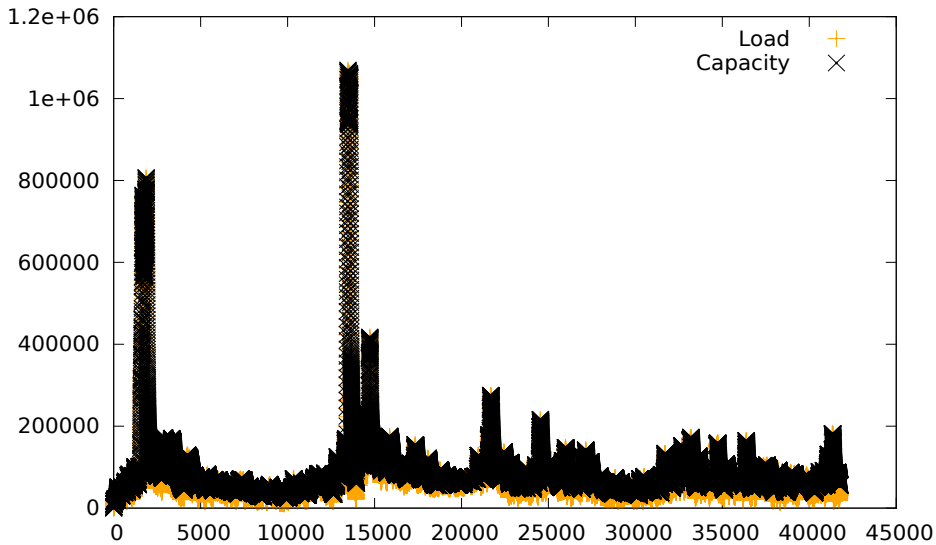


(a) $C_{Proactive}$

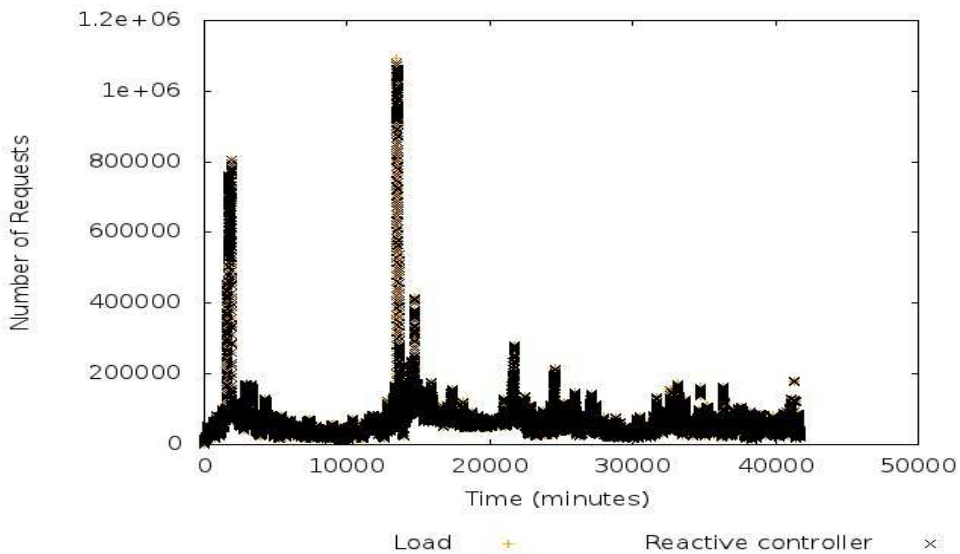


(b) C_{Reactive}

Figure 5: Number of buffered requests over time for the cluster workload.

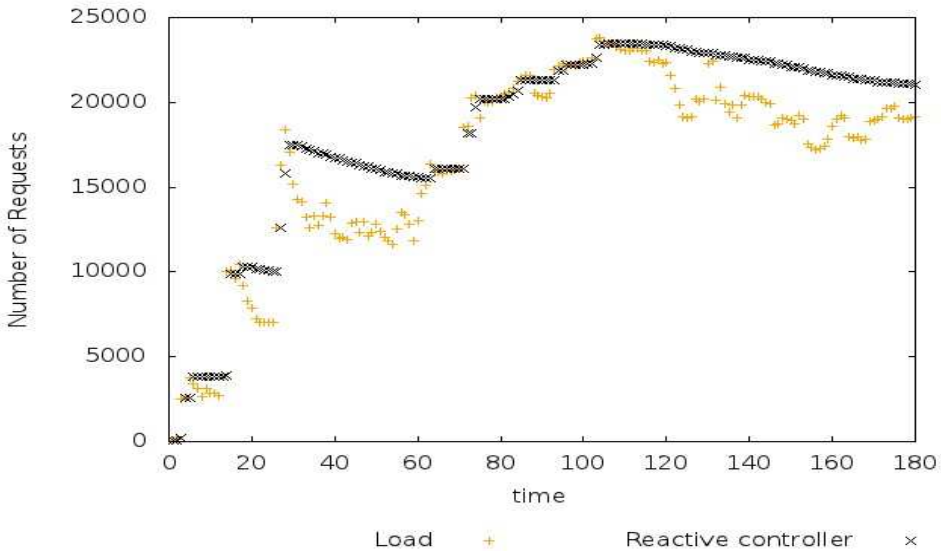


(a) $C_{Proactive}$

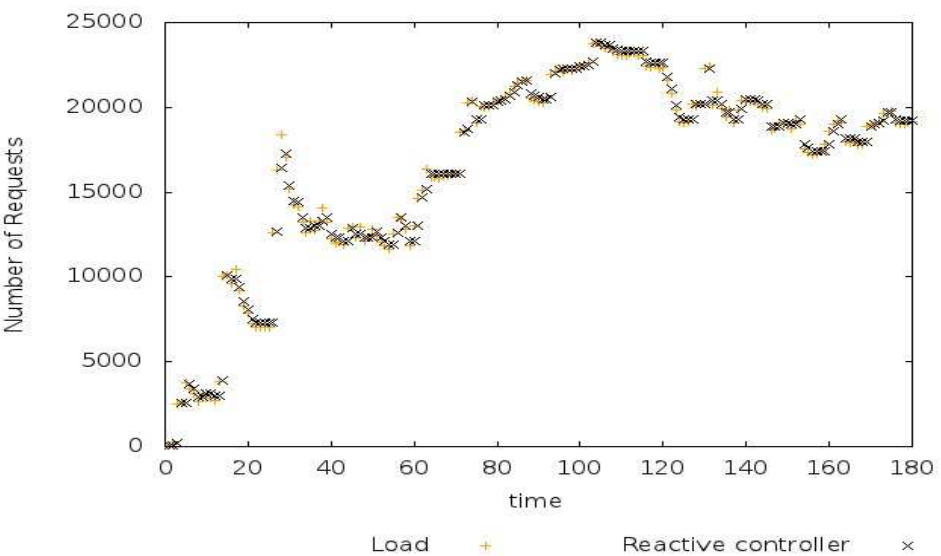


(b) C_{Reactive}

Figure 6: Load and the provisioned capacity over time for the cluster workload.



(a) C_{Hybrid}



(b) C_{Reactive}

Figure 7: Load and the provisioned capacity for 3 hours of the cluster workload.

Table 5: *Cluster workload performance overview.*

	$C_{\text{Proactive}}$	C_{Reactive}
$\overline{OP}$	164	1.369
$\overline{UP}$	1.76	5.384
$\overline{N}$	847	686.9
V	3.48	10.22
X_R	75415.0	505289
X_P	78564.0	N/A

We also note that in total, the number of VMs added and removed by the reactive controller is 505289 compared to 153979 by the proactive controller. This means that the reactive controller results in more oscillations also for the cluster workload.

Figures 6(a) and 6(b) show the load and provisioned capacity for $C_{\text{Proactive}}$ and C_{Reactive}. The proactive controller tracks the envelope of the workload, i.e., the capacity stays ahead of the load most of the time, whereas the reactive controller always lags the load by at least one time unit. Due to the large number of points plotted, the load appears as if it is completely covered with the capacity. In order to see the micro behavior of the two controllers we plot the load and capacity for both controllers for the first 3 hours of the trace in figures 7(a) and 7(b). The figures show how oscillations are reduced using the proactive controller. For example, for the sudden decreases in load at minutes 15 and 30, C_{Reactive} quickly deallocated VMs followed by reallocations as load increased again. In contrast, $C_{\text{Proactive}}$ kept most of the allocated VMs, causing less oscillations. To summarize the experiments, the workload characteristics and the SLA requirements influence the performance of both controllers considerably. We also note that our elasticity controller is highly scalable with respect to service workload and infrastructure size. In the performed evaluations, the controller required on average a few milliseconds to make a decision.

4 Related work

Elasticity is an incarnation of the dynamic provisioning problem which has been studied for over a decade [7] from the perspectives of both server provisioning and cloud computing. Different approaches have been proposed to solve the problem in both its old and new incarnations. Some previous research considered only vertical elasticity [17, 27]. while many others considered horizontal elasticity in different contexts [20, 28].

Urgaonkar et al. [24] were among the first to discuss the effect of virtualization on the provisioning problem or what we call horizontal elasticity. They proposed

an adaptive controller composed of a predictive and a reactive components. The predictive component acts in the time scale of hours or days. It provisions resources based on the tail distribution of the load. The reactive component acts in the time scale of minutes to handle flash crowds by scaling up the resources provisioned. The model of the predictive controller is tuned according to the under-provisioning of resources seen in the past a few hours. Scale down is not considered.

Gandhi et al. [10] propose a similar controller. The main difference is in the predictive controller design. Their predictive controller identifies patterns in the workload using a workload forecaster which discretizes it into consecutive, disjoint time intervals with a single representative demand value. Workload forecasting is done on the time scale of days i.e., the model of the predictive controller is changed at most once a day. In their approach there is no way to tune the model of the predictive controller and they do not consider scale down of resources.

Malkowski et al. [18] propose a controller for n-tiered applications. They add to the predictive and reactive controller a database of previous system states with good configurations. The elasticity controller starts by looking up if the current state of the system in the database. If the state is found then the configuration corresponding to the state is used. Otherwise, the reactive controller determines the underutilized state or over-utilized state and provisions resources according to the load. In addition, the predictive controller uses Fourier transforms to forecast the future workload for each tier from the past.

A much simpler approach is proposed by Calheiros et al. [5]. They model a cloud provider using basic queueing theory techniques. They assume heterogeneous requests that take constant time to process.

5 Conclusion

In this paper, we consider the problem of autonomic elasticity control for cloud infrastructures. The infrastructure is modeled as a $G/G/N$ queue with variable N. The model is used to design an adaptive proportional controller that proactively adapts based on the changes in the load dynamics. The controller takes into account resource heterogeneity, delayed requests, and variable VM service rates. A hybrid controller combines the designed controller with a reactive controller that reacts to sudden increases in the load. The combined controller tries to follow the workload envelope and avoids premature resource deallocation.

Using simulations we compare the proposed controller to a completely reactive controller. Two traces with different characteristics are used, Web traces from the FIFA world cup that are quite bursty in nature with simple requests and cluster traces from Google with jobs as long as 1 hour. Simulation results show that our proposed controller outperforms the reactive controller by decreasing the SLA violation rate

by a factor between 70 for the Web workload and 3 for the cluster one. The reactive controller required three times larger buffers compared to our controller. The results also show that the proposed controller reduces resource oscillations by a factor of seven for the Web workload traces and a factor of three for the cluster traces. As a tradeoff, the hybrid controller over-provisions between 20% and 30% resources as compared to a few percent for the reactive one.

6 Acknowledgments

We would like to thank the reviewers for their constructive comments. Financial support has been provided in part by the European Community's Seventh Framework Programme under grant agreement #257115, the Lund Center for Control of Complex Engineering Systems, and the Swedish Government's strategic effort eSSENCE.

References

[1] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *NOMS 2012, IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2012. in press.

[2] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long. Managing flash crowds on the Internet. 2003.

[3] M. Arlitt and T. Jin. "1998 world cup web site access logs", August 1998.

[4] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice-Hall, Upper Saddle River, N.J., fourth edition, 2005.

[5] R. N. Calheiros, R. Ranjan, and R. Buyya. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *International Conference on Parallel Processing (ICPP)*, pages 295 –304, sept. 2011.

[6] K. Chard, M. Russell, Y. Lussier, E. Mendonca, and J. Silverstein. Scalability and cost of a cloud-based approach to medical nlp. In *Computer-Based Medical Systems (CBMS), 2011 24th International Symposium on*, pages 1–6. IEEE, 2011.

[7] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 103–116. ACM, 2001.

[8] T. Chieu, A. Mohindra, A. Karve, and A. Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, pages 281 –286, oct. 2009.

[9] A. J. Ferrer, F. Hernandez, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgo, T. Sharif, and C. Sheridan. Optimis: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77, 2012.

[10] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *International Green Computing Conference and Workshops (IGCC)*, pages 1 –8, july 2011.

[11] T. Gunarathne, T. Wu, J. Qiu, and G. Fox. Cloud computing paradigms for pleasingly parallel biomedical applications. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 460–469. ACM, 2010.

[12] J. Herskovic, L. Tanaka, W. Hersh, and E. Bernstam. A day in the life of PubMed: analysis of a typical day's query log. *Journal of the American Medical Informatics Association*, 14(2):212, 2007.

[13] H. Khazaei, J. Misic, and V. Misic. Modelling of cloud computing centers using m/g/m queues. In *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on*, pages 87 –92, june 2011.

[14] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12(1):1–15, 2009.

[15] H. Li. Realistic workload modeling and its performance impacts in large-scale escience grids. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):480–493, 2010.

[16] H. Li and T. Yang. Queues with a variable number of servers. *European Journal of Operational Research*, 124(3):615–628, 2000.

[17] X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal. Optimal multivariate control for differentiated services on a shared hosting platform. In *46th IEEE Conference on Decision and Control*, pages 3792–3799. IEEE, 2007.

[18] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 131–140. ACM, 2011.

[19] M. Morari. Robust stability of systems with integral control. *IEEE Transactions on Automatic Control*, 30(6):574–577, 1985.

[20] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 15–28. USENIX Association, 2008.

[21] K. Ogata. *Modern control engineering*. Prentice Hall PTR, 2001.

[22] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. *SIGPLAN Not.*, 46:111–120, Mar. 2011.

[23] H. Truong and S. Dustdar. Cloud computing for small research groups in computational science and engineering: current status and outlook. *Computing*, pages 1–17, 2011.

[24] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3:1:1–1:39, March 2008.

[25] J. Vöckler, G. Juve, E. Deelman, M. Rynge, and G. Berriman. Experiences using cloud computing for a scientific workflow application. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 15–24. ACM, 2011.

[26] J. Wilkes. More google cluster data, November 2011.

[27] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. *International Conference on Autonomic Computing*, page 25, 2007.

[28] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 304–307. ACM, 2010.

Paper III

How will your workload look like in 6 years? Analyzing Wikimedia's workload

A. Ali-Eldin, A. Rezaie, A. Mehta, S. Razroevy, S. Sjöstedt-de Luna, O. Seleznjev, J. Tordsson, and E. Elmroth

In Proceedings of the 2014 IEEE International Conference on Cloud Engineering (IC2E), pages 349-354, IEEE Computer Society, 2014.

How will your workload look like in 6 years? Analyzing Wikimedia's workload*

Ahmed Ali-Eldin[†], Ali Rezaie[‡], Amardeep Mehta[†],
Stanislav Razroev[‡], Sara Sjöstedt-de Luna[‡], Oleg Seleznjev [‡],
Johan Tordsson[†], and Erik Elmroth[†]

Abstract

Accurate understanding of workloads is key to efficient cloud resource management as well as to the design of large-scale applications. We analyze and model the workload of Wikipedia, one of the world's largest web sites. With descriptive statistics, time-series analysis, and polynomial splines, we study the trend and seasonality of the workload, its evolution over the years, and also investigate patterns in page popularity. Our results indicate that the workload is highly predictable with a strong seasonality. Our short term prediction algorithm is able to predict the workload with a Mean Absolute Percentage Error of around 2%.

1 Introduction

The performance of an Internet-scale system is affected by three main factors: the design of the system, the implementation of the system, and the load on the system [12]. When a system is designed, there is typically an underlying assumption of the load range for the system operation. On the other hand, the actual workload, and thus the performance, is only known when the system becomes operational, turning system design and workload analysis into a chicken-and-egg problem. One way to design a new system is to analyze workloads of existing (similar) systems and draw similarities and possible discrepancies between the existing system and the new system.

*The paper has been re-typeset to match the thesis style. Reproduced with the permission of IEEE.

[†]Department of Computing Science, Umeå University, Sweden, email: {ahmeda, amardeep, tordsson, elmroth}@cs.umu.se

[‡]Department of Mathematics and Mathematical Statistics, Umeå University, Sweden, email: {alaree02, stvrav01}@student.umu.se and {sara,oleg.seleznjev}@math.umu.se

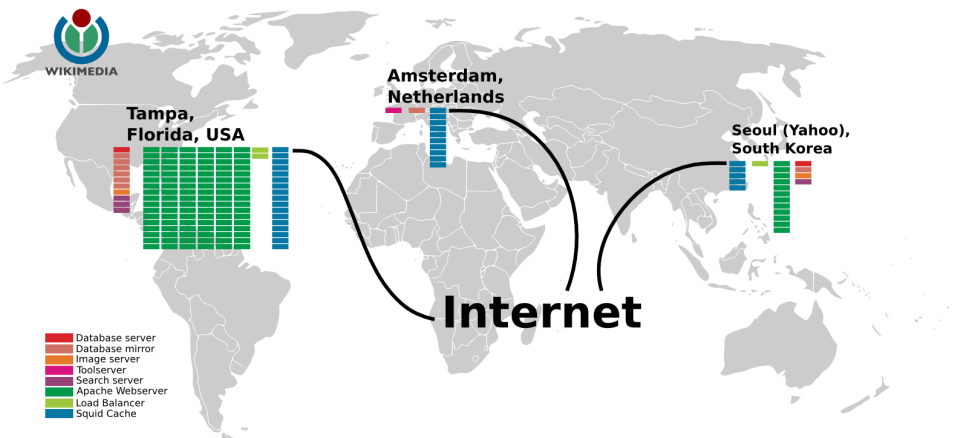


Figure 1: Wikimedia foundation's servers (Source: Wikimedia Foundation).

Workload analysis is also important for differentiating between legitimate and illegitimate traffic [15], cache design [6], and capacity planning [22]. Within cloud resource management [17], problems related to admission control [24], scheduling [18], virtual machine migration [23] and elasticity [3] are all examples that require deep understanding of the expected operating workloads in order to design efficient management algorithms. As a prerequisite for designing better algorithms, we analyze the workload of a large-scale website, which is a typical application for the cloud. The selected workload is from the Wikimedia foundation servers, most known for operating Wikipedia, the sixth most popular site on the web [2]. While three months of this workload has been analyzed previously [25], we analyze a much larger data-set spanning the period between June, 2008 until October, 2013, making our study the largest workload study we are aware of.

This paper contributes to understanding the evolution of an Internet-scale online service for a period of roughly 5.25 years. We describe the workload and introduce statistical models of both its long-term evolution and its short-term trends. The short term model is useful for management tasks such as admission control and auto-scaling. In addition, we analyze and describe the workload dynamics on popular pages. Knowledge about object popularity is useful for designing algorithms for scheduling and caching.

Table 1: *Number of pages on Wikipedia in September each year (in millions).*

2008	2009	2010	2011	2012	2013
11.3	13.9	16.7	19.8	23.2	29.6

2 Workload description

2.1 Overview

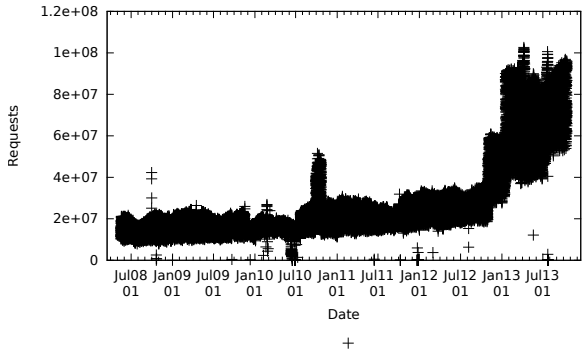
During the five years period we study, Wikipedia kept evolving, e.g., Table 1 shows the total number of pages in Wikipedia through our study. The underlying Wikipedia server architecture also evolved. At the beginning of our study, the foundation was operating all its projects including Wikipedia using around 300 Servers in Florida, which acted as the primary site, 26 in Amsterdam and 23 in Korea as shown in Figure 1. In January, 2013, the foundation was running around 885 servers and building a new cluster in San Francisco that went in production in March, 2013. Today, the foundation's servers are distributed on 5 different sites; Virginia, which acts as a primary site, Florida and San Francisco in the United States and two cluster in Amsterdam, the Netherlands.

The dataset studied consists of hourly logged files [1] where each file contains the total number of requests directed to a page hosted on the Wikimedia foundation's servers, the page's name, to which project it belongs and the total amount of bytes transferred for the page. The files sizes are between 20 MB and 120 MB of compressed data. While the logging started in December, 2007, there was a problem in the reported data transfered values until May, 2008. Since our plan is to study both the total number of requests sent per hour and also the data transfered, we start our analysis from May, 2008 till October, 2013, i.e. roughly 47479 hours (and files) studied. Due to monitoring errors and system failures, 424 files are missing. All monitoring errors lasted for a few consecutive hours, with the longest failure period between the 21st of September, 2009 until the 1st of October, 2009.

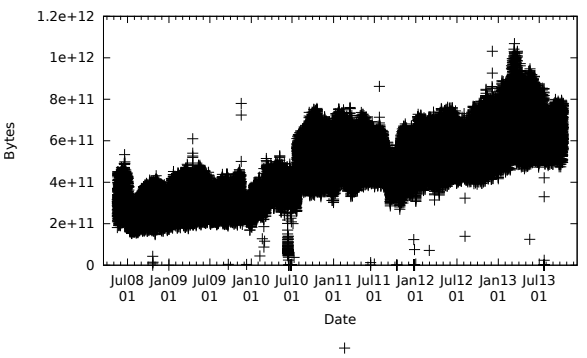
2.2 Global workload characteristics

Figure 2 shows the aggregate load on Wikimedia's servers. In both figures 2(a) and 2(b), there are clear dips where the number of requests and the data sent goes to zero or are reduced significantly. Further investigation showed that these are mostly times of, or preceding, periods of (monitoring) system failures or server overloads. The upper bursts on the other hand represent real increases in the system workload.

We used time-series analysis to model the workload request flow. In the classical decomposition model [9], an observed time series $\{x_t, t \in T\}$ is modeled as a real-



(a) Total number of requests received per hour.



(b) Total amount of data (bytes) sent per hour.

Figure 2: Load on the Wikimedia foundation's servers.

ization of a sequence of random variables $\{X_t, t \in T\}$, $X_t = T_t + S_t + R_t$, where T_t is a slowly changing function (the *trend* component), S_t is a periodic function with known period m (the *seasonal* component), and R_t is a random noise component (assumed to be a stationary time series [10]). For modeling the request flow, we use (seasonal) Autoregressive Integrated Moving Average (ARIMA) models. A seasonal ARIMA$(p,d,q)(P,D,Q)_m$ (m-periodic) process $\{Y_t\}$ is defined by

$$\Phi(B^m)\phi(B)(1-B^m)^D(1-B)^d Y_t = \Theta(B^m)\theta(B)Z_t,$$

where $\phi(z), \theta(z), \Phi(z), \Theta(z)$ are polynomials of order p, q, P, Q, respectively, with no roots inside the unit circle, B is the backshift operator $B(Y_t) = Y_{t-1}$, $\{Z_t\}$ is white noise with zero mean and variance σ^2.

It is clear from Figure 2(a) that the workload dynamics changed tremendously during the period studied with visible *steps*, e.g., at the end of 2012 and early 2013.

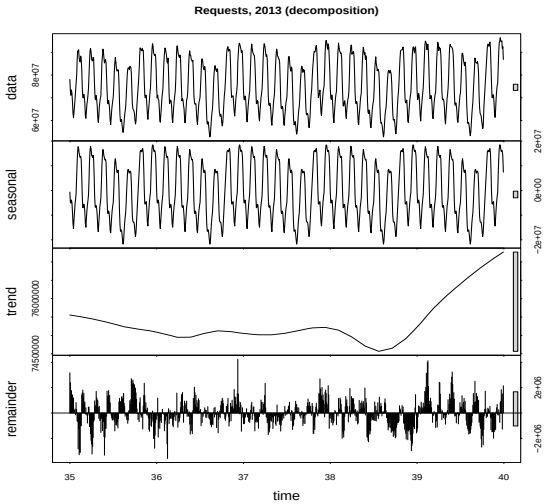


Figure 3: Workload, Wikipedia, 3 September-8 October 2013, Seasonal-Trend decomposition.

This suggests a change in the underlying process of the workload. In this case trying to build a single global model can be deceiving and inaccurate. Instead of building a single global model, we modeled smaller periods of the workload where there was no significant step. Figure 3 shows the decomposition of the request flow for the period between the 3rd of September till the 8th of October, 2013. The seasonality graph suggests high daily and weekly correlations between the number of requests during this period. The trend graph can be approximated with piecewise-linear functions. We fitted the remainder to a seasonal ARIMA(2, 0, 2)(0, 0, 1) model using the R forecast package [13].

Figure 4 shows some diagnostic plots of the fit, namely, the standardized residuals, autocorrelation for residuals, and
p-values for Ljung-Box statistics for the fitted model [9]. While most of the standardized residuals are well inside the [-2, 2] interval, in some cases they are outside. The autocorrelation function (ACF) also suggests some daily correlation still present in the residuals, so the fitted model could still be improved. We leave this for future work.

We have repeated similar analysis for different (regular) parts of the workload and summarized our results in Table 2. The first column is the period studied. The second column summarizes the trend characteristics. For most parts of the workload, the variance in the trend is less than 5% of the average value. The second column summarizes

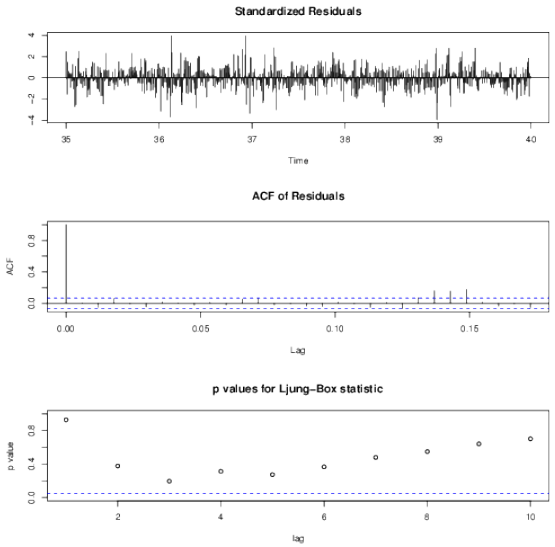


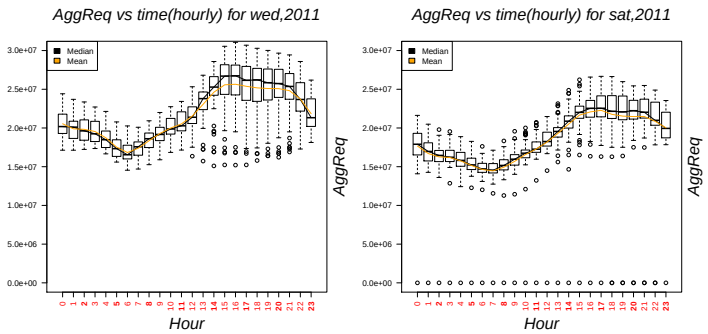
Figure 4: Workload, Wikipedia, 3 September-8 October 2013, diagnostics for ARIMA fitting.

the workload seasonality. For each seasonality plot, we have computed the FFT. The only two dominant frequencies are at 1 day and 1 week. Their amplitudes are also noted. The last column describes the fits for the remainders. The remainders are fitted using seasonal ARIMA models. We note that the models are of low order and are not too far from each other with even some models repeating, e.g., ARIMA(1,0,2)(0,0,1) is repeated 4 times. We leave further investigation of the relationship between the residuals and the goodness of fits for future work.

To better understand the differences between the load on the servers during weekends vs. weekdays, Figure 5 shows box-plots for the aggregate hourly number of requests for all Wednesdays in 2011 and all Saturdays in 2011. A box-plot is a way to visualize the quartiles and the dispersion of the distributions of the data [19]. The medians and the means for the different hours are plotted. It is clear from the figure that the time of the day affects the number of requests significantly with the lowest activity at 5, 6 and 7 a.m. and the highest activity during the afternoons and the nights. The data dispersion is also affected by the time of the day and the day of the week. More accurate predictions can be done at times with lower dispersion. While there are discrepancies between the mean and the median especially at hours of higher activity, which might suggest skewed distributions, such a claim is incorrect to make without

Table 2: *Seasonal-Trend decomposition for some Workload intervals, 2008-2013, 5 weeks intervals, weekly (168 hours) seasonal ARIMA model*

Period	Trend	Seasonal	Remainder
(y, d/m)	min, mean, max (10^6)	periods/amplitudes (days)/(10^6)	ARIMA[168]
2008, 17/05-21/06	15.01, 15.14, 15.28	1.00/6.57, 6.74/2.89	(1,0,2)(0,0,1)
2008, 31/07-04/09	12.49, 13.36, 13.86	1.00/4.40, 7.26/2.35	(1,0,2)(0,0,1)
2008, 26/10-30/11	14.58, 15.22, 15.63	1.00/7.57, 7.26/2.14	(1,0,0)(1,0,0)
2009, 09/01-13/02	15.42, 16.13, 16.33	1.00/7.84, 6.74/2.26	(1,0,0)(1,0,1)
2009, 23/04-28/05	15.68, 16.28, 16.89	1.01/7.52, 7.26/2.90	(1,0,2)(0,0,1)
2009, 17/05-21/06	15.40, 16.10, 16.50	0.99/6.56, 7.14/2.87	(2,0,1)(0,0,1)
2010, 28/07-01/09	18.50, 19.08, 20.01	1.01/6.67, 7.26/3.03	(1,0,2)(0,0,1)
2010, 20/08-24/09	19.24, 19.87, 20.63	1.00/8.18, 6.85/3.14	(2,0,2)(1,0,1)
2011, 01/01-04/02	18.73, 21.34, 22.19	1.00/10.20, 6.74/2.53	(3,0,3)(1,0,1)
2011, 20/06-25/07	19.34, 19.75, 20.39	1.00/6.90, 7.26/2.44	(1,0,3)(0,0,0)
2011, 01/11-06/12	22.98, 23.35, 23.78	1.01/9.71, 7.26/2.94	(1,0,0)(1,0,0)
2012, 01/01-04/02	23.01, 25.15, 25.85	1.01/10.14, 6.74/2.05	(3,0,3)(0,0,0)
2012, 07/03-11/04	23.09, 23.73, 25.23	1.01/9.62, 7.26/2.81	(1,0,0)(1,0,1)
2012, 25/08-29/09	25.85, 26.56, 27.23	1.00/9.39, 6.74/2.89	((1,0,1)(1,0,1)
2013, 19/01-23/02	65.87, 66.83, 67.79	1.01/33.96, 6.74/9.77	(1,0,2)(1,0,1)
2013, 15/04-20/05	60.09, 61.07, 64.24	1.01/30.09, 6.74/8.81	(1,0,0)(0,0,1)
2013, 03/09-08/10	74.57, 75.33, 77.27	0.99/26.56, 6.93/6.56	(2,0,2)(0,0,1)



(a) Time of the day for Wednesdays vs aggregate request distribution.

(b) Time of the day for Saturdays vs aggregate request distribution.

Figure 5: Time of the day/week effect on the aggregate number of requests and their distribution (Year 2011).

further analysis that we leave for future work [26]. Similar results were obtained for different days and years for both the number of requests and the sent data.

2.3 Top 500 popular pages

During peak hours, less than one third of Wikimedia's pages are accessed while during hours of lower activity around one sixths of all pages are accessed. This pattern did not change for the period of study. To better understand the dynamics of the workload on individual pages, we keep track of the top 500 pages having highest number of requests for the period of study. This is a highly dynamic list with over 650000 pages joining the list for sometime during the period of the study. The least accessed page in the top 500 list had on average less than 1200 page views per hour for the period of the study, i.e, less than 20 page views per minute. On the other hand, the most popular pages in the list where usually general pages, e.g, the English Main Page. These pages had on average more than 500000 page views at the beginning of our study and around 20 Million views at the end.

Figure 6 shows the histogram of the number of consecutive hours a page stays popular before leaving the list. Most pages have volatile popularity, with 41.58% of the top 500 pages joining and leaving the top 500 list every hour, 87.7% of them staying in the top 500 list for 24 hours or less and 95.24% of the top-pages staying in the top 500 list for a week or less. The distribution has a long tail. Since there are some monitoring gaps in the workload, we were not able to infer which pages were in the top 500 list during these gaps. This adds some uncertainty to the preceding results. In order to reduce this uncertainty, we plot Figure 7 which shows the percentage of

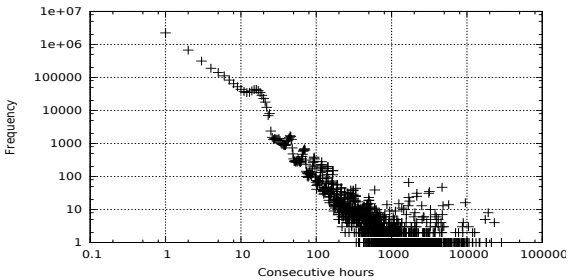


Figure 6: Time a page stays in the top 500 list.

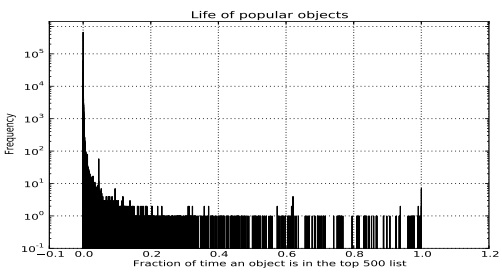


Figure 7: Ratio between life of an object in the top 500 list and the total time of the study.

time a page is in the top list during the study period. The x-axis represents the ratio between the total time an object stays in the top 500 list and the total time of the study while the y-axis shows the frequency of objects with a certain ratio. Around 9 pages where in the top 500 list for the whole period of the study, these are mostly the main pages for different Wikipedia projects. On the other hand, Figure 7 confirms that most objects are popular for only short periods of time.

2.4 Load on a popular page

We now shift our focus towards studying the effect of major events on the workload on associated pages. We focus on two major events, Michael Jackson's death and the Super bowl XLV. While the load on Michael Jackson's page has been studied for the period of two days after his death [8], we study a much longer period. For both events, we report results for the *collateral load* that accompanied the load on the main page. We define the collateral load as the load increase on pages associated with the page of the main event excluding the load on the page dedicated to that event. In order to

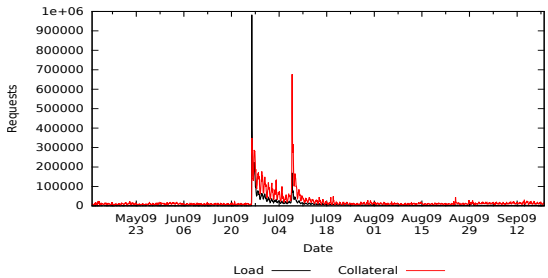


Figure 8: Load on Michael Jackson's page and the collateral load on the servers.

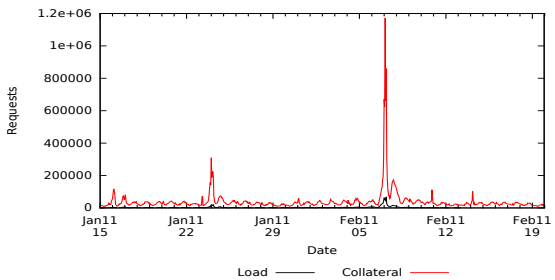


Figure 9: Load on the Super Bowl XLV page and the collateral load on the servers.

extract the collateral load, we parse the main event's page for the webpages that it links to.

Figure 8 shows the workload on Michael Jackson's page and the collateral load. When Michael Jackson died, the load on his page increased by around four orders of magnitude. This increase was accompanied by an increase in the load on all pages that his page linked to, but with a smaller yet significant amplitude. Both the load and the collateral load started decreasing shortly after a few hours but with the collateral load decreasing slower than the load. After 12 days, on the 7th of July, Michael Jackson's memorial service took place, resulting in another significant load spike on the load on his Wikipedia entry, but in a much larger spike in the collateral load.

Figure 9 shows the load on the Super Bowl XLV page and the collateral workload on the Wikimedia servers before and after the event. Although the main event was the Super Bowl, the load spike was in the collateral workload dwarfing the spike on the Super Bowl page. We obtained similar results for the FIFA 2010 worldcup, the Eurovision 2010 and the Egyptian revolution articles where for all of them the collateral workload was typically orders of magnitude than the load on the main article. This phenomena seems to be common for planned events, where the collateral load surpasses the load on the original item. We plan to study the collateral load effect

in more details and find its implication on resource management problems such as placement and resource provisioning.

3 A predictive workload model

As seen in Figure 3 and confirmed by Table 2, the workload (Y_t) has a pronounced repetitive weekly pattern, which slowly varies over time. We call this the (weekly) pattern, P_t, and estimate it by fitting cubic splines to the data [20]. Cubic splines are flexible and able to pick up the features of the pattern. The workloads' deviation from the pattern ($res_t = Y_t - P_t$) is called the residual and will typically have a positive autocorrelated structure that also brings information about future values. It may be captured by an autoregressive model, e.g. in the simplest case with lag structure one,

$$res_{t+1} = a_0 + a_1 res_t + e_{t+1}, \tag{1}$$

where e_t is white noise.

The workload is also characterized by occasional outliers, having extremely large or small values compared to the surrounding workload values. As discussed earlier, outliers typically come in time consecutive groups. Downward outliers occur due to monitoring or system problems and do not reflect the true workload. Using bogus monitoring values for management decisions can lead to severe decrease in the application performance. They should therefore be ignored and the corresponding (unobservable) workload be predicted by the estimated pattern. Upward outliers (large values) on the other hand are real workload increases and should be predicted as accurately as possible. While it is not possible to predict a completely random event, i.e, foresee the first upward outlier in a group, it is important for the predictor to adapt quickly and start mitigating for the workload change. We thus predict the next workload value by the estimated pattern plus the prediction error of the last workload,thus aiming at catching up as fast as possible to the "explosive" nature of the upward outliers. If the last value was not an outlier, we predict the next workload by the estimated pattern plus the residual estimated from the autoregressive model in Equation 1.

The repetitive weekly pattern slowly changes with time in amplitude, level and shapes. Thus, when estimating the pattern and the residual AR model, and when identifying outliers, it has to be done locally, say using only the two last weeks of workload. Moreover the pattern and the AR model should be estimated without the influence of outliers. We propose to do the following: Let Y_t^* denote the true workload Y_t if it is not an outlier (to be defined), and let it correspond to the estimated pattern $\hat{P}_t$ otherwise. Suppose we have chosen a two week window of workload data without outliers, $Y_{t-335}^*, ..., Y_t^*$ and want to predict the workload at time $t+1$. Estimate the weekly pattern Y_t, by first overlaying the two weeks of data on top of each other,

such that there are two workloads for each hour over a week. Fit a cubic spline to these data, putting knots at every second hour of a week (87 knots in total over 168 hours of a week) and using B-splines as basis functions. Now compute the residuals, $res^*_h = Y^*_h - \hat{P}_h$, $h = t-335,...,t$, noting that $\hat{P}_h = \hat{P}_{h+168}$, and estimate the coefficients of the AR model using Equation 1 by a least squares fit of $res^*_{t+1} = a_0 + a_1 res^*_t$.

If the last workload, Y_t, is not an outlier, predict the next workload, Y_{t+1}, by

$$\hat{Y}_{t+1} = \hat{P}_{t+1} + \hat{a}_0 + \hat{a}_1(Y_t - \hat{P}_t). \tag{2}$$

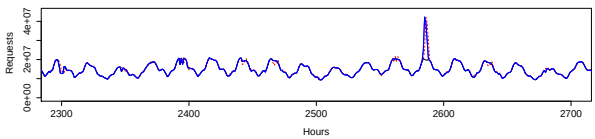
If Y_t is an upward outlier predict Y_{t+1}, by

$$\hat{Y}_{t+1} = \hat{P}_{t+1} + (Y_t - \hat{P}_t). \tag{3}$$

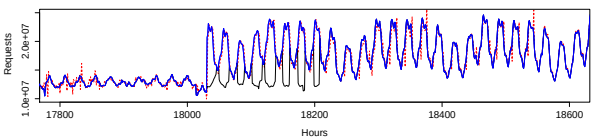
If Y_t is a downward outlier predict Y_{t+1}, by

$$\hat{Y}_{t+1} = \hat{P}_{t+1}. \tag{4}$$

Outliers are defined as follows. First compute the standard deviation s_{res} of the residuals res^*_h, $h = t-335,...,t$. Suppose that we go forward in time and observe Y_{t+1}. Then Y_{t+1} is defined to be an upward outlier if Y_{t+1} is greater than the maximum value of the estimated pattern (over the last two weeks) plus $4s_{res}$. Moreover, Y_{t+1} is defined to be a downward outlier if it is zero or less than the minimum value of the estimated pattern (over the last two weeks) minus $4s_{res}$. To predict the workload at time $t+2$, having observed Y_{t+1}, we slide the two week window one hour ahead in time and repeat the above algorithm. The predictive model was applied to both the flow of requests and the data flow for the Wikimedia workload. Examples of prediction performance for the request flow can be seen in Figure 10. The actual workload is shown in blue solid lines, the predicted value in red dotted lines and the predicted value without compensation for the bursts in solid black lines. In Figure 10(a), there is a single short spike where the workload almost doubled, while in Figure 10(b), the workload almost doubled but stayed like that. It is clear that using our corrective mechanism, the predictor is able to rapidly cope to the changing workload dynamics. Comparable results were obtained for predicting the data flow which we omit due to the lack of space. We calculated the Mean Absolute Percentage Error (MAPE) for our predictions for both the requests flows and the data flows. When excluding outliers, the MAPE for the predicted request flow is 2.2% while for the data flow is 2.3%, that is, on average we miss the true workload one step ahead by around 2% for both the data and request streams. When only considering spikes, the MAPE for the request stream is 4.6% and is 11% for the data stream.



(a) Predictions for the period between 21st of Aug. 2008 till the 6th of September, 2008.



(b) Predictions for the period between 29th of May 2010 till the 12th of Aug., 2010.

Figure 10: Load on the Wikimedia foundation's servers.

4 Related work

Several projects have analyzed in depth publicly available workloads. All of them differ from our current work in either the type of the workload analyzed, the period of the analysis or the magnitude of the workload studied [5, 7, 14, 21].

Urdenta et. al. [25] studied 10% of all requests directed to all the wiki projects for a period of 107 days between September 19th, 2007 to January 2nd, 2008. They give more focus to the load directed to the English Wiki. They report some statistics describing page popularity, save operations and other Wiki operations. .

Kang et. al. crawled Yahoo! videos website for 46 days [16]. Around 76% of the videos crawled are shorter than 5 minutes and almost 92% are shorter than 10 minutes. They discuss the predictability of the arrival rate with different time granularities. A load spike typically lasted for no more than 1 h and the load spikes are dispersed widely in time making them hard to predict.

Arlitt et. al. [6] analyzed 6 different web server access logs with different durations. The workloads varied in magnitude of server activity, ranging from a total of 653 requests per day to 355787 requests per day. All workloads studied were logged between October 1994 and August 1995. They predicted that video and audio workload growth may have a dramatic impact on the web servers. They noted 10 invariants in the 6 workloads. Ten years later, Williams et. al. [27] repeated the analysis for three of the workloads used in the original study. The most notable difference they noted was the increase in web traffic volume, sometimes up to 32 folds. With 7 out of the

10 invariants studied in 1997 unchanged, they conclude that there are no dramatic changes in webserver workload characteristics between 1995 and 2005. Faber et. al [11] have also reexamined the 10 invariants for scientific websites. They studied access logs for 4 websites different from the ones studied in [6]. They found that only 3 invariants were unchanged for the scientific website. These results are not aligned with the results in [27] suggesting that their might be different sets of invariants for different workloads [4].

Bodik et. al. [8] identify volume spikes as spikes that result from a sudden increase in the total workload of a system. They define a data spike as a sudden increase in demand for certain objects, or more generally, a pronounced change in the distribution of object popularity on the site. They analyze spikes in different workloads and propose a methodology for capturing both volume and data spike with a statistical seven-parameter model. The authors also propose a closed-loop workload generator based on the proposed model.

5 Conclusion

We analyze and describe one of the largest publicly available workloads, the Wikimedia foundation's load. The top 500 pages by number of page accesses were studied. We model different parts of the workload using time-series analysis. Major events can cause a collateral load on the servers of the system resulting from pages related to the event's page. This collateral load can be orders of magnitude larger than the load on the original page. We have also introduced a simple prediction algorithm that can be used to predict short term aggregate workload on the system. Our results show that, although the Wikimedia workload has complex patterns and dynamics, a simple cubic spline prediction algorithm can predicts it with MAPE of 2% suggesting that complex prediction algorithms are not needed for the Wikimedia workload. For future work, we will further investigate how to build a workload generator for Wikipedia like systems with a similar popularity model. We also plan to find better spike detection algorithms for short term predictions.

6 Acknowledgment

Financial support has been provided in part by the Swedish Government's strategic effort eSSENCE and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control.

References

[1] Page view statistics for wikimedia projects. Accessed: May, 2013, URL: http://dumps.wikimedia.org/other/pagecounts-raw/.

[2] Top Sites. Accessed: November, 2013, URL: http://www.alexa.com/topsites.

[3] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *ACM ScienceCloud*, pages 31–40. ACM, 2012.

[4] A. Ali-Eldin, J. Tordsson, E. Elmroth, and M. Kihl. Workload classification for efficient auto-scaling of cloud resources. 2013.

[5] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, 2000.

[6] M. F. Arlitt and C. L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM ToN*, 5(5):631–645, 1997.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[8] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *ACM SoCC*, pages 241–252. ACM, 2010.

[9] P. J. Brockwell and R. A. Davis. *Time series: theory and methods*. Springer, 2009.

[10] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning. Stl: A seasonal-trend decomposition procedure based on loess. *Journal of Official Statistics*, 6(1):3–73, 1990.

[11] A. M. Faber, M. Gupta, and C. H. Viecco. Revisiting web server workload invariants in the context of scientific web sites. In *ACM/IEEE SC*, page 110. ACM, 2006.

[12] D. G. Feitelson. Workload modeling for computer systems performance evaluation. *Book Draft, Version 0.41*, 2013.

[13] R. J. Hyndman and Y. Khandakar. Automatic time series forecasting: the forecast package for R. *Journal of Statistical Software*, 26(3):1–22, 2008.

[14] M. Jeon, Y. Kim, J. Hwang, J. Lee, and E. Seo. Workload characterization and performance implications of large-scale blog servers. *ACM TWEB*, 6(4):16, 2012.

[15] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *ACM WWW*, pages 293–304. ACM, 2002.

[16] X. Kang, H. Zhang, G. Jiang, H. Chen, X. Meng, and K. Yoshihira. Understanding internet video sharing site workload: A view from data center design. *Journal of Visual Communication and Image Representation*, 21(2):129–138, 2010.

[17] M. Kihl, E. Elmroth, J. Tordsson, K.-E. Årzén, and A. Robertsson. The Challenge of Cloud Control. 2013.

[18] W. Li, J. Tordsson, and E. Elmroth. Modeling for dynamic cloud scheduling via migration of virtual machines. In *IEEE CloudCom*, pages 163–171. IEEE, 2011.

[19] R. McGill, J. W. Tukey, and W. A. Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[20] J. O. Ramsay. *Functional data analysis*. Wiley Online Library, 2006.

[21] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC*, pages 7:1–7:13, 2012.

[22] M. Stokely, A. Mehrabian, C. Albrecht, F. Labelle, and A. Merchant. Projecting disk usage based on historical trends in a cloud environment. In *ScienceCloud*, pages 63–70. ACM, 2012.

[23] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. *ACM Sigplan Notices*, 46(7):111–120, 2011.

[24] L. Tomas and J. Tordsson. Cloudy with a chance of load spikes: Admission control with fuzzy risk assessments. In *IEEE/ACM UCC*, 2013.

[25] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[26] P. T. von Hippel. Mean, median, and skew: Correcting a textbook rule. *Journal of Statistics Education*, 13(2):n2, 2005.

[27] A. Williams, M. Arlitt, C. Williamson, and K. Barker. Web workload characterization: Ten years later. In *Web content delivery*, pages 3–21. Springer, 2005.

Paper IV

Measuring Cloud Workload Burstiness

A. Ali-Eldin, O. Seleznjev, S. Sjöstedt-de Luna, J. Tordsson, and E. Elmroth

IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC), pages 566 - 572, IEEE, 2014.

Measuring Cloud Workload Burstiness*

Ahmed Ali-Eldin†, Oleg Seleznjev‡,
Sara Sjöstedt-de Luna‡, Johan Tordsson†, and Erik Elmroth†

Abstract

Workload burstiness and spikes are among the main reasons for service disruptions and decrease in the Quality-of-Service (QoS) of online services. They are hurdles that complicate autonomic resource management of datacenters. In this paper, we review the state-of-the-art in online identification of workload spikes and quantifying burstiness. The applicability of some of the proposed techniques is examined for Cloud systems where various workloads are co-hosted on the same platform. We discuss Sample Entropy (*SampEn*), a measure used in biomedical signal analysis, as a potential measure for burstiness. A modification to the original measure is introduced to make it more suitable for Cloud workloads.

1 Introduction

Workload spikes and burstiness decrease online applications' performance and lead to reduced QoS and service disruptions [6, 34]. We define a workload spike (sometimes referred to as a burst or a flash crowd [5]) to be a sudden increase in the demand on an object(s) hosted on an online server(s) due to an increase in the number of requests and/or a change in the request-type mix [29].

Some spikes occur due to a non-predictable event in time with non-predictable load volumes while others occur due to a planned event but with non-predictable load volumes. Figure 1 shows the load on Michael Jackson's English Wikipedia page during the period around his death. Two significant spikes can be seen in the figure. The first spike occurred right after his death with the load on the page increasing by

*The paper has been re-typeset to match the thesis style. Reproduced with permission of IEEE.

†Department of Computing Science, Umeå University, Sweden, email: {ahmeda, tordsson, elmroth}@cs.umu.se

‡Department of Mathematics and Mathematical Statistics, Umeå University, Sweden, email: {oleg.seleznjev, sara}@math.umu.se

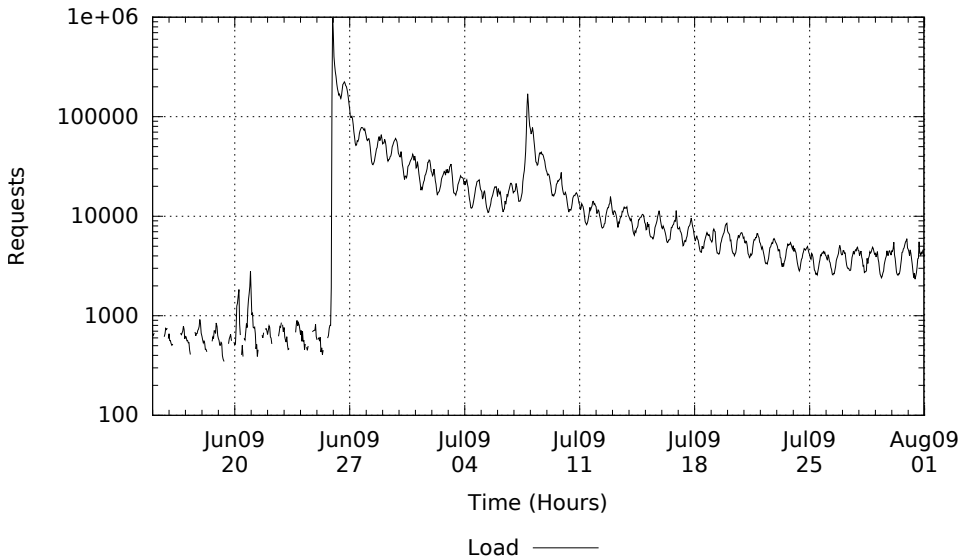
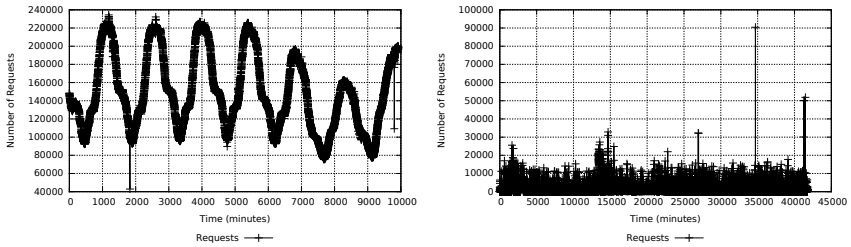


Figure 1: Spike on the Michael Jackson Wikipedia entry after his death.

three orders of magnitude. The second visible spike occurred 12 days after his death during his memorial service. Before the memorial service, the load on the page was still higher than normal by one order of magnitude. The memorial service resulted in another increase, one order of magnitude larger than the load before the service. While it is impossible to predict the first spike, it is important to detect and mitigate against the spike with minimal reduction in the QoS. The second spike on the other hand can be mitigated against before its occurrence since it is a planned event.

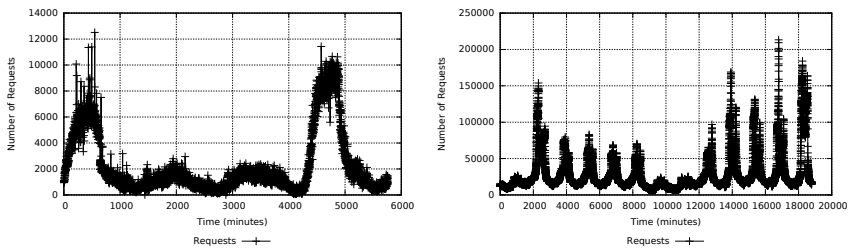
A *bursty* workload is a workload having a significant number of spikes. The spikes make it harder to predict the future value of the load. An example of a workload that exhibits little burstiness is shown in Figure 2(a) for 10% of all user requests issued to Wikipedia [30]. Most of the variations in the load are due to the diurnal variations in the usage of Wikipedia. Figure 2(c) shows the load on IR-Cache servers [18] deployed in San Diego (SD-IRCache) and Figure 2(d) shows the load on the servers of the FIFA 1998 world cup during the last two weeks before the games ended, with both workloads having moderate burstiness. These bursts are either due to planned events with hard to predict magnitude or just increased service usage. Figure 2(b) shows strong burstiness in the number of tasks submitted per minute to a Google cluster [36].

Bursty workloads complicate cloud resource management since cloud providers host a multitude of applications with different workloads in their datacenters. Prob-



(a) Workload with little burstiness (Wikipedia).

(b) Bursty workload (Tasks in a Google cluster).



(c) Workload with spikes that have a hard to predict magnitude (IRCache servers).

(d) Workload with spikes that have a hard to predict magnitude (FIFA world cup servers).

Figure 2: Different workloads have different burstiness.

lems such as service admission control, Virtual Machine (VM) placement, VM migration and elasticity [14] are examples of resource management problems that are complicated due to workload spikes and burstiness. Our interest in quantifying burstiness is driven by our work on cloud elasticity [3]. Workload burstiness has a profound effect on the performance of any autoscaling algorithm. Some elasticity algorithms are better suited for periodic workloads, where the repetitive workload patterns can be used for estimating the future load [2]. Other algorithms are better performing on bursty workloads without repetitive patterns [3]. Since there is no one-size-fits-all solution for workload predictions for elasticity control, we needed a measure for burstiness that is able to compare and classify workloads [4].

We identified some requirements for a burstiness metric to be robust and work on a wide range of scenarios.

1. The metric should be able to capture changes in a wide set of workload types. For example, using the exponential increase will result in missing out bursts where the required number of servers increases from 1000 servers to 1500 servers.

2. The parameters used for calculating the metric should be intuitive, and therefore easy to set.
3. The metric should be able to operate on short data sequences and to be fast to compute.
4. The metric should differentiate between a gradual workload increase and a sudden one. For example, techniques using entropy are not able to do that.

This work describes our search for a measure of burstiness that is able to differentiate between different workloads while fulfilling the above requirements. Section 2 discusses the state-of-the-art in both burst detection and quantifying burstiness. While burst detection can be considered as a different problem from quantifying workload burstiness, we include some of the techniques for burst detection and discuss how we considered modifying these to be able to quantify bursts. The limitations of the various reviewed measures are discussed. We propose a measure, *AvgSampEn*, for quantifying burstiness based on *SampEn*, a measure used for quantifying abnormalities in physiological signals. Section 3 explains the limitations of *SampEn* and our modifications to make it suitable for cloud workloads. We compare the proposed measure with the state-of-the-art in quantifying burstiness.

2 State-of-the-art

The problem of burst detection has been studied in many contexts including bursts in human communications, neuron spike trains, and seismic signals [13, 15, 19]. We have surveyed a large body of work on signal analysis, time-series analysis, workload analysis, spike detection and modeling, workload generation and autonomic management. Unfortunately, all the surveyed papers do not agree on a single definition of what a true burst is. We believe that the techniques discussed in this section are a representative set of the state-of-the-art. While visual inspection has been used for workload spike detection [6, 32], we focus on automated techniques for detecting spikes and quantifying burstiness.

2.1 Detecting bursts

2.1.1 Index of dispersion

The index of dispersion (I) is a measure of burstiness used in communication networks [12]. It was used by Caniff et al. [7] to detect bursts in server workloads. Given a time-series with inter-arrival times (weakly stationary), I is defined as,

$$I = SCV(1 + 2\sum_{k=1}^{\infty} \rho_k), \tag{1}$$

where *SCV* is the squared-coefficient of variation (the ratio between the variance and the mean squared) and ρ_k is the lag-k autocorrelation coefficient. Since it is not practical to calculate the autocorrelation coefficient for the inter-arrival time series at ∞, the authors use a batch of $K = arrival\_rate \times C$ requests, where $arrival\_rate$ is the request arrival rate and C is a constant which they set to 2000 in their design. The authors substitute the infinite sum in the exact equation by the first 10% of the batch size. To detect the start and end of a spike, they use the algorithm shown in Algorithm 1. The algorithm detects a burst by comparing the changes in I through time. While choosing the different parameters of the algorithm is not intuitive, e.g., C, they can be set by the application owners.

Algorithm 1: Using the index of dispersion for burst detection.

for *current batch of K requests* **do**
  current_I← calculate I for current batch
  batch_rate← arrival rate for current batch
  total_rate← update average arrival rate (all requests)
  SCV ← update *SCV* (all requests)
  if $(|current\_I\text{-}old\_I| > 2 \times SCV)$ *AND* $(\frac{old\_I}{current\_I} > 2$ *OR* $\frac{old\_I}{current\_I} < 0.5)$ **then**
    if $batch\_rate > total\_rate$ **then**
      burst starts
    else
      burst ends
  old_I← current_I

2.1.2 Exponential increase

Wendel and Freeman [35] define a *flash crowd* as a period over which request rates increase exponentially. If r_{t_i} is the average request rate per unit time over a period t_i, then a spike should satisfy the following conditions,

$$r_{t_i} > 2^i \times r_{t_0}, \forall i \in [0,k], \tag{2}$$

and,

$$max_i(r_{t_i}) > m \quad and \quad max_i(r_{t_i}) > n \times r_{avg}, \tag{3}$$

where m is the minimum request rate required to consider the workload increase as a spike, and k is the least sustained modest period of continuous workload increase required to consider a change in the request rate to be a spike. The constant n specifies how much increase in the maximum sustained rate must be achieved compared to the

service's average rate, r_{avg}, in order to consider the increase a spike. A spike is over when

$$2^{-1} \times r_{t_j} < r_{t_{j+1}} < 2 \times r_{t_j} \tag{4}$$

The choice of the three parameters, k, m and n, is arbitrary and in most cases application dependent. These three parameters can be chosen by the application owner or after profiling an application before its deployment. The main limitation of this method is the assumption that a spike should have exponential growth. One example where spikes were not increasing exponentially is the caching servers' workload in Figure 2(c) where some spikes occurred during the first day where the load increased by less than a factor of two. Another example is large systems serving millions of requests. For a system serving more than 0.5 million requests per minute like Wikipedia, an increase of as little as 10% in the total load can be considered as a spike [2].

2.1.3 Standard deviation from the moving average

Algorithm 2: Using the standard deviation from the moving average for burst detection.

1 Calculate Moving Average MA_w over a period of w time units
2 Calculate $Threshold \leftarrow (MA_w) + x \times std(MA_w)$
3 bursts$\leftarrow \{|MA_w(i)| > Threshold\}$

Vlachos et al. [31] use the simple algorithm shown in Algorithm 2 to detect bursts in online search queries. The algorithm calculates the moving average and the standard deviation of a workload over a window of w time units. A burst is any increase above a threshold equals to the summation of the average and x standard deviations. By setting x large enough, only real spikes are captured by the algorithm.

2.1.4 Limitations

With the exception of the index of dispersion based method, the measures and techniques described above are only suitable for online burst detection but not for quantifying workload burstiness. While the index of dispersion method can be used as a measure of burstiness, the choice of C is not intuitive. One possible modification to quantify burstiness using the above methods is to identify and count spikes in each workload over a certain period of time. The more bursty workloads will have larger number of spikes over the same period. The drawback of this technique is its inability to discover periodic bursts, e.g., more people search for the word "Cinema" on search engines during the weekends [31].

2.2 Quantifying burstiness

Many of the work on quantifying burstiness is based on the information theoretic entropy (H) [11] which is a measure of uncertainty in a workload X. Entropy is calculated using,

$$H(X) = -\sum_{i=1} p_i \times \log p_i, \tag{5}$$

where p_i is the probability of the workload having the value X_i to occur. The main limitations with using entropy are the need for a long sequence to be able to calculate the probabilities, its inability to differentiate between slowly increasing workloads and sudden spikes, and its inability to take into account burst periodicity.

2.2.1 Slope of entropy plots

Wang et al. [33] introduce entropy plots, plots showing the entropy of a workload at different aggregation levels for the workload. If entropy is calculated at a fine resolution, no correlation in the load is observed. The higher the resolution of aggregation is, the more correlations that can be captured by H. If the plot shows a linear relationship then the correlation and burstiness is stable across different workload resolutions. The slope can then be used as an indicator for workload burstiness. If there is a change in the workload burstiness and the plot is non-linear, then this method does not work [21].

2.2.2 Normalized entropy

Minh et al. [21] introduce the use of normalized entropy as a measure of workload burstiness. It is known that the maximum value achievable for H is $\log N$ [11]. Given a time interval T , Minh et al. divide the interval into N equal sized intervals. The normalized entropy is then defined as,

$$H_{NE}(X) = -\frac{\sum_{i=1}^{N} p_i \times \log p_i}{\log N}, \tag{6}$$

where p_i denotes the probability that a job arrives in interval i. H_{NE} is bounded between 0 and 1. According to Minh et al., H_{NE} values close to zero indicates stronger burstiness.

2.2.3 Burst density

Shen et. al. [28] propose to use signal processing techniques to measure the burst density in a workload. The authors use the Fast Fourier Transform (FFT) to calculate the coefficients that represent the amplitude of each frequency component for recent

resource usage, e.g., $L = \{l_{t-W_a}, ..., l_{t-1}\}$ where L is the time series of recent resource usage and W_a is the number of measurements considered. They consider the top k (e.g., 80%) frequencies in the frequency spectrum as high frequencies and apply inverse FFT over the high frequency components to synthesize the burst pattern. They calculate a burst density metric β as the percentage (or number) of positive values in the extracted burst pattern compared to the total signal. Higher percentages of β indicate strong burstiness and vice versa. One main disadvantage of this technique is the complexity in choosing k, the number of top frequencies that are considered as burst components. We discuss the limitations for both the normalized entropy and burst density techniques in Section 3.3 in more details.

2.2.4 The Hurst parameter

The Hurst parameter (exponent or coefficient) has been used as a measure for burstiness in the literature for numerous workloads [17, 20, 27]. The Hurst parameter, H, is a measure of the level of self-similarity of a time-series. Self-similar processes have heavy-tailed distributions, and thus, there burstiness can be observed at all time scales.

Various practical issues with estimating the Hurst exponent have been discussed in the literature [9]. In the literature, there are many suggested techniques to estimate the Hurst exponent differ for the same time-series to an extent making the estimation unreliable [22]. Just picking one algorithm from the literature to estimate the Hurst parameter and applying it to different workloads "is likely to end up with a misleading answer or possibly several different misleading answers" [9]. Using the Hurst parameter to quantify cloud workload burstiness can thus lead to wrong conclusions about the workloads running resulting in "bad" management decisions.

3 A new burstiness measure

Sample Entropy (*SampEn*) is a robust burstiness measure that was developed by Richman et al. over a decade ago [25, 26]. It is used to classify abnormal (bursty) physiological signals. It was developed as an improvement to another burstiness measure, Approximate Entropy, widely used previously to characterize physiological signals [24]. Sample Entropy is defined as *"the negative natural logarithm of the (empirical) conditional probability that sequences of length m similar point-wise within a tolerance r are also similar at the next point"*.

It has two advantages over Shannon's entropy: i) being able to operate on short data sequences and, ii) it takes into account gradual workload increases and periodic bursts. These advantages make it an interesting potential measure for workload burstiness as

a workload having periodic bursts, e.g., every weekend, is easier to manage compared to workloads with no repetitive bursts.

Three parameters are needed to calculate *SampEn* for a workload. The first parameter is the pattern length m, which is the size of the window in which the algorithm searches for repetitive bursty patterns. The second parameter is the deviation tolerance r which is the maximum increase in load between two consecutive time units before considering this increase as a burst. The last parameter is the length of the workload which can easily be computed. We therefore focus on m and r and their choice.

The deviation tolerance defines what is a normal increase and what is a burst. When choosing the deviation tolerance, the relative and absolute load variations should be taken in account, For example, a workload increase requiring 25 extra servers for a service having 1000 VMs running can probably be considered withing normal operating limits, while if that increase was for a service having only 3 servers running then this is a significant burst. Thus by carefully choosing an adaptive r, *SampEn* becomes normalized for all workloads. If *SampEn* is equal to 0 then the workload has no bursts. The higher the value for *SampEn*, the more bursty the workload is.

3.1 Implementation of the algorithm

There is one main limitation of *SampEn*, it is expensive to calculate both CPU-wise and memory-wise. The *computational complexity (in both time and memory) of SampEn is* $O(n^2)$ where n is the number of points in the trace [1]. [1] In addition, workload characteristics might change during operation, e.g., when Michael Jackson died, 15% of all requests directed to Wikipedia were to the article about him creating spikes in the load [6]. If *SampEn* is calculated for a long history, then recent changes are hidden by the history.

To address these two points, we modified the sample entropy algorithm [1] by dividing the trace into smaller equal sub-traces. *SampEn* is calculated for each sub-trace. A weighted average, *AvgSampEn*, is then calculated for all *SampEn* values for the sub-traces. More weight can be given to more recent *SampEn* values. This way the time required for computing *SampEn* is reduced since n is reduced significantly. Our modification also enables online characterization of workloads since *SampEn* is not recomputed for the whole workload history but rather for the near past. This approach of dividing the workload into smaller sub-traces is similar to the approach used by Costa et al. [10] where they divide a signal to calculate its multi-scale entropy. The main difference between the two approaches occurs after *SampEn* is computed for the smaller sub-trace, Costa et al. plot the results while we take a weighted average.

[1] There are some suggested implementations that significantly improve the complexity to be between $O(n)$ and $O(n^{3/2})$ [23] making *SampEn* an even more attractive measure.

Algorithm 3: The algorithm for calculating *AvgSampEn*.

Data: r, m, T, L
Result: *AvgSampEn*

1 $N \leftarrow Length(L)$;
2 $P_{divided} \leftarrow \{T(L.k).....T(L.(k+1)\}\ \forall k \in \{0,N\}$;
3 $Tot_{SampEn} \leftarrow 0$;
4 **for** W *in* $P_{divided}$ **do**
5 $\quad n \leftarrow Length(W)$;
6 $\quad B_i \leftarrow 0$;
7 $\quad A_i \leftarrow 0$;
8 $\quad X_m \leftarrow \{X_m(i)|X_m(i) = [x(i),...x(i+m-1)]\ \forall\ 1 < i < n-m+1\}$;
9 $\quad$ **for** $(X_m(i), X_m(j))$ *in* X_m*:* $i \neq j$ **do**
10 $\quad\quad$ Calculate $d[X_m(i), X_m(j)] = \max(|x(i+k) - x(j+k)|)\ \forall\ 0 \leq k < m$;
11 $\quad\quad$ **if** $d[X_m(i), X_m(j)] \leq r$ **then**
12 $\quad\quad\quad B_i \leftarrow B_i + 1$;
13 $\quad B^m(r) \leftarrow \frac{1}{n-m} \sum_{i=1}^{n-m} \frac{1}{n-m-1} B_i$;
14 $\quad m = m+1$
15 $\quad X_m \leftarrow \{X_m(i)|X_m(i) = [x(i),...x(i+m-1)]\ \forall\ 1 < i < n-m+1\}$;
16 $\quad$ **for** $(X_m(i), X_m(j))$ *in* X_m*:* $i \neq j$ **do**
17 $\quad\quad$ Calculate $d[X_m(i), X_m(j)] = \max(|x(i+k) - x(j+k)|)\ \forall\ 0 \leq k < m$;
18 $\quad\quad$ **if** $d[X_m(i), X_m(j)] \leq r$ **then**
19 $\quad\quad\quad A_i \leftarrow A_i + 1$;
20 $\quad A^m(r) \leftarrow \frac{1}{n-m} \sum_{i=1}^{n-m} \frac{1}{n-m-1} A_i$;
21 $\quad Tot_{SampEn} \leftarrow |-\log[\frac{A^m(r)}{B^m(r)}]| + a \times Tot_{SampEn}$;
22 $AvgSampEn \leftarrow Tot_{SampEn}/N$;

Our algorithm is shown in Algorithm 3. [2] T is the workload for which *SampEn* is calculated. The trace is divided into N sub-traces of length L (lines 1 to 3). For each sub-trace, W, *SampEn* is calculated. The first loop in the algorithm (lines 9 to 14) calculates $B^m(r)$, the estimate of the probability that two sequences in the workload having m measurements do not have bursts. The second loop in the algorithm (lines 15 to 19) calculates $A^m(r)$, the estimate of the probability that two sequences in the workload having $m+1$ measurements do not have bursts. Then *SampEn* for the sub-trace is calculated and is added to the sum of the *SampEn* values of all previous

[2] While there are implementations of *SampEn* in Matlab, we chose to implement our algorithm in Python.

sub-traces multiplied by a weighting factor a (line 20). The average *SampEn* for the whole trace is then calculated.

3.2 Choosing the parameters

SampEn has been shown relatively robust towards the choice of the two main parameters, the deviation tolerance, r, and the pattern length, m [8, 25]. Ideally, r should be chosen such as to capture all true bursts while m should be long enough to capture interesting periodic patterns. While choosing m, another contributing factor is the granularity of data, studying patterns occurring yearly is different from studying patterns occurring daily or hourly. Some studies suggest different techniques to set the two parameters in the context of biomedical signals [8, 16].

Choosing r, some studies suggest the use of some percentage of the standard deviation of the signal [24, 25]. Lake et. al. [16] show that this can lead to a reduction in the calculated *SampEn* due to the inflation of the standard deviation by the spikes. We have thus chosen a different approach to calculate r. Instead of using the standard deviation of the trace, r is set as a percentage of some high percentile of the workload, e.g., 50% of the 75^{th} percentile of the load values. Any increase above that value will be considered as a spike. Since percentiles in general are a *robust measure of scale*, the errors described by Lake et. al. are no longer an issue.

For choosing m, using Auto-Regressive models [16] or using the combination of the first minimum value of the nonlinear correlation function called average Mutual Information (MI) and the calculation of False Nearest Neighbor (FNN) [8] has been proposed. Cloud workloads are different from biomedical signals since most cloud workloads will have some inherent pattern, e.g., hourly, daily and weekly patterns. Therefore, m should be picked up to capture these patterns instead of using more complicated models or methods suitable for biomedical signals. The value of L should be chosen to be longer than m. Since L controls the rate at which *AvgSampEn*, it should be chosen to suit the frequency with which decisions are taken.

3.3 Comparison to the state-of-the-art

We calculated *AvgSampEn* for a set of more than 70 workloads (more than 10 real workloads and 60 synthetic traces). Due to space limitations, we only show *AvgSampEn* values for a subset of these loads, the workloads in Figure 2. The traces shown all have minutes time granularity. We set the pattern length m to one hour and the value of L to one day. The deviation tolerance r is set to 30% of the 70^{th} percentile of the load values during a period L. The weighting factor a is set to one.

Table 1 shows the *AvgSampEn* values for the 4 workloads Figure 2 compared to the use of Normalized entropy described in Section 2.2.2 and the burst density metric described in section 2.2.3. The burst density metric, β, was computed when different

Table 1: *Workload analysis results.*

Workload	*AvgSampEn*	H_{NE}	β_{80}	$\beta_{0.0001}$
Wikipedia	0.22	0.998	0.45	0.5
FIFA	0.48	0.988	0.27	0.6
SD-IRCache	3.3	0.975	0.25	0.23
Google	236.25	0.9	0.28	0.4

percentage of frequencies constitute the burst pattern. We choose to show only two values representing two extremes, when the top 80% and the top 0.0001% frequencies constitute the burst pattern.

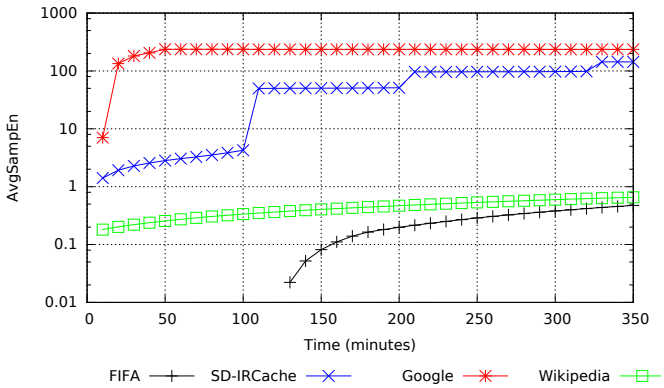
Since the Wikipedia workload shows very little burstiness and is repetitive, the value of *AvgSampEn* is very low, almost zero, indicating no burstiness. The SD-IRCache has more bursts where the load almost doubles in a very short period and thus the value of *AvgSampEn* increases and is almost one. The number of tasks submitted to the Google cluster is very bursty in nature, thus *AvgSampEn* is very high. Our experiments on the other workloads showed similar results for the values of *AvgSampEn* with different workloads. We therefore used *AvgSampEn* in our workload analyzer and classifier [4].

The values of H_{NE} given in Table 1 show that H_{NE} changes marginally with the different workloads. A workload with clear and strong burstiness such as the Google workload has an H_{NE} value almost equal to the Wikipedia workload which is very stable. On the other hand, *AvgSampEn* varies significantly with the workload burstiness. The increase in *AvgSampEn* with increasing burstiness is non-linear due to the logarithmic function. The burst density metric is clearly unable to differentiate the workloads in terms of burstiness. It is also not robust with changing parameters, i.e., the order of the workloads changes with changing the percentage of frequencies considered.

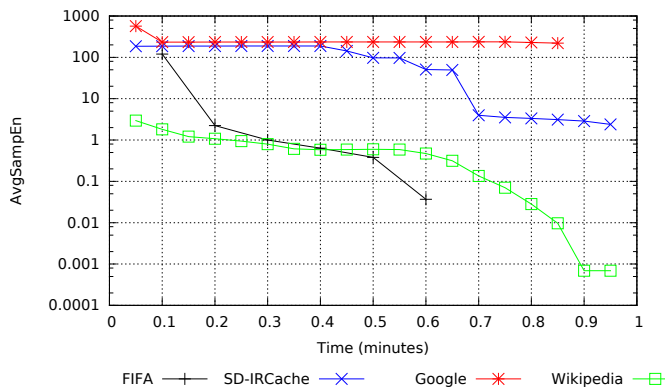
3.4 Sensitivity of *AvgSampEn*

To check the sensitivity of *AvgSampEn* while varying the two main parameters, r and m, we calculate the value of *AvgSampEn* when the value of m varies between ten minutes and six hours with steps of ten minutes, i.e., when m takes the value of 10, 20, 30, ..., 360 minutes. For each step for m we vary r between the 5^{th} percentile of all workload values multiplied by 0.05 to the 95^{th} percentile multiplied by 0.95 with an increase of 5 percent in the percentiles and 0.05 in the multiplicative factor.

To give an example, Figure 3(a) shows the ranking of the values of *AvgSampEn* when r is set to the 50^{th} percentile of the workload values, i.e., the median workload



(a) Sensitivity of the Workload when r is set to be the 50^{th} percentile of the workload and m varies between ten minutes to six hours.



(b) Sensitivity of the Workload when m is set to five hours and r varies between the 5^{th} percentile to the 95^{th} percentile of the load.

Figure 3: The ranking of the workloads is relatively stable for varying m but shows less stability when r varies and the two workloads have *AvgSampEn* values close to each other.

value, multiplied by 0.5. We multiply the percentile by a factor to make the differences between the tolerances chosen large and to be consistent with the way we defined r in 3.3. The pattern length, m, varies in the figure between ten minutes to 6 hours. It can be seen that the ranking of the workloads is robust with respect to the pattern length. On the other hand, this ranking differs with the ranking in Table 1. The difference in ranking is for the FIFA and Wikipedia workloads, the least two bursty workloads.

Figure 3(b) shows the ranking when m is set to 5 hours while r is left to vary as described above. While the ranking of the two most bursty workloads is stable, the least two bursty workloads ranking switch when r is set to the 40^{th} percentile of the workload values multiplied by 0.4. A low value for r, the deviation tolerance, results in less stable workload ranking since r defines the sensitivity of the measure to what a "true burst" is. It is thus advised to use a large enough r to make sure that only "true bursts" are taken in to account when estimating the burstiness of a workload.

4 Discussion

While *AvgSampEn* has been able to quantify burstiness in all the workloads we have tested it with, it is still far from perfect. The computational complexity of the algorithm is $O(n^2)$, even for the modified algorithm [23]. The time required to update *AvgSampEn* depends highly on L and m. In addition, the choice of the parameters, while intuitive, is arbitrary. Despite all the limitations, we were able to use *AvgSampEn* for workload classification and assignment to the most suitable elasticity algorithm with very high accuracy [4]. The sensitivity analysis showed that the measure is sensitive to the deviation tolerance chosen. To choose a "correct" deviation tolerance, one has to have a definition of a "true" burst. Defining bursts is still a problem that is far from being solved in the literature and it requires more focus within the community. We leave this for future work.

5 Acknowledgment

We thank Xiaohui Gu and Hiep Nguyen for providing us with their implementation of the burst density calculation algorithm. Financial support has been provided in part by the Swedish Government's strategic effort eSSENCE and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control.

References

[1] M. Aboy, D. Cuesta-Frau, D. Austin, and P. Micó-Tormos. Characterization of sample entropy in the context of biomedical signal analysis. In *IEEE EMBS*,

pages 5942–5945. IEEE, 2007.

[2] A. Ali-Eldin, A. Rezaie, A. Mehta, S. Razroev, S. Sjöstedt-de Luna, O. Seleznjev, J. Tordsson, and E. Elmroth. How will your workload look like in 6 years? Analyzing Wikimedia's workload. In *IC2E*. IEEE Computer Society, 2014.

[3] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *NOMS*, pages 204–212. IEEE, 2012.

[4] A. Ali-Eldin, J. Tordsson, E. Elmroth, and M. Kihl. Workload classification for efficient auto-scaling of cloud resources. Technical Report UMINF 13.13, Umeå University, Department of Computing Science, 2013. Available: www8.cs.umu.se/research/uminf/index.cgi?year=2013&number=13.

[5] I. Ari, B. Hong, E. L. Miller, S. A. Brandt, and D. D. Long. Managing flash crowds on the internet. In *MASCOTS*, pages 246–249. IEEE, 2003.

[6] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC*, pages 241–252. ACM, 2010.

[7] A. Caniff, L. Lu, N. Mi, L. Cherkasova, and E. Smirni. Fastrack for taming burstiness and saving power in multi-tiered systems. In *ITC*, pages 1–8, Sept 2010.

[8] X. Chen, I. Solomon, and K. Chon. Comparison of the use of approximate entropy and sample entropy: Applications to neural respiratory signal. In *IEEE-EMBS*, pages 4212–4215, Jan 2005.

[9] R. Clegg. A practical guide to measuring the hurst parameter. In *Proceedings of 21st UK Performance Engineering Workshop, School of Computing Science, Technical Repo*. N. Thomas, N. Thomas, 2005.

[10] M. D. Costa, C.-K. Peng, and A. L. Goldberger. Multiscale analysis of heart rate dynamics: Entropy and time irreversibility measures. *Cardiovascular Engineering*, 8(2):88–93, 2008.

[11] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[12] R. Gusella. Characterizing the variability of arrival processes with indexes of dispersion. *IEEE Journal on Selected Areas in Communications*, 9(2):203–211, 1991.

[13] M. Karsai, K. Kaski, A.-L. Barabási, and J. Kertész. Universal features of correlated bursty behaviour. *Scientific reports*, 2, 2012.

[14] M. Kihl, E. Elmroth, J. Tordsson, K.-E. Årzén, and A. Robertsson. The challenge of cloud control. In *8th International Workshop on Feedback Computing*, 2013.

[15] J. Kleinberg. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery*, 7(4):373–397, 2003.

[16] D. E. Lake, J. S. Richman, M. P. Griffin, and J. R. Moorman. Sample entropy analysis of neonatal heart rate variability. *American Journal of Physiology-Regulatory, Integrative and Comparative Physiology*, 283(3):R789–R797, 2002.

[17] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic. In *SIGCOMM*, volume 23, pages 183–193. ACM, 1993.

[18] N. S. C. A. Logs. Url: ftp://ircache. nlanr. net, 2000.

[19] M. Mathioudakis and N. Koudas. TwitterMonitor: Trend detection over the twitter stream. In *SIGMOD*, pages 1155–1158. ACM, 2010.

[20] D. Menascé, V. Almeida, R. Riedi, F. Ribeiro, R. Fonseca, and W. Meira Jr. In search of invariants for e-business workloads. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 56–65. ACM, 2000.

[21] T. N. Minh, L. Wolters, and D. Epema. A realistic integrated model of parallel system workloads. In *CCGrid*, pages 464–473. IEEE, 2010.

[22] A. Montanari, M. Taqqu, and V. Teverovsky. Estimating long-range dependence in the presence of periodicity: an empirical study. *Mathematical and computer modelling*, 29(10):217–228, 1999.

[23] Y.-H. Pan, Y.-H. Wang, S.-F. Liang, and K.-T. Lee. Fast computation of sample entropy and approximate entropy in biomedicine. *Computer methods and programs in biomedicine*, 104(3):382–396, 2011.

[24] S. M. Pincus. Approximate entropy as a measure of system complexity. *Proceedings of the National Academy of Sciences*, 88(6):2297–2301, 1991.

[25] J. S. Richman, D. E. Lake, and J. R. Moorman. Sample entropy. *Methods in enzymology*, 384:172–184, 2004.

[26] J. S. Richman and J. R. Moorman. Physiological time-series analysis using approximate entropy and sample entropy. *AJP-Heart and Circulatory Physiology*, 278(6):H2039–H2049, 2000.

[27] A. Riska and E. Riedel. Disk drive level workload characterization. In *ATC*, pages 97–102. Usenix, 2006.

[28] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *SoCC*, page 5. ACM, 2011.

[29] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *ICAC*, pages 21–30. ACM, 2010.

[30] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[31] M. Vlachos, C. Meek, Z. Vagena, and D. Gunopulos. Identifying similarities, periodicities and bursts for online search queries. In *SIGMOD*, pages 131–142. ACM, 2004.

[32] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *NOMS*, pages 96–103. IEEE, 2010.

[33] M. Wang, A. Ailamaki, and C. Faloutsos. Capturing the spatio-temporal behavior of real traffic data. *Performance Evaluation*, 49(1):147–163, 2002.

[34] Q. Wang, Y. Kanemasa, M. Kawaba, and C. Pu. When average is not average: large response time fluctuations in n-tier systems. In *ICAC*, pages 33–42. ACM, 2012.

[35] P. Wendell and M. J. Freedman. Going viral: Flash crowds in an open cdn. In *SIGCOMM*, IMC '11, pages 549–558, New York, NY, USA, 2011. ACM.

[36] J. Wilkes. More Google cluster data. Accessed: May, 2013.

Paper V

Analysis and Characterization of a Video-on-Demand Service Workload

A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth

Proceedings of the 6th ACM Multimedia Systems Conference, pages 189-200, ACM, 2015.

Analysis and Characterization of a Video-on-Demand Service Workload*

Ahmed Ali-Eldin†, Maria Kihl‡, Johan Tordsson†, and Erik Elmroth†

Abstract

Video-on-Demand (VoD) and video sharing services account for a large percentage of the total downstream Internet traffic. In order to provide a better understanding of the load on these services, we analyze and model a workload trace from a VoD service provided by a major Swedish TV broadcaster. The trace contains over half a million requests generated by more than 20000 unique users. Among other things, we study the request arrival rate, the inter-arrival time, the spikes in the workload, the video popularity distribution, the streaming bit-rate distribution and the video duration distribution. Our results show that the user and the session arrival rates for the TV4 workload does not follow a Poisson process. The arrival rate distribution is modeled using a lognormal distribution while the inter-arrival time distribution is modeled using a stretched exponential distribution. We observe the "impatient user" behavior where users abandon streaming sessions after minutes or even seconds of starting them. Both very popular videos and non-popular videos are particularly affected by impatient users. We investigate if this behavior is an invariant for VoD workloads.

1 Introduction

Over the past decade, Video on Demand (VoD) and Video sharing online services have been on the rise. A recent report estimated that more than 50% of the total downstream traffic during peak periods in North America originate from

*The paper has been re-typeset to match the thesis style. Produced with permission of ACM.

†Department of Computing Science, Umeå University, Sweden, email: {ahmeda, tordsson, elmroth}@cs.umu.se

‡Department of Electrical and Information Technology, Lund University, Sweden, email: {Maria.Kihl}@eit.lth.se

Netflix and YouTube [15]. It is thus required to analyze and characterize VoD workloads in order to understand how to improve and optimize the network usage and the perceived Quality-of-Service (QoS) by the service users.

Many VoD service providers utilize the power of cloud computing to host their services [3, 20, 35]. Since a typical cloud hosts multitudes of applications with differing workload profiles [1], Cloud service providers need to understand the workload characteristics of the running applications including the VoD workload dynamics. This understanding is crucial as application co-hosting can result in performance interference between collocated workloads [9, 37]. Furthermore, resource management problems such as service admission control [45], Virtual Machine (VM) placement [10], VM migration [44] and elasticity [38] can be further complicated depending on the workload characteristics [30]. To better understand VoD workloads, we obtained recent workload traces from TV4, a major Swedish VoD service provider, detailing the requests issued by the premium service subscribers to TV4's VoD service. The VoD service is hosted on a number of cloud platforms. We provide an extensive analysis and characterization study of the traces.

Table 1: *Example of one entry in the trace.*

Video title	Farmen del 1_2255111
viewer ID (hashed)	a257d2e7788db3238f
Streaming start time	2013-01-13 17:00:00
Streaming end time	2013-01-13 17:08:00
Number of minutes viewed	8
average Bitrate (Mbps)	0.8
categories	None
category Tree	Nöje/Farmen
video Category	Farmen

Table 1 shows an example entry in the trace. Each entry contains nine fields, out of which the following seven are used in the analysis, the title of the video requests, the hashed ID of the premium user who requested the video, the time the streaming of the video started and ended, the number of minutes the video was streamed, and the average bitrate of the stream. The remaining two are not used in the analysis since they are missing for some videos.

The traces contain logged data between the 31^{st} of December 2012 until the 18^{th} of March, 2013 from two cities. [1] City A is one of the largest cities in the Nordic countries with a population over half a million inhabitants while

[1]The non-disclosure agreement prohibits us from revealing the city names

City B is a much smaller city with less than fifty thousand inhabitants. The number of premium users who used the service during that period is 23102 users. The users viewed 17131 unique videos and started 532421 streaming sessions. After analyzing the workloads separately, and combined, we found that the results from our analysis is the same for all three cases with respect to workload dynamics and characteristics. The difference is in the scale, e.g., spikes occur at the same times but with higher magnitudes for the combined workload. Due to space limitations, we show only the analysis of the combined workload.

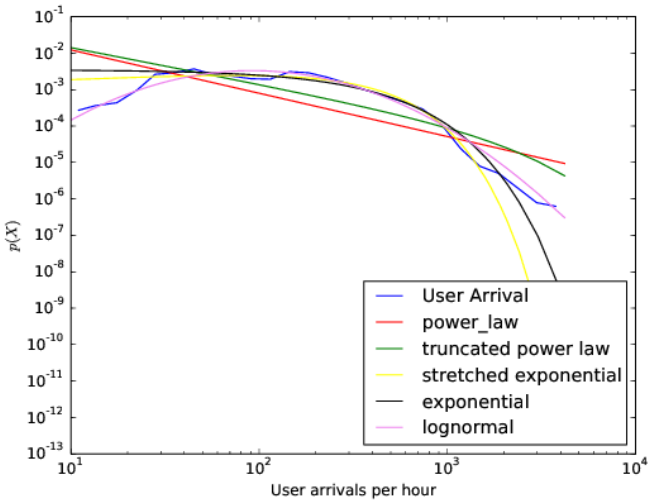
There are two main contributions of this work. First, in Section 2, we provide an extensive study of the workload properties related to the session arrivals. We show that the session arrivals and the user arrivals for the TV4 traces can not be modeled using a Poisson process, contrary to what has been assumed [25] or reported [48] in some prior work in the literature. We acknowledge that the limitation of this finding since our dataset is from a medium sized VoD service provider. We identify the main events that resulted in spikes in the number of arrivals. Since the spikes cause non-stationarity in the workload, the Hilbert-Huang transform is used to perform spectral analysis on the workload.

Second, Section 3 contains analysis for the properties of the workload related to the streamed sessions. Our analysis shows that a large percentage of the sessions started are terminated within the first few minutes before the video ends. Based on this behavior and the popularity distribution of the videos, we suggest a new caching strategy to help reduce the wasted resources by the VoD service provider. We review the state-of-the-art in Section 4 and conclude in Section 5.

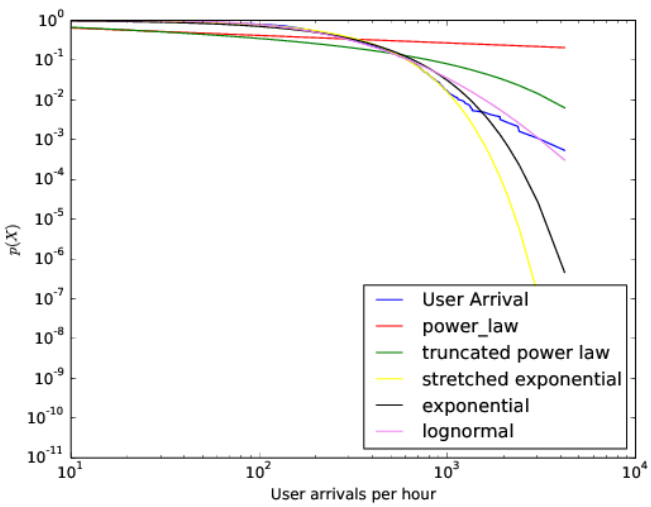
2 Workload Analysis: Arrivals

2.1 Arrival rate

The Probability Distribution Function (PDF) and the complementary Cumulative distribution Function (CCDF) of the hourly session arrival rate is shown in Figure 1 (in blue) on a Log-Log plot. An almost identical plot was also obtained for the user arrival rate since one user almost always does not start more than one session per hour. The PDF suggests that the arrival rate process can be modeled using a heavy tailed distribution. we have fitted the arrival rate data to different distributions and compared the goodness of fits in order to find a good fit. The data was fitted to lognormal, exponential, truncated power law, stretched exponential, gamma and power law distributions, as shown in Figure 1.



(a) PDF of the distribution of the arrival rate and different fits.



(b) CCDF of the distribution of the arrival rate and different fits.

Figure 1: The hourly request arrival rate distribution can be modeled as a lognormal distribution as suggested by the Log-Log plots of the PDFs and CCDFs of the different fits.

The plots show that either a lognormal distribution, an exponential distribution or a stretched exponential distribution is a good fit. To choose the best fit, we used the Kolmogorov-Smirnov (KS) test [13]. The p-value for both the lognormal distribution and the stretched-exponential distribution [34] was greater than 0.05, the least significance level required to validate the null hypothesis that the empirical data does not follow the distribution. To be precise, the KS distance for the lognormal distribution is 0.077 with a p-value of 0.09, and the KS distance for the stretched exponential distribution is 0.059 with a p-value of 0.31.

Both the lognormal and the stretched exponential distributions are possible fits given the p-values of the KS test. To identify the better fit, we use the log-likelihood ratio between the distributions [13]. The log-likelihood ratio of the lognormal distribution was higher with a p-value of 0.01. We thus conclude that the lognormal distribution is the best distribution to fit our data from the distributions tested. The fitted lognormal distribution is different from the arrival rate distribution of the VoD service provided by China Telecom discussed by Yu et al. [48] where the arrival rates follows a modified Poisson distribution.

2.2 Inter-Arrival time

Figure 2 shows the PDF of the video sessions' inter-arrival times (seconds) on a log-log scale. More than 50% of the sessions start after one or less than one second from the arrival of the previous session and around 90% of the sessions start within a minute from a previous session. The maximum inter-arrival time is around 24 minutes. We have again tried fitting a distribution to the Inter-Arrival time following the same steps described above. Again, the KS test showed that both the lognormal distribution (p-value=$0.08 > 0.05$) and the stretched exponential distribution (p-value=$0.08 > 0.05$) to be two viable fits. Testing using the log-likelihood method described by Clauset et al. [13], the stretched-exponential distribution has a higher likelihood than the lognormal distribution with a p-value=0.

While it is popular to model session and user arrival rates as Poisson processes in workload generators [25], our results suggest that for different VoD services, different models of arrival might occur. Poisson processes require the inter-arrival time distribution to be exponential. Figure 2 shows also the best exponential distribution fit we could achieve for the inter-arrival data. the deviation clearly shows that the inter-arrival time distribution is not exponential, and thus the arrivals do not follow a Poisson process. Poisson processes were considered the defacto processes to model network arrivals until the seminal work by Paxson and Floyd [39]. It is thus worth investigating if Poisson processes fail also to

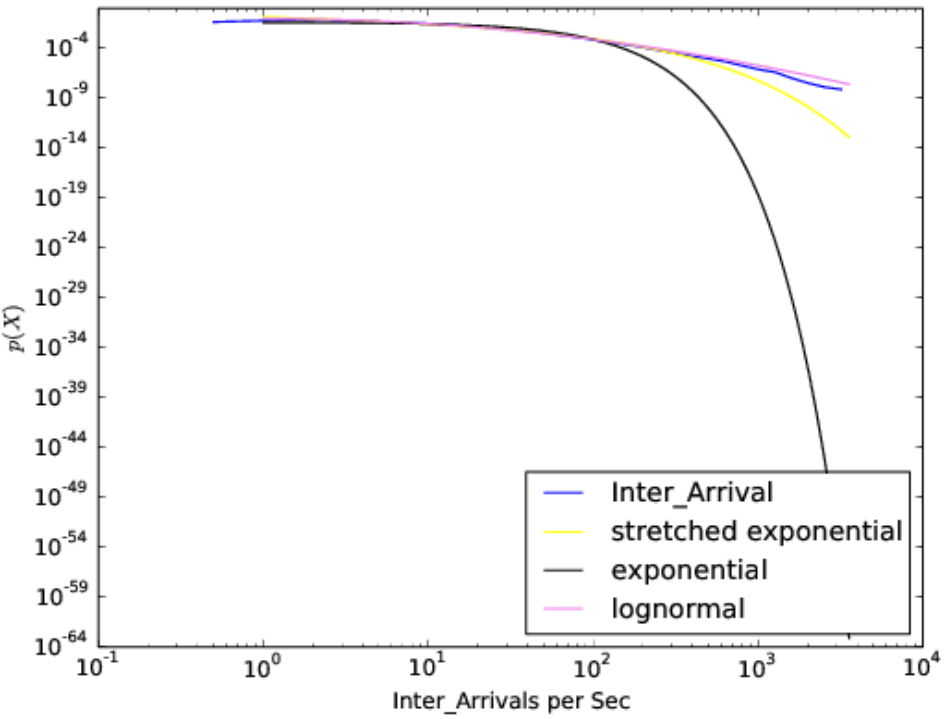


Figure 2: PDF of inter-arrival time (Log-Log plot).

model arrival processes for VoD systems. We unfortunately do not have sufficient data from enough VoD providers to come to such a conclusion.

Since at least for the TV4 workload, the user and request arrival processes can not be modeled using Poisson processes, many of the previously developed theories and models based on the assumption of requests/users generated from a Poisson process will either be inaccurate or will be wrong for systems like TV4 [18,21,25]. Since VoD workloads are scarce, we can not compare our results with systems other than the very few available in the literature.

2.3 Workload Spikes

Figure 3 shows the number of video sessions started per hour in the trace. The workload has a very clear daily pattern and a weaker weekly pattern. The pattern is violated due to four main significant spikes. The most significant spike occurred on Sunday, the 24^{th} of February, 2013 when the load increased from 687 video sessions at 18:00, to 4670 video sessions at 20:00. We investigated

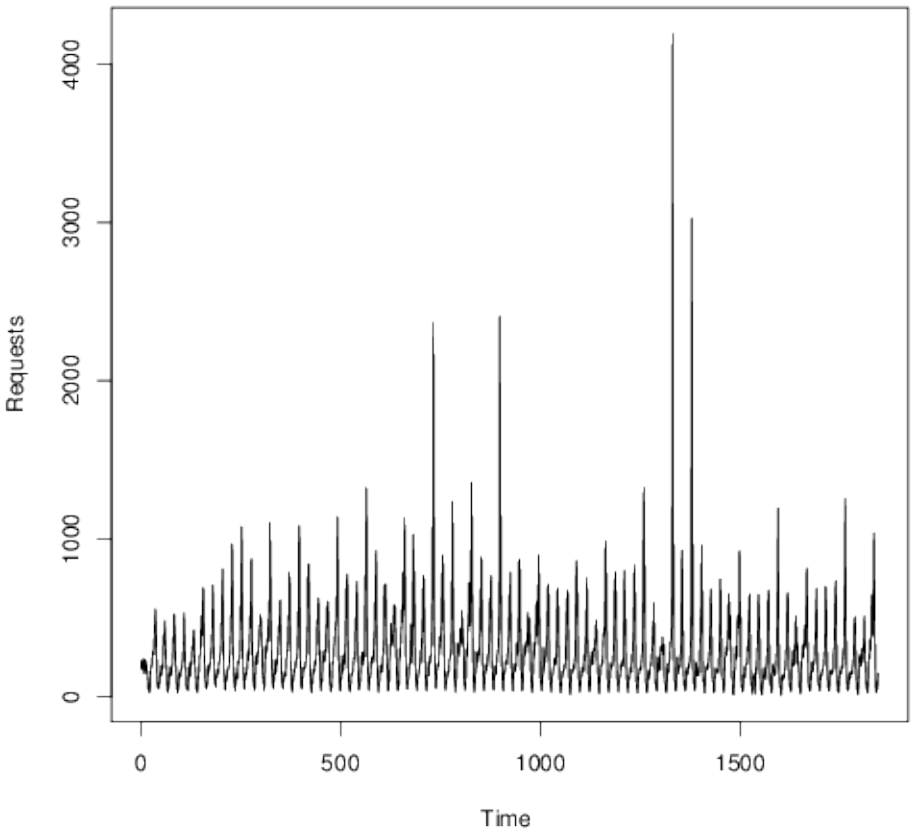


Figure 3: Number of video sessions per hour starting from 09:00 on the 31^{st} of December, 2012, to 09:00 on the 18^{th} of March, 2013. Some sessions run for a few seconds while others run for hours.

the main cause of this spike. To our surprise, the most viewed video was a live stream of a football game between two French teams in the French soccer league, Paris Saint-Germain and Olympique Marseille. Paris Saint-Germain is the team where, Zlatan Ibrahimovic, one of Sweden's favorite soccer players play [43]. The spike caused a workload increase by roughly four to six folds from normal behavior seen in the previous weeks.

The second most significant spike occurred two days later on the 26^{th} of February, when the load increased from 582 sessions at 19:00 to 3025 session at 21:00. Again, the main cause of the spike was a semi-final match in the Spanish cup between Barcelona and Real-Madrid. The third largest peak occurred on the 6^{th} of February, 2013 at 20:00 when an international football match was played between the Swedish and the Argentinian national teams. The load increased to 2405 sessions in that hour. The fourth largest peak occurred on the 30^{th} of February, 2013 at 21:00 when the load reached 2365 sessions. Again, the main cause of the spike was a football match in the Spanish cup between Barcelona and Real Madrid, the first leg prior to the match that caused the second largest spike.

Three of the four major spikes in the workload are generated by events that are not directly related to Sweden. This is a clear example showing the complexity of workload spike and burstiness management. A service provider should be able to cope with such events with no reduction in the QoS perceived by the service customers. The easy but expensive way to provide high QoS guarantees in the presence of spikes, is to over-provision, buy or rent enough server resources to handle the largest future spike well in advance before such a spike occurs. Another solution would be to utilize the power of cloud computing where new resources can be provisioned whenever needed and released when not used anymore [4]. The problem with the second approach is the difficulty of detecting spikes as they occur to be able to provision resources with no QoS degradation.

2.4 Daily patterns

Figure 4 shows a box and whiskers plot showing the effect of the time of the day on the number of video sessions. A box-plot is a way to visualize the quartiles and the dispersion of the distributions of data [36]. A box is plotted for all 24 hours of the day where the lower edge of the box represents the first quartile of the number of sessions arriving on any particular hour. The third quartile is represented by the top edge of the box. From figure 4, the diurnal pattern of the trace is evident. the hour with the least arrivals is at 05:00 every morning with almost no variability. The hours with the highest arrival rates are at 20:00, 21:00 and 22:00 at night, both of which also show significant variance and deviations from the normal behavior. Since there is very little variability in the load between mid-night and 15:00, and, the arrivals at these hours are low, a VoD service provider can use the available unused resources for business analytics [33] or can release them to save costs. Before hours with higher variability, the provider can provision more resources.

Similar patterns can be seen in other workloads, e.g., the load on Wikipedia [5], where the load decreases significantly between mid-night and noon. A cloud

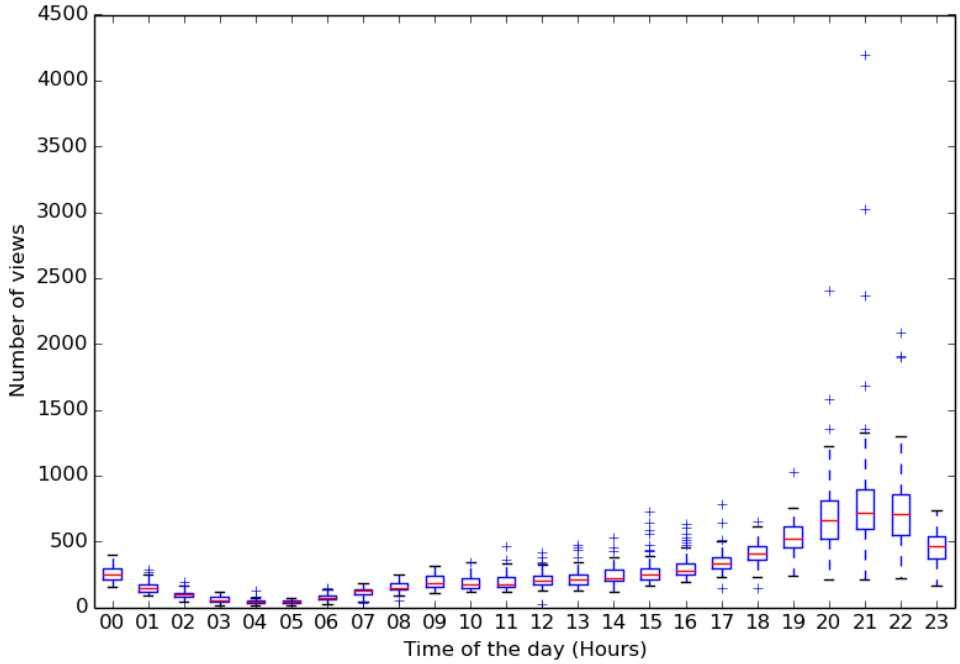
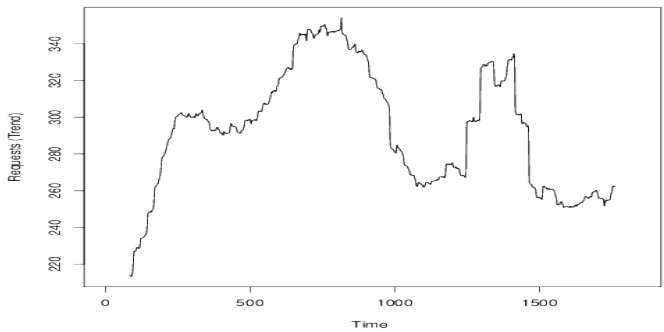


Figure 4: A Box and Whiskers graph showing the effect of the time of the day on the number of sessions.

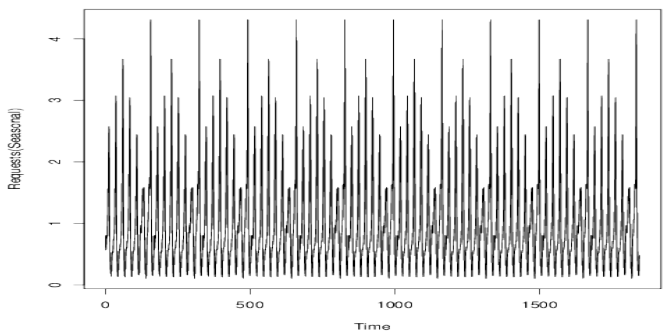
service provider can thus benefit from having services from different time-zones running in the datacenter. The multiplexing between the different services from different time-zones should provide higher revenues with a much lower risk of service performance interference.

2.5 Frequency representation

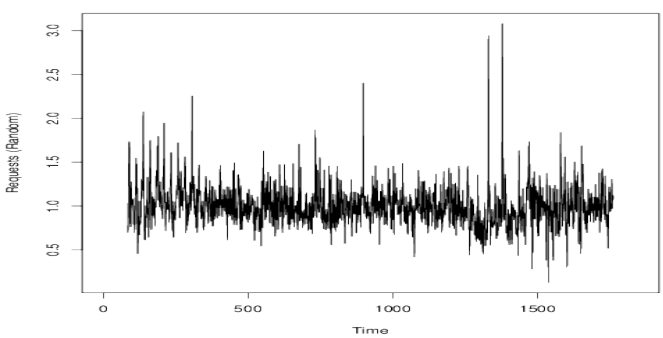
The request arrival rate represents a time-series. Any time series, X, can be decomposed into three components, the trend, the seasonality and the random components [28]. The trend, T is a slowly changing component which captures the change in the mean of the time series with time. The seasonality, S, represents the periodic components in the load. The random component, r is the remaining signal. The decomposition can be performed such that $X = T + S + r$ or such that $X = T \times S \times r$ [28]. Figure 5 illustrates the decomposition of the arrivals time-series, in Figure 3, into multiplicative factors. The trend component is a DC component with no frequency variations. The weak weekly patterns are



(a) The trend component T.



(b) The seasonality component S with clear diurnal and weekly patterns.



(c) The random component r.

Figure 5: The result from using multiplicative decomposition of the workload.

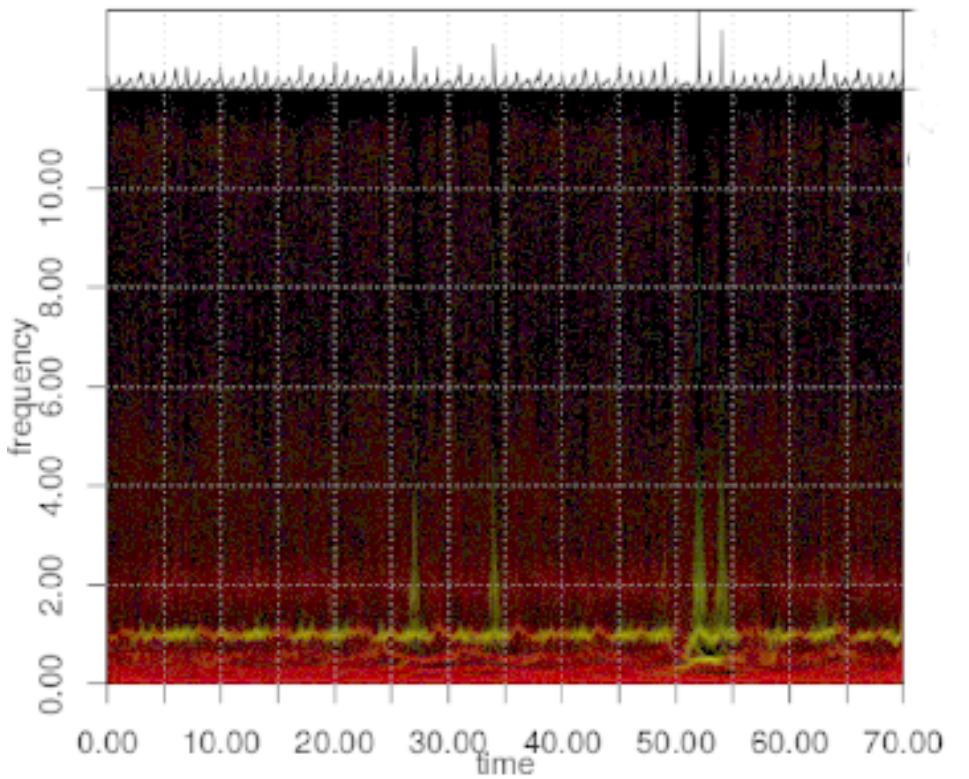


Figure 6: The Hilbert spectrum for the session arrivals time-series.

clear in Figure 5(b) since the decomposition removes any part of the signal that smears the periodic pattern.

Time-series can be classified into either stationary or non-stationary time-series. A stationary time-series has a non-changing mean and variance. A non-stationary time-series has one or both of mean and variance changing. To use traditional time-series analysis models such as ARIMA models, the studied time-series needs to be stationary [14]. The request arrivals time series for the TV4 data is non-stationary due to the presence of large spikes. In their seminal work to model non-stationary time-series, Huang et al. introduced a novel empirical method to characterize the frequency variations in non linear and non-stationary time-series [23], recently, known as the Hilbert-Huang Transform (HHT) [22].

At the the core of the HHT is the *Empirical Mode decomposition (EMD)* method and its different variations [17, 23, 46]. The EMD (and all its other variations) are methods with which any complicated data set can be decomposed

into a finite and often small number of *Intrinsic Mode Functions (IMF)* that admit well-behaved Hilbert transforms. The Hilbert spectrum can then be used to visualize the produced IMFs and frequency variations in the original signal. Since the number of IMFs produced is low, it is a more efficient way of spectral analysis compared to, for example, the Fourier transform which typically requires an infinite number of sinusoidal frequencies to represent any time-series. Huang et al. and others have discussed the strengths and weaknesses of their proposed method and showed the superiority of the HHT compared to other available spectral analysis methods such as the wavelet transforms and Fourier transforms [17, 23, 42, 46]

Performing spectral analysis on the signal as is with the trend distorts the spectrum. We have thus used the HHT method to perform spectral analysis on $X = S \times r$. The Hilbert spectrum is shown in Figure 6. The X-axis is the time in days and the Y-axis is the frequency in weeks. The colors represent the intensity of the frequency component at any point in time. On top of the graph, the analyzed time-series is plotted. The low frequency components dominate the time-series. The strongest of these components is the weekly component.

At the times of the four major spikes discussed previously, between 25 and 35 days, and 50 and 55 days in Figure 6, the spectral pattern is distorted. The spikes cause an increase in the power and dispersion of the spectrum of the time-series. This suggests that a possible way to detect spikes as they occur would be to use spectral analysis methods to detect the beginning of the spike [6]. We leave this for future work.

3 Workload Analysis: Video Sessions

3.1 Video popularity

Videos offered by a VoD service provider differ in popularity among the users. Figure 7 shows the CCDF of the popularity distribution of all videos viewed in our trace. The Figure suggests a heavy-tailed distribution would be good. We have fitted the popularity data to different distributions in a way similar to the way the arrival rate was fitted. Using the KS test, all the tried fits had a very low p-value and therefore were bad. Popularity in often modeled with a power law or a Zipfian distribution [2, 25, 48].

3.2 Video duration

Videos provided by the VoD service are of variable length. Not all users who start viewing a video stream continue to watch until the end. Some users keep

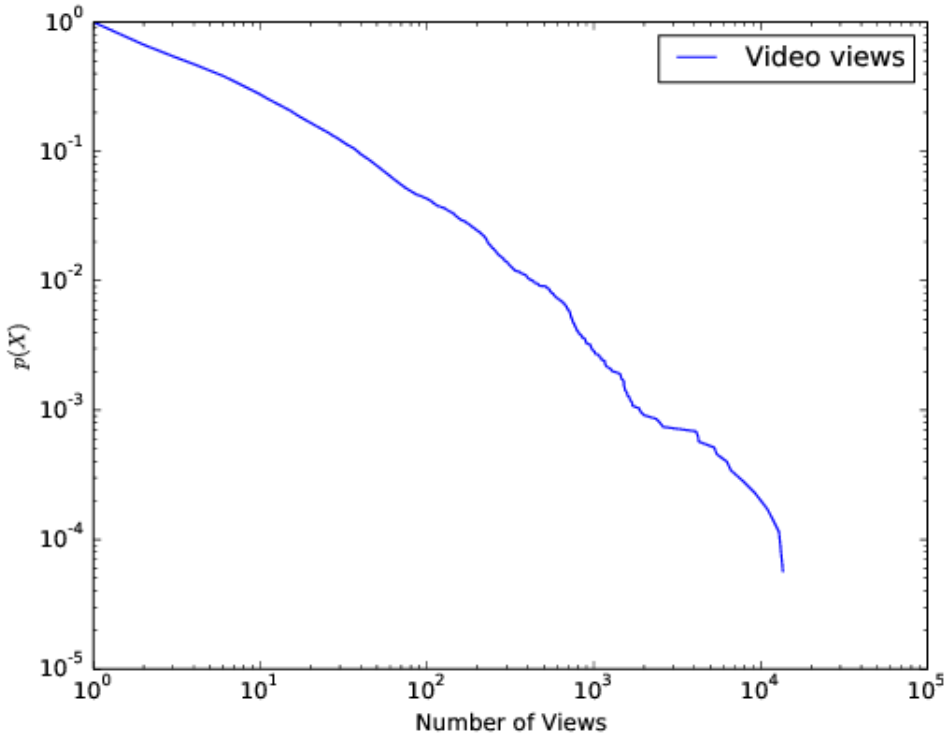


Figure 7: The empirical CCDF of the popularity of videos(Log-Log plots). We were not able to find a suitable fit for the data we have.

replaying the video, going back and forth in the video and pausing the video. This leads to sessions having very different length. Figure 8 shows the complementary CDF (CCDF) for the length of the VoD sessions (in seconds). More than 90% of the sessions last for less than one hour, with more than 50% of the total sessions lasting less than 12 minutes. More than 20% of the sessions gets terminated within the first 30 seconds from their start time. While we were not provided with the length of each single video hosted by TV4, we know that the median length of the videos is longer than 12 minutes. The videos are the ones viewed by the premium service users where most of the shows average anything from 20 minutes to 2 hours.

These numbers confirm the "impatient user behavior" discussed in previous studies described by Yu et al. [48]. Although the difference between our study and Yu et al.'s study is around 6 years, the numbers we find here do not differ considerably from their study. For example, Yu et al. found that 50% of the

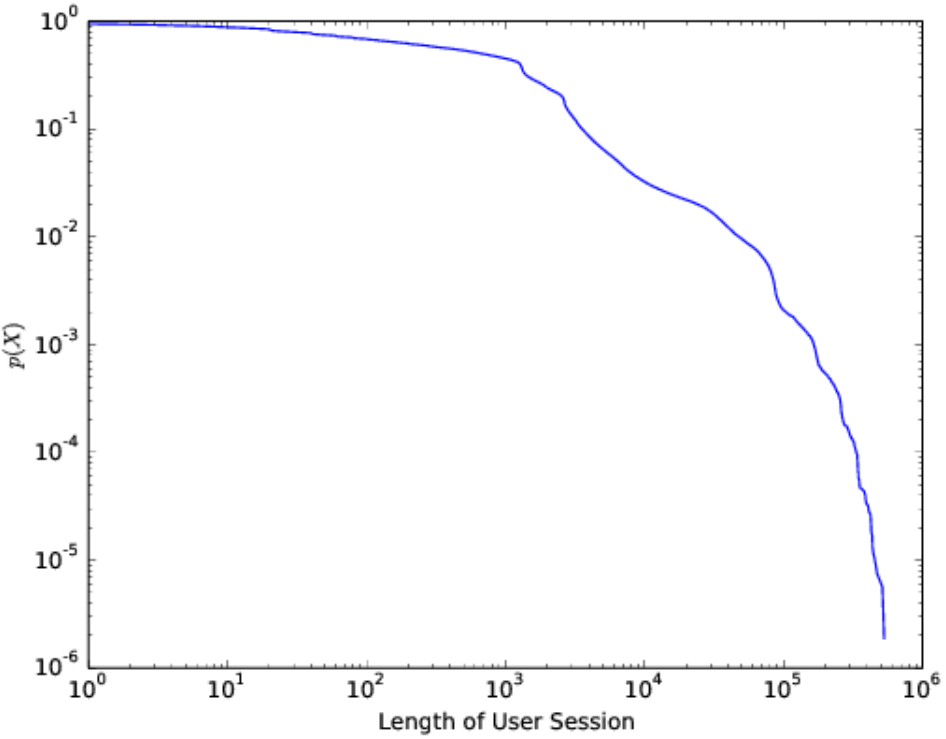


Figure 8: CCDF of the length of the sessions in seconds (Log-Log plot).

users terminate a session within the first ten minutes from when they start it and that more than 90% of all sessions terminate within 60 minutes from when they start. The main difference between our study and Yu et al.'s study in this respect is that the users of the TV4 VoD are more likely to stay than the users in Yu et al.'s study if they make it past the first 10 minutes.

One session lasted for 528771 seconds, which is equivalent to over 6 days. During the session, only one video has been streamed. The length of the video on the VoD service servers is around one hour. The average bit-rate for this stream was zero Mbps. We suspect that this is a session started by a user, paused and then forgotten about for six days with the receiving device on during that whole period. Another possibility is a fault in the receiver device resulting in no proper termination for the session.

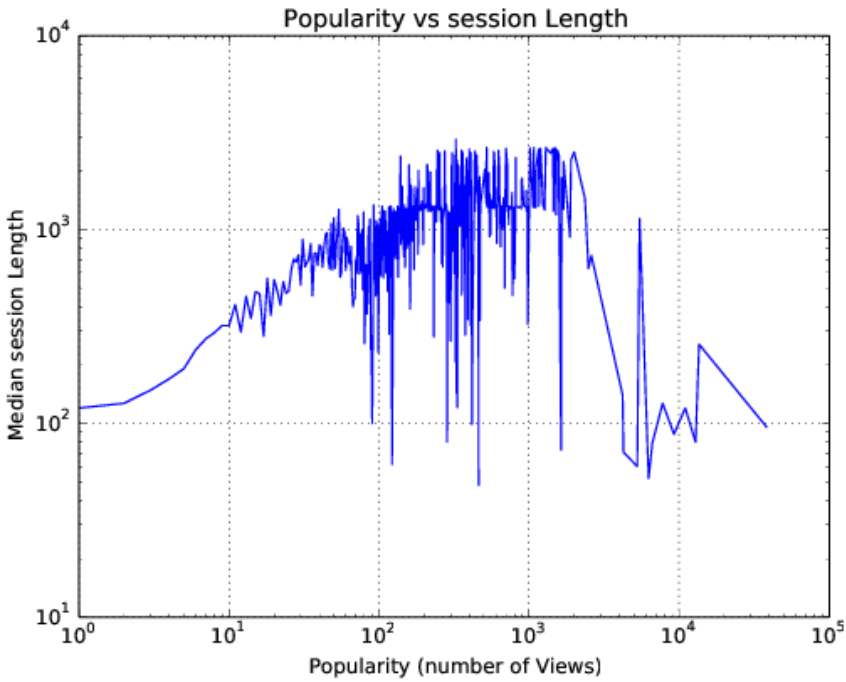


Figure 9: Video popularity versus median session length (Log-Log plot).

3.3 Caching

To handle the "impatient crowd", Yu et al. suggest to cache the first 10 minutes of videos to handle the load from up to 50% of the viewers. Since the popularity of the available videos are not the same, it would be a waste of resources to cache the first 10 minutes of unpopular videos or videos which get abandoned by most users before 10 minutes. Figure 9 shows how the median session length changes with the popularity of the videos (number of views). For extremely unpopular and extremely popular videos, the "impatient user behavior" is quite high with the median session length of around 100 seconds. Videos with a medium popularity seem to have longer session times. An advanced caching and prefetching policy [29] should utilize this difference to be able to improve the QoS while reducing wasted resources, e.g., by caching the first 16 to 20 minutes for videos with medium popularity, streaming the first 3 minutes for videos with low popularity and caching and prefetching the first 10 minutes for videos with high popularity. This caching strategy is suitable when users do not seek

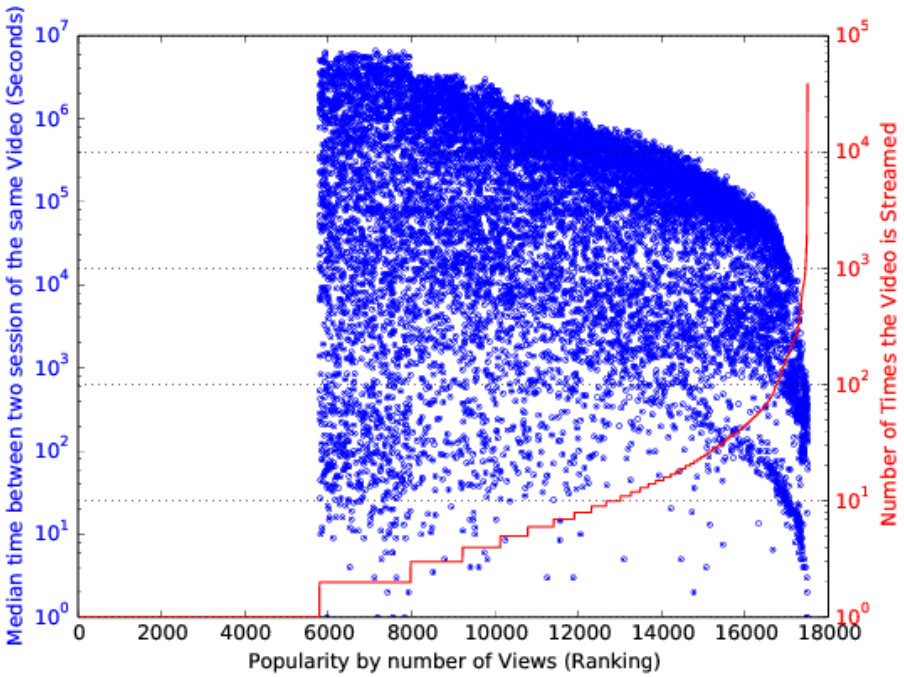


Figure 10: Video popularity versus the median time between two streams of the video.

forward and backward while watching a video to get to a more interesting part of the video [26].

In order to further understand better how to improve the caching strategy, we plot Figure 10. The X-axis of the figure represents the rank of the videos based on the number of times it has been streamed starting from least popular (with rank 1) to the most popular (with rank 17506). We then plot the median time between the arrivals of two consecutive streaming sessions for the same video (the blue dots). We also plot the total number of times the video has been streamed (the red line). The least popular 5791 videos were streamed only one time during the whole period. It is therefore useless to cache these videos since they are seldom streamed.

Looking at Figure 9, the median time for a session for these videos is less than 200 seconds. As the video popularity increases, the median time between two streaming sessions decreases considerably. Video popularity is volatile. Many of the higher ranked videos are streamed many times for a short period and never

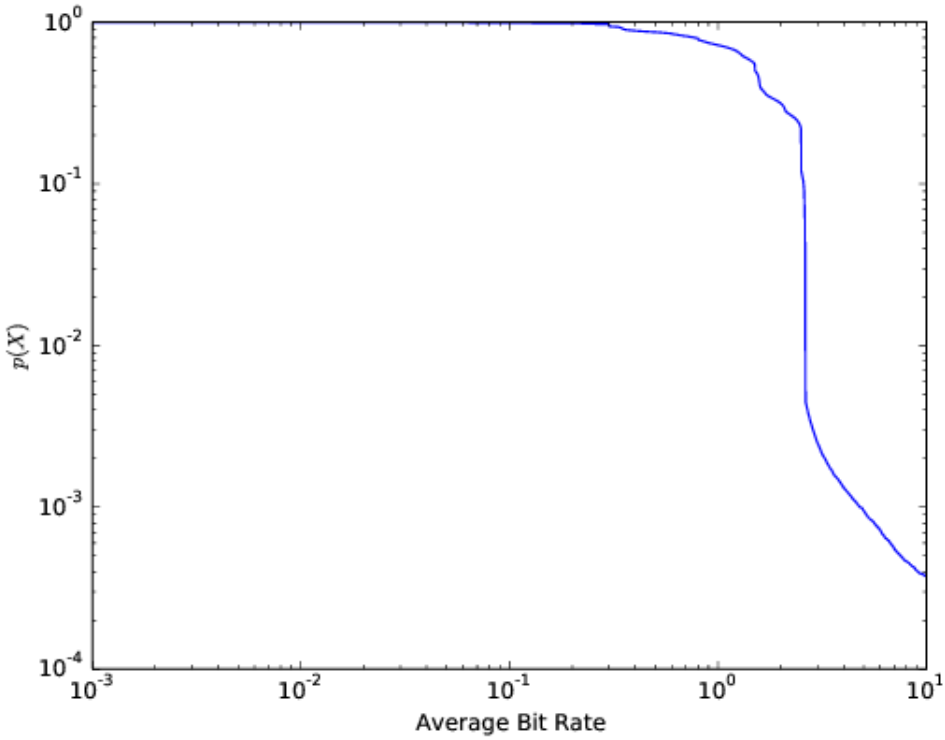


Figure 11: CCDF of the streaming bit-rate in Mbps (Log-Log plot).

streamed again, e.g., football matches are mostly streamed live in the dataset. Their popularity decay right after the match.

3.4 Bit-rates

Figure 11 shows the CCDF of the distribution of the average bit-rate transfer speed in Mbps. More than 90% of the VoD service users were streaming at a bit rate equal to or greater than one Mbps. More than 99% of the service users were streaming at a bit rate less than 3 Mbps. The bit-rate is highly correlated with the length of the session. Extremely low average bit-rate measurements should be correlated to the session length in order not to draw the wrong conclusions about the network performance.

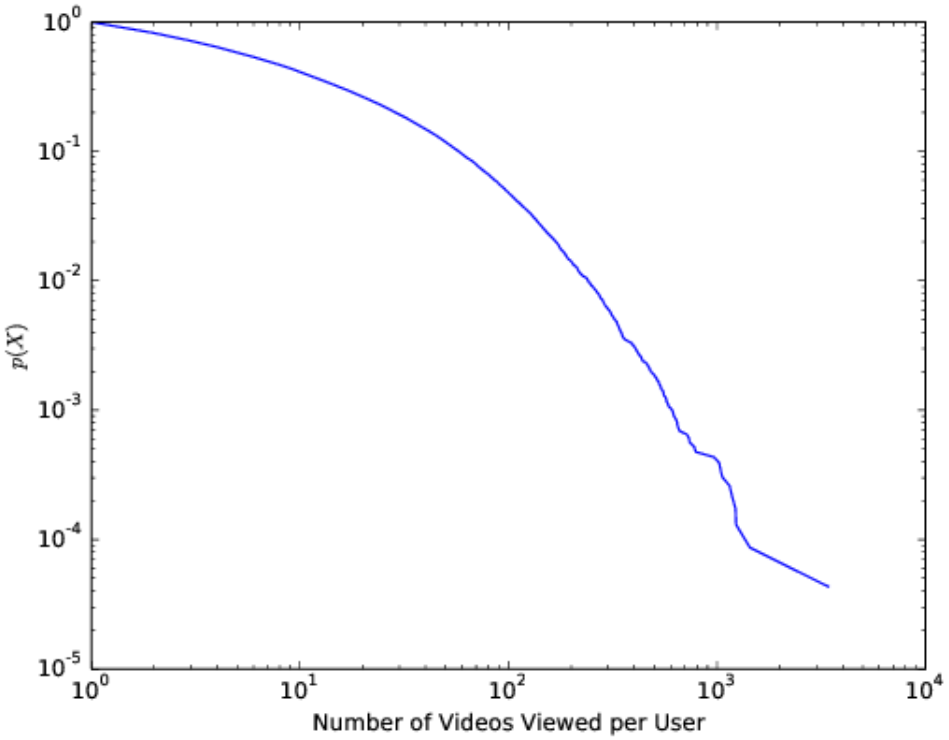


Figure 12: CCDF of the distribution of the number of videos viewed per user (Log-Log plot).

3.5 Video views per user

Figure 12 shows the CCDF of distribution of the number of videos viewed per user. More than 90% of the service users view less than 70 videos during the period of the study, i.e., less than one video per day. Many of these sessions last for less than 10 minutes. To see how long a user uses the VoD service, Figure 13 shows the CCDF of the total time a user used the VoD service. Some users have used the service for just a few minutes, with more than 25% of the users using the service for 45 minutes or less during the span of the three months which the workload covers. Other users have used the service heavily. The longest usage was by a customer who used the service for a total of 45 days and a few hours. This can be either a user who has the service running for over 15 hours per day, like a restaurant using the service, or a customer who has multiple devices all connected to the service using the same ID.

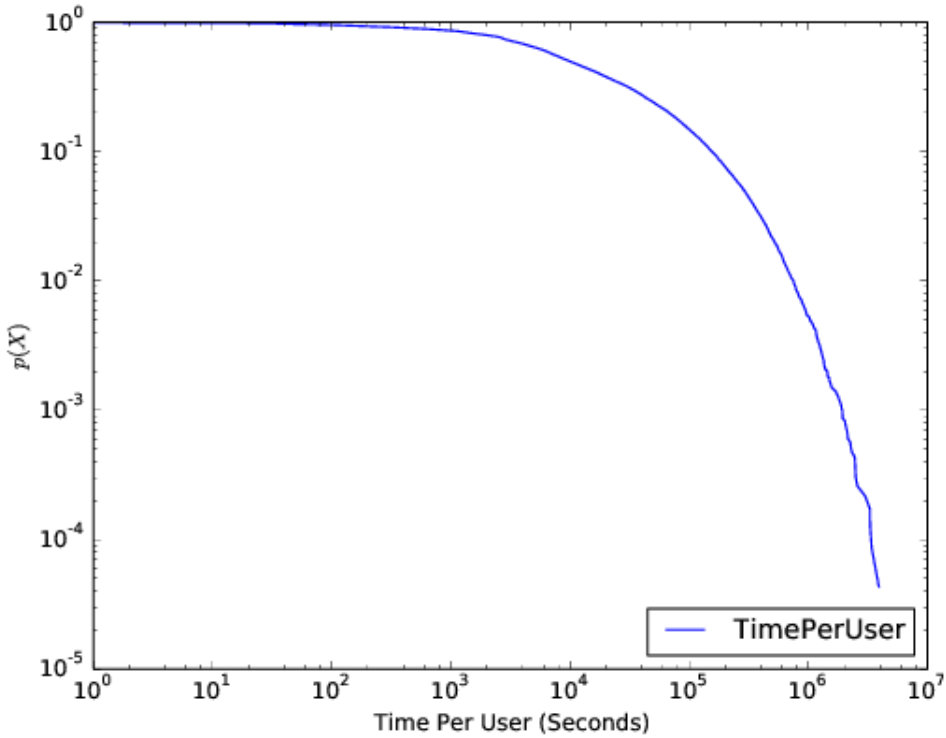


Figure 13: CCDF of the distribution of the number of seconds viewed per user (Log-Log plot).

3.6 Impatient users

To better understand why users abandon streams early, we investigated two hypotheses. The first hypothesis was that users abandon sessions due to low quality of the streaming, i.e., low bit-rate. Figure 14 shows how the average median session length of all users changes with the average streaming bit-rate. The figure shows that across most of the seen average bit-rates, the behavior of impatient users does not change. From Figure 11, average bit-rates more than 3 Mbps are rare, and thus the variation seen when the bit-rates are more than 3 Mbps in Figure 14 should not be interpreted as a change in the user-behavior but rather as outliers.

The second hypothesis was that users who use the service more will have a different average median session length. Figure 15 shows that the session length does not differ between users who use the service very often from those who do

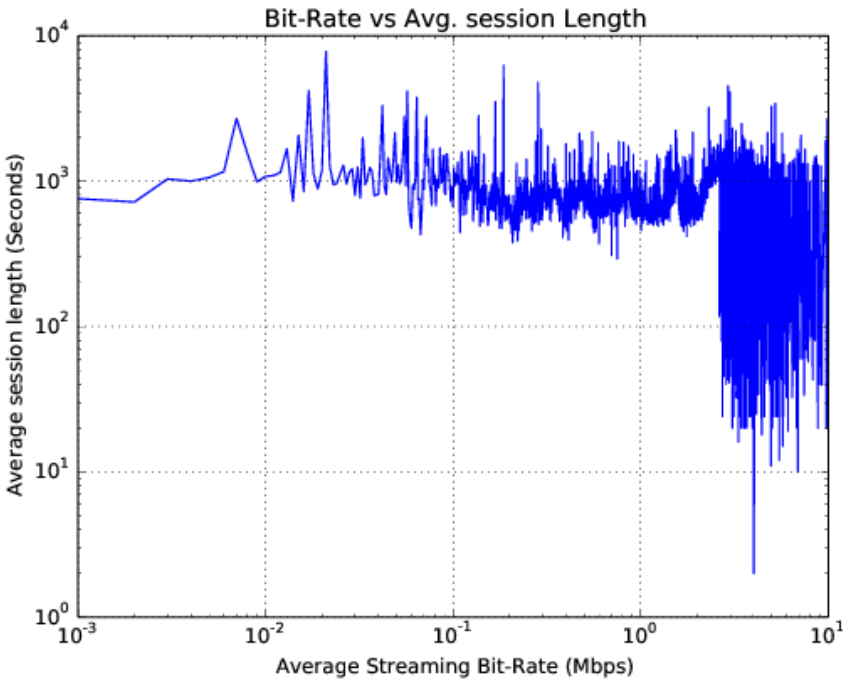


Figure 14: The bit-rate does not correlate with the decision of a user to abandon a session early (Log-Log plot).

not. Thus, both hypotheses are not true. The session length distribution is an invariant in the system.

4 Related work

Several server workloads for different services have been analyzed in depth previously [7,8,24,41,47]. Many of these studies focused on online video services and video streaming. One of the first and largest studies was conducted by Yu et al. [48] on a VoD system deployed by China Telecom, covering a total of 1.5 million unique users for a period of seven months in 2004. They focus their analysis on logs from a single representative city with a total user base of 150,000 users. They study the user arrival rates, session lengths and video popularity dynamics, and how they can affect the caching strategy used. While our study is for a much smaller dataset, we still believe there is value in studying VoD workloads for small and medium VoD providers with a very local user base.

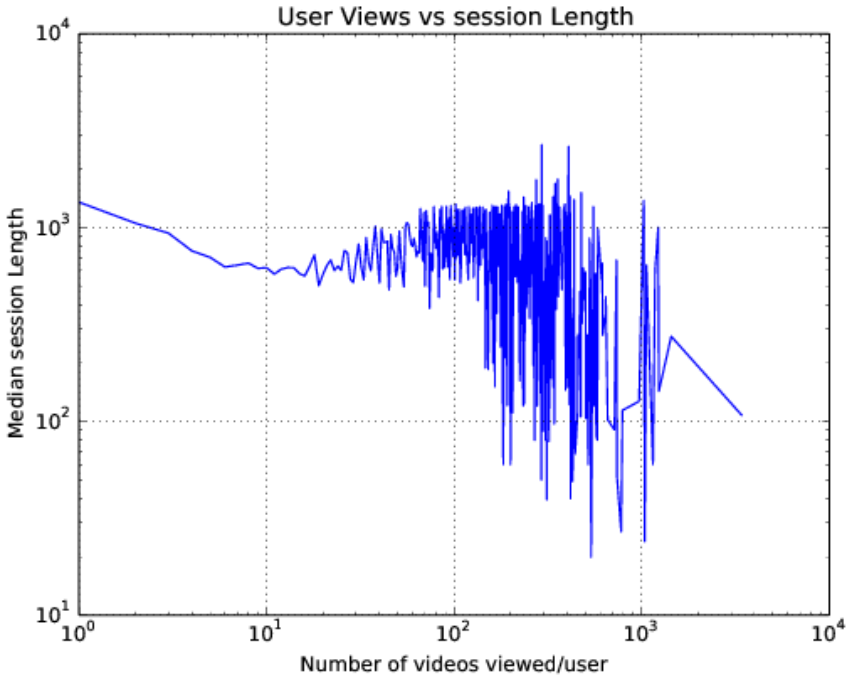


Figure 15: Users have an almost homogeneous median session lengths no matter how often do they use the service (Log-Log plot).

Choi et al. analyzed service logs generated for every VoD request or VoIP call made on a day in April, 2009 in a nationwide commercial IP network in Korea [12]. The number of subscribers for both services is over 1.2 million generating over 10.5 million requests in total during that day. The authors focus on workload characteristics having a direct effect on the performance of IP networks such as session arrivals and session holding times.

A recent study analyzed both sender and receiver side logging data provided by a popular Norwegian streaming provider [32]. The logs are 24-hour logs collected for 8 soccer games with 1328 unique client IP addresses. The authors deduce from the segments downloaded multiple times that at least 3% of the bytes downloaded could have been saved if more HTTP caches or a different adaptation strategy had been used.

Multiple studies have crawled and analyzed traces from YouTube, Yahoo! videos and DailyMotion. Video streaming from an ISP perspective has been studied by Pilssonneau and Biersack by analyzing 10 packet traces from a

residential ISP network [40]. They focus on video streams from YouTube and DailyMotion with a focus on analyzing the flow performance of the videos and the user behavior of the two services. The authors study the influence of the reception quality on the users ans show that videos with bad reception quality are seldom fully downloaded.

Khemmarat et al. [29] collect user browsing pattern data for YouTube. They show that video buffering affects the QoS for YouTube users due to disruptions for buffering. The authors propose a video prefetching approach for user-generated video sharing sites like YouTube based on the site's recommended videos list for any video. They show that prefetching considerably improves the QoS of YouTube.

Kang et. al. crawled Yahoo! videos website for 46 days [27]. Around 76% of the videos crawled are shorter than 5 minutes and almost 92% are shorter than 10 minutes. They discuss the predictability of the arrival rate with different time granularities. A load spike typically lasted for no more than one hour and the load spikes are dispersed widely in time making them hard to predict. Gill et al. [19] collected data on all YouTube usage at the University of Calgary network for 85 consecutive days, starting January 14, 2007. In addition, they monitored the 100 most popular videos on YouTube for the same period. They examined the usefulness of caching and content distribution networks for improving performance and scalability of similar applications. Similarly, Chang et al. crawled YouTube for four months in early 2007 collecting data for more than 3 million videos [11]. Their study did not consider the rate of request arrivals for the different videos but rather focused on some statistics such as the video category, length, size and bitrate. They also discuss some of the social networking aspects of YouTube.

Barker and Shenoy studied the effect of background workloads on the QoS of a multimedia service hosted on a cloud system [9]. They show that co-located applications can affect the QoS perceived by the multimedia service customer considerably. The degree of interference variations is most pronounced for disk-bound latency-sensitive tasks, which can degrade by nearly 75% under sustained background load. Their experiments revealed two main insights, the lack of proper disk isolation mechanisms in the hypervisor between co-located VMs can hurt performance, and that network isolation mechanisms in the hypervisor present a trade-off between mean latency and metrics such as jitter and timeouts. Having dedicated caps on the network usage yield lower average latency, while fair sharing the network between the VMs yields lower timeouts and somewhat lower jitter.

The impact of video streaming quality on user engagement and abandonment rates has been studied by Dobrian et al. [16] and Krishnan and Sitaraman [31].

Both studies assert the effect of buffering on user engagement. Dobrian et al. also show the significance of the average streaming bit-rate on live content.

5 conclusion

Video-on-Demand workloads are not well studied in the literature. Our analysis of VoD traces from a Swedish service provider is aimed to provide some insights to better understand and design VoD systems. The results of our analysis show that the user and request arrival rates *can not* be modeled as a Poisson process in the analyzed traces. The arrival rates can be modeled using a lognormal distribution while the inter-arrival time can be modeled using an extended exponential distribution. There are four spikes in the workload caused due to football matches that interested the Swedish audience. Three of these matches were played in foreign football leagues, unrelated or weakly related to Sweden, making the spikes in the load hard to plan for without extensive social analysis of what attracts the local population. Comparing the user behavior in our study to the user behavior in Yu et al.'s study [48], we can conclude that the rate of users abandoning streaming sessions a few minutes from when they start it seems to be an invariant in VoD workloads. In both studies, 50% of the sessions started were abandoned after less than 12 minutes from their beginning by the highly "impatient users" of the VoD services. That is despite of our study being conducted on a Swedish VoD service and their study being conducted on a Chinese VoD Service with almost six years between the two studies. This impatient behavior can be used to improve prefetching and caching of the videos provided by the VoD service provider.

6 Acknowledgment

We thank TV4 for providing us with the workload traces. Financial support has been provided in part by the Swedish Government's strategic effort eSSENCE, the European Unionâs Seventh Framework Programme under grant agreement 610711 for the project CACTOS, and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control. We would also like to thank the anonymous reviewers for their constructive comments.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). `https://aws.amazon.com/solutions/case-studies/`. Accessed: October, 2014.

[2] L. A. Adamic. Zipf, power-laws, and pareto-a ranking tutorial. *Xerox Palo Alto Research Center, Palo Alto, CA, http://ginger.hpl.hp.com /shl/papers/ranking/ranking.html*, 2000.

[3] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang. Unreeling netflix: Understanding and improving multi-cdn movie delivery. In *INFOCOM*, pages 1620–1628. IEEE, 2012.

[4] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *ScienceCloud*, pages 31–40. ACM, 2012.

[5] A. Ali-Eldin, A. Rezaie, A. Mehta, S. Razroev, S. Sjöstedt-de Luna, O. Seleznjev, J. Tordsson, and E. Elmroth. How will your workload look like in 6 years? analyzing wikimedia's workload. In *IC2E*, pages 349–354. IEEE, 2014.

[6] A. Ali-Eldin, O. Seleznjev, S. Sjöstedt-de Luna, J. Tordsson, and E. Elmroth. Measuring cloud workload burstiness (to appear). In *UCC*. IEEE Computer Society, 2014. Preprint available at: `https://www8.cs.umu.se/~ahmeda/CloudControl6.pdf`.

[7] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, 2000.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *PER*, volume 40, pages 53–64. ACM, 2012.

[9] S. K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMSys*, pages 35–46. ACM, 2010.

[10] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *IM*, pages 119–128. IEEE, 2007.

[11] X. Cheng, C. Dale, and J. Liu. Statistics and social network of youtube videos. In *IWQoS*, pages 229–238. IEEE, 2008.

[12] Y. Choi, J. A. Silvester, and H.-C. Kim. Analyzing and modeling workload characteristics in a multiservice ip network. *IEEE Internet Computing*, 15(2):35–42, 2011.

[13] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[14] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning. Stl: A seasonal-trend decomposition procedure based on loess. *Journal of Official Statistics*, 6(1):3–73, 1990.

[15] S. N. Demographics. Global internet phenomena report: Autumn 2013, 2013.

[16] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *SIGCOMM*, pages 362–373. ACM, 2011.

[17] P. Flandrin, G. Rilling, and P. Goncalves. Empirical mode decomposition as a filter bank. *IEEE Signal Processing Letters*, 11(2):112–114, 2004.

[18] L. Gao and D. Towsley. Supplying instantaneous video-on-demand services using controlled multicast. In *ICMCS*, volume 2, pages 117–121. IEEE, 1999.

[19] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. Youtube traffic characterization: A view from the edge. In *IMC*, pages 15–28. ACM, 2007.

[20] K. HIRAI. A philosophy of kando: Cultivating curiosity to reclaim the power of âWOWâ. online: http://blog.sony.com/press/sony-ces-2014-keynote-transcript/.

[21] C. Huang, J. Li, and K. W. Ross. Can internet video-on-demand be profitable? *ACM SIGCOMM Computer Communication Review*, 37(4):133–144, 2007.

[22] N. E. Huang and S. S. Shen. *Hilbert-Huang transform and its applications*, volume 5. World Scientific, 2005.

[23] N. E. Huang, Z. Shen, S. R. Long, M. C. Wu, H. H. Shih, Q. Zheng, N.-C. Yen, C. C. Tung, and H. H. Liu. The empirical mode decomposition and the hilbert spectrum for nonlinear and non-stationary time series analysis. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1971):903–995, 1998.

[24] M. Jeon, Y. Kim, J. Hwang, J. Lee, and E. Seo. Workload characterization and performance implications of large-scale blog servers. *ACM TWEB*, 6(4):16, 2012.

[25] S. Jin and A. Bestavros. GISMO: a generator of internet streaming media objects and workloads. *ACM SIGMETRICS PER*, 29(3):2–10, 2001.

[26] F. Johnsen, T. Hafsoe, C. Griwodz, and P. Halvorsen. Workload characterization for news-on-demand streaming services. In *IPCCC*, pages 314–323. IEEE, April 2007.

[27] X. Kang, H. Zhang, G. Jiang, H. Chen, X. Meng, and K. Yoshihira. Understanding internet video sharing site workload: A view from data center design. *Journal of Visual Communication and Image Representation*, 21(2):129–138, 2010.

[28] M. Kendall, A. Stuart, and J. K. Ord. The advanced theory of statistics. *The advanced theory of statistics.*, (4th Ed), 1983.

[29] S. Khemmarat, R. Zhou, L. Gao, and M. Zink. Watching user generated videos with prefetching. MMSys, pages 187–198. ACM, 2011.

[30] M. Kihl, E. Elmroth, J. Tordsson, K.-E. Årzén, and A. Robertsson. The challenge of cloud control. In *8th International Workshop on Feedback Computing*, 2013.

[31] S. S. Krishnan and R. K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. *IEEE/ACM ToN*, 21(6):2001–2014, Dec. 2013.

[32] T. Kupka, C. Griwodz, P. Halvorsen, D. Johansen, and T. Hovden. Analysis of a real-world http segment streaming case. In *EuroiTV*, pages 75–84. ACM, 2013.

[33] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf. Monalytics: online monitoring and analytics for managing large scale data centers. In *ICAC*, pages 141–150. ACM, 2010.

[34] J. Laherrere and D. Sornette. Stretched exponential distributions in nature and economy:âfat tailsâ with characteristic scales. *The European Physical Journal B-Condensed Matter and Complex Systems*, 2(4):525–539, 1998.

[35] R. Lawler. Verizon taps clearleap for cloud-based vod content delivery. online: http://gigaom.com/2010/07/26/verizon-taps-clearleap-for-cloud-based-vod-content-delivery/.

[36] R. McGill, J. W. Tukey, and W. A. Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[37] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. In *ACM EuroSys*, pages 237–250. ACM, 2010.

[38] D. Niu, H. Xu, B. Li, and S. Zhao. Quality-assured cloud bandwidth auto-scaling for video-on-demand applications. In *INFOCOM*, pages 460–468. IEEE, 2012.

[39] V. Paxson and S. Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM ToN*, 3(3):226–244, 1995.

[40] L. Plissonneau and E. Biersack. A longitudinal view of http video streaming performance. In *MMSys*, pages 203–214. ACM, 2012.

[41] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, pages 7:1–7:13. ACM, 2012.

[42] G. Rilling, P. Flandrin, P. Goncalves, et al. On empirical mode decomposition and its algorithms. In *IEEE-EURASIP workshop on nonlinear signal and image processing*, volume 3, pages 8–11. NSIP-03, Grado (I), 2003.

[43] B.-M. Ringfjord. Learning to become a football star : Representations of football fan culture in swedish public service television for youth. In *We love to hate each other : mediated football fan culture*, pages 285–299. 2012.

[44] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. *ACM Sigplan Notices*, 46(7):111–120, 2011.

[45] L. Tomas and J. Tordsson. An autonomic approach to risk-aware data center overbooking. *IEEE ToCC*, 2014. Pre-print.

[46] Z. Wu and N. E. Huang. Ensemble empirical mode decomposition: a noise-assisted data analysis method. *Advances in adaptive data analysis*, 1(01):1–41, 2009.

[47] H. Xi, J. Zhan, Z. Jia, X. Hong, L. Wang, L. Zhang, N. Sun, and G. Lu. Characterization of real workloads of web search engines. In *IISWC*, pages 15–25. IEEE, 2011.

[48] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng. Understanding user behavior in large-scale video-on-demand systems. In *EuroSys*, pages 333–344. ACM, 2006.

Paper VI

PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications.

A. Papadopoulos, A. Ali-Eldin, J. Tordsson, K.E. Årzén, and E. Elmroth

Submitted for Journal Publication.

PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications*

Alessandro Papadopoulos[†], Ahmed Ali-Eldin[‡]
Karl Erik Årzén[†], Johan Tordsson[‡], and Erik Elmroth[‡]

Abstract

Numerous auto-scaling strategies have been proposed in the last few years for improving various Quality of Service (QoS) indicators of cloud applications, e.g., response time and throughput, by adapting the amount of resources assigned to the application to meet the workload demand. However, the evaluation of a proposed auto-scaler is usually achieved through experiments under specific conditions, and seldom includes extensive testing to account for uncertainties in the workloads, and unexpected behaviors of the system. These tests by no means can provide guarantees about the behavior of the system in general conditions. In this paper, we present PEAS, a Performance Evaluation framework for Auto-Scaling strategies in the presence of uncertainties. The evaluation is formulated as a *chance constrained optimization problem*, which is solved using *scenario theory*. The adoption of such a technique allows one to give probabilistic guarantees of the obtainable performance. Six different auto-scaling strategies have been selected from the literature for extensive test evaluation, and compared using the proposed framework. We build a discrete event simulator and parameterize it based on real experiments. Using the simulator, each auto-scaler's performance is evaluated using 796 distinct real workload traces from projects hosted on the Wikimedia foundations' servers, and their performance is compared using PEAS. The evaluation is carried out using different performance metrics, highlighting the flexibility of the framework, while providing probabilistic bounds on the evalu-

*The paper has been re-typeset to match the thesis style (with authors biographies excluded)

[†]Department of Automatic control, Lund University, Sweden, email: {alessandro.papadopoulos,karlerik}@@control.lth.se

[‡]Department of Computing Science, Umeå University, Sweden, email: {ahmeda, tordsson, elmroth}@cs.umu.se

ation and the performance of the algorithms. Our results highlight the problem of generalizing the conclusions of the original published studies and show that based on the evaluation criteria, a controller can be shown to be better than other controllers.

1 Introduction

Elasticity can be defined as the property of a cloud infrastructure (datacenter) or a cloud application to dynamically adjust the amount of allocated resources to meet changes in workload demands. In principle, resources allocated to a running service should be varied such that the Quality of Service (QoS) requirements are preserved at minimum cost. A large number of auto-scalers have been designed for this purpose, using different approaches such as control theory [41], neural networks [33], second order regression [32], histograms [63], time-series models [29], the secant method [47], and look-ahead control [58]. While very different in nature, all of these controllers have one thing in common: They are designed to provision resources according to the changing workloads of the applications deployed in a cloud.

Cloud (and datacenter) workloads may have very different characteristics. They vary in mean, variance, burstiness, periodicity, type of required resources, and many other aspects. When studying workload periodicity and burstiness profiles, one can find workloads that have repetitive patterns with daily, weekly, monthly and/or annual cycles, e.g., the Wikipedia workload shown in Figure 1(a) has diurnal patterns with small deviations in the pattern from one day to the other [5]. Other workloads may have uncorrelated spikes and bursts that occur due to an unusual event, e.g., Figure 1(b) shows the load on the Wikipedia page about Michael Jackson with two significant spikes that occurred immediately after his death and during his memorial service. On the other hand, some workloads have weak or no recognizable patterns at all, e.g., the workload shown in Figure 1(c) for tasks submitted to a Google cluster [67]. Most often, the characteristics of the workloads of new applications are unknown in advance.

Unfortunately, most of the state-of-the-art auto-scalers have been evaluated using less than three real workload traces – in a real deployment or in simulation – which makes it impossible to generalize the obtained results [29, 32, 41, 59, 63]. In addition, the traces used for testing are typically short spanning a few minutes, hours or days [29,58,63]. As the workload profile is well known to affect the performance of a controller and the running application, it is not possible to tell whether one auto-scaler is better than the other for a certain workload [7, 24].

This paper presents PEAS (Performance Evaluation framework for Auto-Scaling strategies), a framework for the evaluation of auto-scaling techniques. PEAS reformulates the performance evaluation problem into a chance constrained optimization problem, whose solution requires an extensive yet targeted testing of the auto-scaling

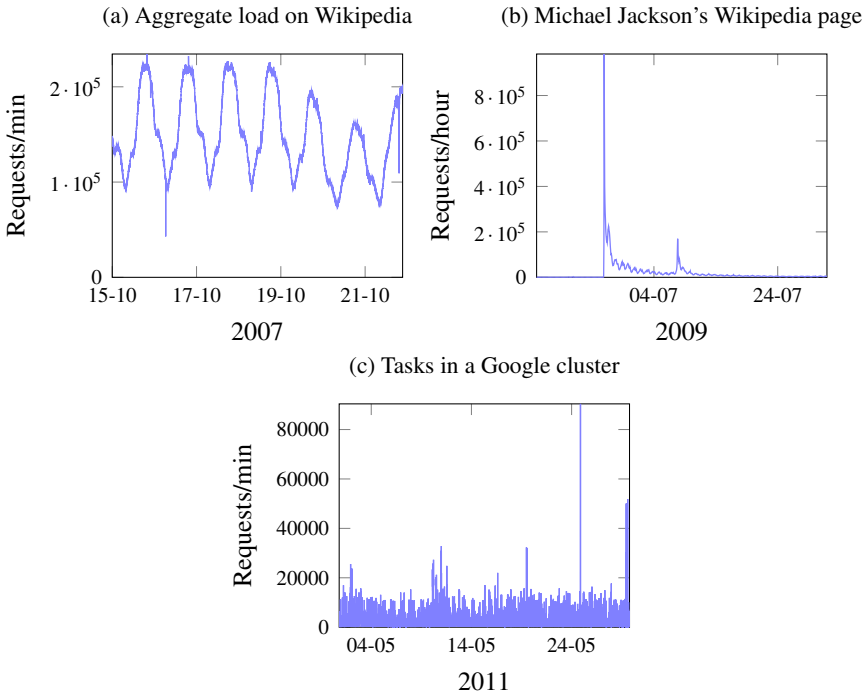


Figure 1: Different workloads have different burstiness. Plot (a) shows a workload with little burstiness and strong periodicity. Plot (b) shows sudden spikes in a mostly periodic workload due to an event. Plot (c) shows a bursty workload.

techniques. The solution of the chance constrained optimization is carried out through *scenario theory* [14, 16, 49], which has been recently applied to control theory problems related to complex, stochastic, or uncertain systems [15, 17, 53]. With the adoption of the scenario theory, PEAS can be used to extensively evaluate the performance of different auto-scaling strategies, providing probabilistic guarantees on the obtained results.

To show the feasibility of using scenario theory to compare auto-scalers, we obtained the implementations of three state-of-the-art auto-scalers from their designers [6, 25, 51]. In addition, we reimplemented three other auto-scalers from the literature [20, 32, 63]. We used a publicly available workload from the Wikimedia foundation spanning roughly six years [48] to evaluate the six algorithms. The workload traces are composed of more than 3500 streams comprising all requests to all Wikimedia foundation's projects with all 287 supported languages, of which a subset of 796 streams is selected.

In summary, the contribution of this work is twofold. First, we introduce a framework that provides probabilistic guarantees on the QoS achieved by auto-scalers while permitting to compare their performance in worst-case scenarios. Secondly, we compare six state-of-the-art auto-scalers using a large real workload spanning 6 years from a production system.

The rest of the paper is organized as follows. Section 2 discusses the problem of evaluating different auto-scalers and comparing them along with the tradeoffs of choosing one auto-scaler over another. Section 3 discusses some of the auto-scaling techniques that have been proposed in the literature. We give special attention to the six algorithms we evaluate with the proposed framework. Section 4 introduces PEAS, focusing on its theoretical aspects. Section 5 discusses the experimental results. We conclude in Section 6.

2 The case for PEAS

While cloud computing is a relatively new technology that was enabled recently by the advances in virtualization; elasticity control and auto-scaling of cloud resources is an incarnation of the dynamic resource provisioning problems in clusters, grids and datacenters. The dynamic resource provisioning problem – and its variants – have been discussed over the past two decades with initially some patents in the early 1990s [23, 42], followed by an interest from the research community later in the late 1990s and early 2000s [19, 64]. During this period, many algorithms have been proposed for dynamic provisioning with different flavors before and after the term cloud computing was coined [20, 25, 27, 63]. Since then, many new auto-scaling techniques have been proposed.

As for the performance assessment of auto-scalers, many published solutions have not been compared to previously proposed ones so far, but have been rather compared with a predefined required response time for the tested application or against static provisioning [2, 20, 29, 32, 40, 58]. In addition, many of the published work have not been tested with more than one real workload [2, 25, 29, 33, 37, 44, 46, 47, 59]. Even when algorithms are tested with more than one workload, many of these workloads are typically quite old, from systems and services that are not even used today [6, 27, 28]. While all published previous work show that "the proposed algorithm" works in certain scenarios, the results in many of these papers cannot be generalized.

Many proposed state-of-the-art auto-scalers use the desired response time or service latency as a reference signal to a controller. A proposed approach then works if it meets the experiments' required response times, mostly set by the authors based on some QoS studies. On the other hand, response time does not show under- and over-provisioning, i.e., if the allocated capacity is too low or too high with respect to the actually needed capacity to serve the incoming requests within the desired response

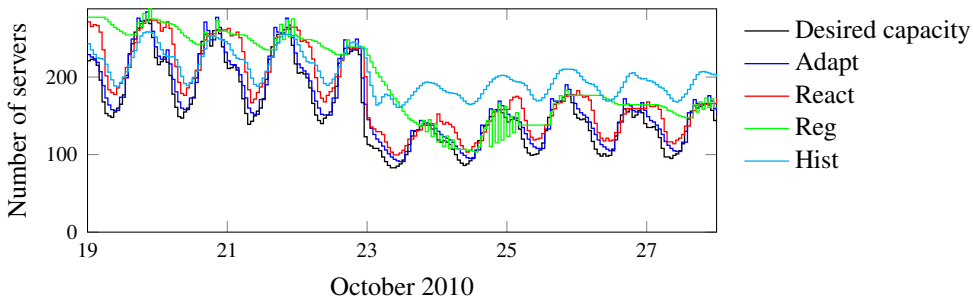


Figure 2: There are tradeoffs when choosing which auto-scaler to use in order to control server provisioning.

time. Under- and over-provisioning are indeed extremely important to compare the performance of different controllers. Two controllers might be achieving the required response time while one of them uses, for example, on average only one tenth of the resources used by the other controller. Similarly, one controller can be dropping, on average, fewer requests than another one [27]. In addition, latencies are in general nonlinear with respect to the provisioned resources. To give an example, suppose there are two workloads running on two different machines and each of them experiences an average service latency of 50ms for a request. If the required QoS is to keep the average service latency below 100ms, can then the two workloads be handled by a single server? The answer is likely to be no, since service latencies do not depend linearly on the amount of resources provisioned. This nonlinear relationship between resources provisioned and response time has been observed in large-scale distributed systems [21]. In addition, latency measurements are often extremely noisy with very high variance, even for constant workloads, [10, 11, 66]. To conclude, request response times and latencies are by themselves not suitable metrics to compare auto-scalers [10].

There are tradeoffs between using different auto-scalers in terms of over-provisioning, under-provisioning, and the time and resources required to compute the required capacity. Figure 2 shows the output for four different controllers, [6] in blue, [20] in red, [32] in green, and [63] in cyan, when used to predict the number of servers required to serve the load on the English Wikipedia pages between October 19th, 2010 and October 28th, 2010. The figure also shows the theoretical optimal number of servers required to serve such a workload (shown as the black line).[1] The first peaks of the load are part of the bursty period. The rest is when the load begins to

[1] Section 4.2.1 explain in more details how the optimal and the predicted number of servers are obtained.

settle down. One can see that during this transient period, the four auto-scalers show different behaviors. The behavior does not just depend on the selected auto-scaler, but also on how they are parameterized. The same auto-scaler with another set of parameters will typically behave in a different way.

In this work, we have been greatly inspired by the seminal work by Keogh and Kasetty which shows that many of the proposed algorithms by the time series data mining community are of very little use [34] due to lack of testing on enough datasets or due to hidden parametrization pitfalls. Our main motivation for PEAS is to help organize the design space of auto-scalers. We believe that PEAS will help the research community to establish a theoretical framework enabling the possibility of comparing in a systematic way different auto-scalers while providing probabilistic guarantees on their performance. Such guarantees allow service providers to choose a suitable auto-scaler that fulfills a desired QoS metric for their infrastructure. We envision that such an approach to testing auto-scalers may lead to a more mathematically grounded definition of Service Level Agreements (SLAs) [60] and better design of auto-scalers.

3 Related work

3.1 Selected auto-scaling methods

A significant number of auto-scalers have been proposed in the literature [43]. We choose six auto-scalers to extensively compare in this work. These six are in our opinion a representative sample of the state-of-the-art. The selected methods have been published in the following years 2008 [63] (with an earlier version published in 2005 [62]), 2009 [20], 2011 [32], 2012 [6], 2013 [51] and 2014 [25]. They thus represent the development of the field over a period of more than 7 years.

1. Urgaonkar et al. [63] propose a provisioning technique for multi-tier Internet applications. The proposed methodology adopts a queuing model to determine how many resources to allocate in each tier of the application. A predictive technique based on building **Hist**ograms of historical request arrival rates is used to determine the amount of resources to provision at an hourly time scale. Reactive provisioning is used to correct errors in the long-term predictions or to react to unanticipated flash crowds. The authors also propose a novel datacenter architecture that uses Virtual Machine (VM) monitors to reduce provisioning overheads. The technique is shown to be able to improve responsiveness of the system, also in the case of a flash crowd. We refer to this technique as **Hist**. The authors test their approach using two open source applications, RUBIS which is an implementation of the core functionality of an auctioning site, and Rubbos a bulletin-board application modeled after an online news forum. The testing is performed using 13 VMs running on Xen. Traces from the FIFA 1998 worldcup

servers are scaled in time and intensity and used in the experiments. We have implemented this auto-scaler for our experiments.

2. Chieu et al. [20] present a dynamic scaling algorithm for automated provisioning of VM resources based on the number of concurrent users, the number of active connections, the number of requests per second, and the average response time per request. The algorithm first determines the current web application instances with active sessions above or below a given utilization. If the number of overloaded instances is greater than a predefined threshold, new web application instances are provisioned, started, and then added to the front-end load-balancer. If two instances are underutilized with at least one instance having no active session, the idle instance is removed from the load-balancer and shutdown from the system. In each case the technique **React**s to the workload change. For the rest of the paper, we refer to this technique as **React**. The authors introduce the scaling algorithm but provide no experiments to show the performance of the proposed auto-scaler. The main reason we are including this algorithm in the analysis is that this algorithm is the baseline algorithm in our opinion since it is one of the simplest possible workload predictors. We have implemented this auto-scaler for our experiments.

3. Iqbal et al. [32] propose a methodology for automatic detection of bottlenecks in a multi-tier web application targeting maximum response time requirements. The proposed technique uses a reactive approach for scaling up resources, and a **Reg**ression model for predicting the amount of resources needed for the actual workload, and possibly retract over-provisioned resources. The technique is tested using RUBIS running on seven physical machines using a EUCALYPTUS cloud installation. httperf is used to generate a workload that increases and decreases in predefined steps. The results are compared to static provisioning. We refer to this technique as **Reg**. We have implemented this auto-scaler for our experiments.

4. Ali-Eldin et al. [6] propose an autonomous elasticity controller that changes the number of VMs allocated to a service based on both monitored load changes and predictions of future load. The predictions are based on the rate of change of the request arrival rate, i.e., the slope of the workload, and aims at detecting the envelope of the workload. The designed controller **Adapt**s to sudden load changes and prevents premature release of resources, reducing oscillations in the resource provisioning. **Adapt** tries to improve the performance in terms of number of delayed requests, and the average number of queued requests, at the cost of some resource over-provisioning. The algorithm was tested using a simulated environment using a non-scaled version of the FIFA 1998 worldcup server traces, traces from a Google cluster and traces from Wikipedia. We refer

to this technique as **Adapt**. We obtained the original code from the authors for our experiments.

5. Nguyen et al. [51] propose **AGILE**, a wavelet-based algorithm to provide a medium-term resource demand prediction with a lead time of about 2 minutes, a time that allows AGILE to possibly start up new application server instances before performance falls short. The framework also uses dynamic VM cloning to reduce application startup times. Its accuracy has been proven to be better than previous schemes in terms of true positives and false negatives. AGILE was implemented on top of KVM on a cloud testbed having 10 physical machines. The algorithm was evaluated using RUBIS and the Apache Cassandra key-value store. Four workloads were used in the evaluation, namely, the FIFA World Cup 1998 webserver trace starting at 1998-05-05:00.00; NASA webserver trace beginning at 1995-07-01:00.00; EPA webserver trace starting at 1995-08-29:23.53; and ClarkNet web server trace beginning at 1995-08-28:00.00. The prediction algorithm was also tested using real system resource usage data collected on a Google cluster. The prediction results are compared to two previously published algorithms, autoregression [13] and PRESS [28]. The provisioning results are compared also to a reactive approach and a threshold based approach that increases and decreases the number of machines based on preset resource usage levels. We obtained the original code from the authors for our experiments.

6. Fernandez et al. [25] present a methodology that scales a web application in response to changes in throughput at fixed intervals of 10 minutes. The proposed algorithm is based on different components. The **P**rofiler measures the computing capacity of different hardware configurations when running an application. The predictor forecasts the future service demand using standard time-series analysis techniques, e.g., Linear Regression, Auto Regressive Moving Average (ARMA), etc. The dynamic **L**oad **B**alancer distributes the requests in the datacenter according to the capacities of the provisioned resources. Finally, the **S**caler uses the predictor and the profiler to find the scaling plan that fulfills a prescribed Service Level Objective (SLO). For evaluation, the authors test their technique on two cloud infrastructures, a private one (the DAS-4, a multi-cluster system hosted by universities in The Netherlands) and a public one (the Amazon EC2 cloud). The authors deployed MediaWiki application instances on both environments using the WikiBench benchmark to run the application. The benchmark uses a copy of Wikipedia, and replays a fraction of the actual Wikipedia access traces. The experiments used up to 12 EC2 VMs of different types and up to 8 machines of varying configurations in the DAS-4 system. In

our work, we refer to this technique as **PLBS**. The code for this auto-scaler is open-sourced. We downloaded the authors' original implementation.

3.2 Performance evaluation methodologies

Whereas a significant effort was spent by many researchers on designing "efficient" auto-scalers, not as much effort was spent on defining an evaluation methodology to compare the designed algorithms. Typically, the designed auto-scalers are evaluated in an *ad hoc* way, with a limited number of experiments, that may not be necessarily representative of the actual workload of a cloud environment, or more generically, of a service provider.

There is, however, an increasing interest in defining some common practices and methodologies for the evaluation of auto-scaling techniques [31, 38, 65]. For example, Ferraris et al. [26] present the results of stressing the auto-scaling mechanisms implemented by Amazon and Flexiscale cloud providers. Netto et al. [50] introduce an index for evaluating the performance achieved by an auto-scaling technique. The index is called "Auto-scaling Demand Index" (ADI), and it is used for penalizing differences between the actual and the desired resource utilization levels.

To the best of our knowledge, there is no standard methodology for the evaluation of auto-scaling strategies, that allows also for the definition of probabilistic guarantees on the performance. The evaluation is typically carried out through few experiments, but by no means to quantify the confidence that the obtained results will hold also in other situations. As a result, it is difficult, if not impossible, to use such results for the definition of performance SLAs.

4 PEAS: the evaluation framework

PEAS is a framework that is able to provide probabilistic guarantees on the performance of auto-scaling strategies. The proposed framework is based on recent developments in the control and optimization community, related to the solution of Chance-Constrained optimization Problems (CCPs) via scenario theory [14, 16, 49, 56]. Scenario theory represents a systematic theoretical foundation of the "many-experiments" approach, i.e., when one wants to evaluate the robustness of the system, a certain number of experiments is performed aimed at quantifying the confidence about the quality of the system. The scenario optimization approach has been adopted to tackle several problems in control theory, ranging from systems and control design [17], to robust control design [15], from model order reduction of stochastic hybrid systems [53] to game theory [12].

The steps of the framework can be summarized as follows.

1. Definition of the performance measurements.

2. Definition of a suitable performance metric.
3. Formulation of the CCP.
4. Application of the scenario theory.
   (a) Choice of the parameters.
   (b) Computation of the experimental results.
   (c) Computation of the randomized solution.

The formulation of the performance evaluation problem as a CCP is a nontrivial task, and is one of the contributions of this paper.

4.1 Chance-constraint optimization and scenario theory

We provide a brief overview of CCP and the main results of the scenario theory before going into the details of the framework. Consider the optimization problem

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} \phi(x) \\ &\text{subject to: } G(x,\xi) \in C. \end{aligned} \tag{1}$$

where $C \in \mathbb{R}^m$ is a closed convex set, and $\phi(x)$ is a real valued function. The mapping $G : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^m$ depends on uncertain parameters ξ which vary in a set $\Xi \in \mathbb{R}^d$. For a fixed ξ the constraint is either satisfied or not. If we consider ξ as a random vector with a probability distribution having support Ξ, finding the optimal value of x would require to have the constraint $G(x,\xi) \in C$ to be satisfied with probability 1. Intuitively, this would mean to have an infinite number of constraints, one for each possible realization of ξ. This is known in the stochastic programming literature as the second stage feasibility problem that has to be solvable with probability 1 [56]. However, this is too conservative in most practical situations. A more realistic requirement is to ensure feasibility with probability close to 1, i.e., at least $1-\varepsilon$. Hence, we can formulate problem (1) as a Chance-Constrained optimization Problem (CCP)

$$\begin{aligned} CCP : &\min_{x\in\mathbb{R}^n} \phi(x) \\ &\text{subject to: } \mathbb{P}\{G(x,\xi) \in C\} \geq 1-\varepsilon. \end{aligned} \tag{2}$$

CCPs have been widely studied in the literature, and are known to be NP-hard [49,56], thus only approximate solutions can be found.

Scenario theory or the "scenario optimization approach" is a technique for obtaining approximate solutions to robust optimization and CCPs based on randomization of the constraints [14, 16, 49]. Scenario theory enables one to decide what is the level

of confidence required, i.e., the value of the parameter ε in the CCP (2), and it tells how many experiments one needs to perform in order to find a feasible estimate solution to the problem. The chance constraints are then replaced with the realizations $G(x,\xi^{(i)}) \in C$, with $\xi^{(i)}$, $i = 1,\ldots,N$, and thus an approximate solution can be easily found.

4.2 Performance evaluation

This section presents the theoretical foundations of PEAS. The framework is based on the idea that one can re-formulate the problem of auto-scalers performance evaluation as a CCP, that can be solved using scenario theory. The framework assumes that the arrival rate λ (measured in request per time unit) is stochastic and the goal of the auto-scaling strategy is to provision the least amount of resources able to serve the incoming traffic with a required QoS. The proposed method involves using as input of the different auto-scaling strategies some realizations of the stochastic input. In practice, this means that either the distribution of the input is known, or some of its realizations are available as historical time series. Our goal is to introduce a method for evaluating the performance of different methods with some probability guarantees, in order to understand which method is behaving better especially in critical scenarios.

4.2.1 Definition of the performance measurements

The cloud infrastructure is modeled as a $G/G/N$ stable queue in which the number of servers N is variable [39]. This is a generalization of the model proposed by Khazaei et al. where a cloud is modeled as an $M/G/m$ queue with a constant m [35]. It is assumed that the servers are used to serve a non-stationary request mix with varying request sizes [59]. Figure 3 shows the system model.

We denote with $k \in \mathbb{N}$ the discrete (slotted) time, that counts the decisions for the auto-scaler. Let then $y(k)$ be the current capacity allocated to the service at time k, and $y^{\circ}(k)$ the required capacity needed to serve the actual incoming traffic at time k. The required capacity depends on the specific application and workload considered. In determining the necessary capacity $y^{\circ}(\cdot)$, our model accounts for queuing effects, load mix variations, and long running requests, i.e., requests requiring long processing time. The required capacity at any time k is thus the total capacity required to process the newly arriving load $\lambda(k)$, the number of requests that were not served during the past and are queued $q(k)$, and the requests that have already been assigned to a server in a previous time unit, $k-m, \forall m < k$, but are still being processed. The total

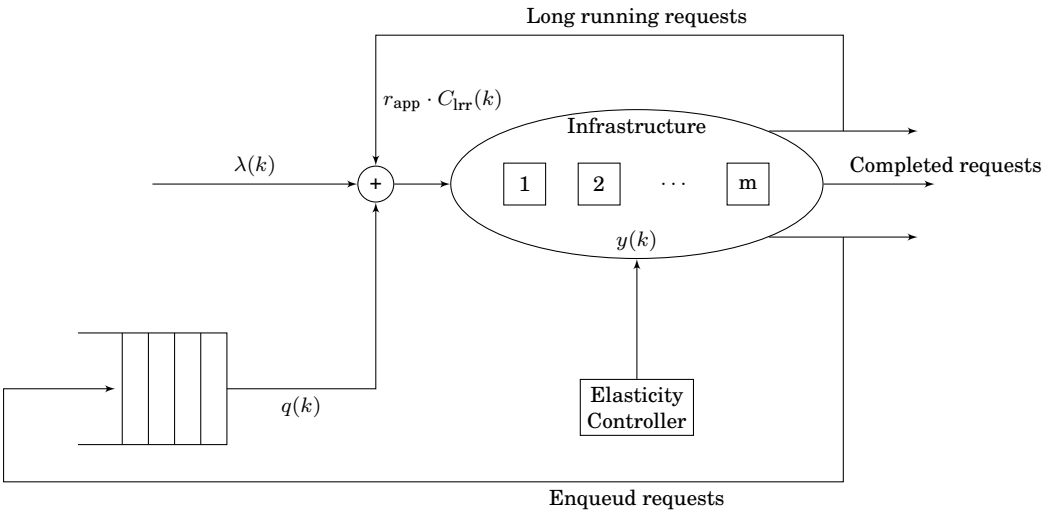


Figure 3: Queuing model for a service deployed in the cloud.

required capacity at time unit k is then computed as

$$y^{\circ}(k) = \left\lceil \frac{\lambda(k) + q(k)}{r_{\text{app}}} \right\rceil + C_{\text{lrr}}(k), \tag{3}$$

where $q(k)$ is the number of requests that are enqueued in the system and that will be processed in the next time unit, r_{app} is the average number of requests per time unit that a single VM can handle for the specific application with an acceptable service time, and $C_{\text{lrr}}(k)$ is the capacity used for processing the long running requests that require more than one time unit to be served. We denote with $\lceil x \rceil$ the ceiling function that returns the smallest integer number greater than or equal to x. An acceptable service time for a request in this case is the maximum service time that the application can tolerate and that the service user requires. For a web request to a static web-page, an acceptable service time is in seconds or even milliseconds. For a Map-Reduce job, the acceptable service time can be in minutes or even hours. This approach for finding the required capacity is similar to what has been used in the literature [27, 62, 63], but it accounts also for queuing effects and long running requests.

Long running requests are any requests that require more than the time interval between two auto-scaler decisions. Since all auto-scaling algorithms calculate the required capacity on discrete intervals, e.g., AGILE and Adapt every time unit based on the sampling interval of the monitoring data, ConPaaS every 10 minutes, and Hist every hour with corrections on a minutes scale, a long running request is a request

that needs to be processed by the system for a period that extends beyond such an interval. The addition of the long running requests effect to the model allows PEAS to take in to account the effect of workload mix changes [59] since a request that sorts a Terabyte of data requires more time than a request that fetches a static web page.

We note that Equation 3 does not represent the state of the system when any of the auto-scaling algorithms is used but rather the requirement to maintain the system with sufficient capacity to process all requests in the minimum required time. An auto-scaling algorithm should at least provision $y^\circ(k)$ resources in order to serve all requests in an acceptable service time. If an auto-scaling algorithm provisions more capacity than $y^\circ(k)$, then it should be able to serve all requests but at a a higher cost. We later discuss how we parameterize our experiments to choose all the parameters in Equation 3 and how we handle the different assumptions in the designs of the different algorithms.

4.2.2 Definition of a suitable performance metric

In order to appropriately evaluate the performance of an auto-scaling strategy, we define a distance $d_{\mathscr{T}}(\cdot,\cdot)$ that maps each pair of trajectories $y(k)$, $k \in \mathscr{T}$, and $y^\circ(k)$, $k \in \mathscr{T}$, into a positive real number $d_{\mathscr{T}}(y,y^\circ)$ that represents the extent to which $y(\cdot)$ is far (as defined below) from the ideal behavior $y^\circ(\cdot)$ along the finite time interval $\mathscr{T}$. Note that $d_{\mathscr{T}}(y,y^\circ)$ is a random quantity since it depends on the realization of the stochastic input $\lambda(k)$.

We consider different distance metrics in order to obtain synthetic results both on the overall performance of the algorithms, and of specific aspects that one could be interested in. The first distance metric that we choose is

$$d^{\text{norm}}_{\mathscr{T}}(y,y^\circ) = \frac{1}{|\mathscr{T}|}\sum_{k\in\mathscr{T}} \|y^\circ(k)-y(k)\|^2, \tag{4}$$

where $|\mathscr{T}|$ is the number of samples contained in the time interval $\mathscr{T}$ of the experiment, and $\|X\|^2$ denotes the squared 2-norm of the matrix X. This distance penalizes under-provisioning (i.e., when the difference $y^\circ - y$ is positive) and over-provisioning (i.e., when the difference $y^\circ - y$ is negative) in the same way, accounting also for those situations where the allocated resources are oscillating around the ideal capacity. Notice that the normalization with respect to $|\mathscr{T}|$ is used to account for scenarios with different length. Notice also, that it is possible to modify (4) in order to weight differently under- and over-provisioning. Whereas distance (4) is able to give synthetic information about the overall performance, its value is difficult to interpret. This is the main reason for introducing other more interpretable quantities.

As argued in [1], the directional Hausdorff distance

$$d^{\text{haus}}_{\mathscr{T}}(y,y^\circ) = \sup_{k\in\mathscr{T}} \inf_{\kappa\in\mathscr{T}} \|y^\circ(k)-y(\kappa)\|, \tag{5}$$

is a sensible choice for $d_{\mathscr{T}}(y,y^{\circ})$ when performing probabilistic verification such as, e.g., estimating the probability that y will enter some set within the time horizon $\mathscr{T}$. This distance disregards the time spent under- and over-provisioning, thus it cannot be used for the evaluation of auto-scalers. Therefore, one can choose a slightly different distance

$$d^{\sup}_{\mathscr{T}}(y,y^{\circ}) = \sup_{k\in\mathscr{T}} \|y^{\circ}(k) - y(k)\|, \tag{6}$$

that accounts also for the maximum discrepancy between the ideal and the actual behavior. This is the second distance that we use for the evaluation.

We here consider also two other metrics that give information about how much the auto-scaling strategy is over- and under-provisioning. These can be thus expressed as

$$d^{\text{over}}_{\mathscr{T}}(y,y^{\circ}) = \sup_{k\in\mathscr{T}} \|\max\{y(k) - y^{\circ}(k), 0\}\|, \tag{7}$$

$$d^{\text{under}}_{\mathscr{T}}(y,y^{\circ}) = \sup_{k\in\mathscr{T}} \|\max\{y^{\circ}(k) - y(k), 0\}\|. \tag{8}$$

However, it is also desirable that $y(\cdot)$ converges towards $y^{\circ}(\cdot)$ as soon as possible, thus producing an error converging to zero. This is extremely important from the service provider perspective, since excessive over-provisioning for long time results in a waste of resources, while excessive under-provisioning results in loss of revenue. In order to account for the time aspect, we introduce two other metrics

$$d^{\text{overT}}_{\mathscr{T}}(y,y^{\circ}) = \frac{1}{|\mathscr{T}|} \sum_{k\in\mathscr{T}} \|\max\{y(k) - y^{\circ}(k), 0\}\|\Delta k, \tag{9}$$

$$d^{\text{underT}}_{\mathscr{T}}(y,y^{\circ}) = \frac{1}{|\mathscr{T}|} \sum_{k\in\mathscr{T}} \|\max\{y^{\circ}(k) - y(k), 0\}\|\Delta k. \tag{10}$$

where Δk is the time interval between two subsequent interventions of the auto-scaling strategy. The interpretation of these two distance is thus, the average over- and under-provisioning in a time unit.

The last distance the framework uses is an adapted version of the Auto-scaling Demand Index (ADI) proposed by Netto et al. [50]. The utilization level can be defined as

$$u(k) = \frac{y(k)}{y^{\circ}(k)}. \tag{11}$$

Notice that $u(k) \geq 0, \forall k \in \mathbb{N}$ but unlike the ADI measure proposed by Netto et al. [50], $u(k)$ can be larger than 1, representing a situation of over-provisioning. The adapted ADI is represented by σ, and defined as

$$\sigma = \sum_{k\in\mathscr{T}} \sigma(k), \tag{12}$$

where

$$\sigma(k) = \begin{cases} L - u(k) & \text{if } u(k) \leq L, \\ u(k) - U & \text{if } u(k) \geq U, \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

with $0 \leq L \leq U$ representing the required lower and upper bound of the utilization. The adapted ADI provides information analogous to (4), as it is evaluating over time the discrepancy between the actual and required capacity. In the following, we set $L = 0.9$ and $U = 1.1$, meaning that we are not penalizing strategies that oscillate in an interval $(L \cdot y^{\circ}(k), U \cdot y^{\circ}(k))$. Since ADI penalizes longer workload traces, for a fair comparison we consider its normalized version with respect to the length of the time horizon $|\mathscr{T}|$,

$$\sigma_{\mathscr{T}} = \frac{\sigma}{|\mathscr{T}|} = \frac{1}{|\mathscr{T}|} \sum_{k \in \mathscr{T}} \sigma(k). \tag{14}$$

4.2.3 Formulation of the CCP

The problem of evaluating the performance of the different auto-scaling strategies can thus be formulated as the CCP,

$$\begin{aligned} CCP : \min_{\rho} \rho \\ \text{subject to: } \mathbb{P}\{d_{\mathscr{T}}(y, y^{\circ}) \leq \rho\} \geq 1 - \varepsilon. \end{aligned} \tag{15}$$

where the performance is evaluated in a "worst-case" fashion. This form for the CCP is sometimes referred to as *percentile optimization*, where ρ can be interpreted as the $(1-\varepsilon)-$percentile.

4.2.4 Application of the scenario theory

Irrespectively of the choice of $d_{\mathscr{T}}(y, y^{\circ})$, finding the optimal solution $\rho^{\star}$ of the CCP (15) is an NP-hard problem. Indeed, it involves determining, among all sets of realizations of the stochastic input that have a probability $1 - \varepsilon$, the one that provides the best (lowest) value for $d_{\mathscr{T}}(y, y^{\circ})$. We then head for an approximate solution. Instead of considering all the possible realizations for the stochastic input, we consider only a finite number N of them, called *scenarios*, randomly extracted according to their probability distribution, and treat them as if they were the only admissible uncertainty instances. This leads to the formulation of Algorithm 1, where the chance-constrained solution is determined using an empirical violation parameter $\eta \in (0, \varepsilon)$. In the following we denote the realization of a scenario with the superscript $^{(i)}$, with $i = 1, 2, \ldots, N$.

Algorithm 1: Computation of the randomized solution.

1: extract N realizations of the stochastic input $\lambda^{(i)}(k)$, $k = 1,2,\ldots,|\mathscr{T}|$, $i = 1,2,\ldots,N$, and let $\kappa = \lfloor \eta N \rfloor$;

2: determine the N realizations of the output signals $y^{(i)}(k)$ $k \in \mathscr{T}$, $i = 1,2,\ldots,N$, when the policy to be evaluated is fed by the extracted input instances;

3: compute
$$\hat{\rho}^{(i)} := d_{\mathscr{T}}(y^{(i)}, y^{\circ,(i)}),\ i = 1,2,\ldots,N;$$

4: determine the indexes $\{h_1, h_2, \ldots h_\kappa\} \subset \{1,2,\ldots,N\}$ of the κ largest values of $\{\hat{\rho}^{(i)}, i = 1,2,\ldots,N\}$

5: **return** $\hat{\rho}^\star = \max_{i \in \{1,2,\ldots,N\} \setminus \{h_1,h_2,\ldots,h_\kappa\}} \hat{\rho}^{(i)}$.

Hence, the CCP problem (15) is translated into a sample-based optimization program

$$\begin{aligned} SP: \min_{\rho} \ & \rho \\ \text{subject to: } & d_{\mathscr{T}}(y^{(i)}, y^{\circ,(i)}) \le \rho, \quad i \in \{1,2,\ldots,N\} \setminus \{h_1, h_2, \ldots, h_k\}, \end{aligned} \tag{16}$$

where $\{h_1, h_2, \ldots, h_\kappa\} \subset \{1,2,\ldots,N\}$ of the $\kappa = \lfloor \eta N \rfloor$ largest values of $d_{\mathscr{T}}(y^{(i)}, y^{\circ,(i)})$.

Notably, if the number N of realizations is appropriately chosen, the obtained estimate of $\rho^\star$ is chance-constrained feasible, with *a priori* specified (high) probability.

Theorem 1. *Select a confidence parameter $\beta \in (0,1)$ and an empirical violation parameter $\eta \in (0,\varepsilon)$. If N is such that*

$$\sum_{i=0}^{\lfloor \eta N \rfloor} \binom{N}{i} \varepsilon^i (1-\varepsilon)^{N-i} \le \beta, \tag{17}$$

then, the solution $\hat{\rho}^\star$ to Algorithm 1 satisfies

$$\mathbb{P}\{d_{\mathscr{T}}(y, y^\circ) \le \hat{\rho}^\star\} \ge 1 - \varepsilon, \tag{18}$$

with probability at least $1-\beta$. □

Theorem 1 provides theoretical guarantees that the chance constraints (18) are satisfied, i.e., that the solution $\hat{\rho}^\star$ of the optimization program SP (16) obtained through Algorithm 1 is feasible for the CCP (15). Theorem 1 is a feasibility theorem,

it was proven in [16], and says that the solution $\hat{\rho}^\star$ obtained by inspecting $\lfloor \eta N \rfloor$ constraints only is a feasible solution for (15) with high probability $1-\beta$, provided that N fulfills condition (17). As η tends to ε, $\hat{\rho}^\star$ approaches the desired optimal chance constrained solution $\rho^\star$. In turn, the computational effort grows unbounded since N scales as $1/(\varepsilon-\eta)$ [16], therefore, the value for η depends in practice on the time available to perform experiments.

As for the confidence parameter β, one should note that $\hat{\rho}^\star$ is a random quantity that depends on the randomly extracted input realizations and initial conditions. It may happen that the extracted samples are not representative enough, in which case the size of the violation set will be larger than ε. Parameter β controls the probability that this happens and the final result holds with probability $1-\beta$. N satisfying (17) depends logarithmically on $1/\beta$ [3]. Therefore, β can be chosen as small as 10^{-10} (and, hence, $1-\beta \simeq 1$) without N growing significantly.

It is worth mentioning that the sensitivity of this methodology mostly depends on the value of ε, rather than on β. Indeed, when β is chosen small enough, the number N of experiments required is not significantly increased, and the only improvement that we get is the confidence level of the feasibility. On the other hand, slightly changing ε affects the number of experiments significantly, but it also means that we are changing the CCP that one wants to solve. A typical use of (17) is thus deciding what is an acceptable level of risk, ε, choosing a small enough value of β, and computing the maximum number of experiments needed, according to the computational limits.

Notice that the guarantees provided by Theorem 1 are valid irrespectively of the underlying probability distribution of the input, which may not even be known explicitly, e.g., when feeding Algorithm 1 with historical time series as realizations of the stochastic input $\lambda^{(i)}$, as we are doing in this work. For the sake of completeness, we performed a correlation analysis on the input realizations used in the experiments. The results of the correlation analysis are presented in Appendix C.

5 Experiments and Results: Three case studies

In this section we consider three case studies, in which we want to understand which is the best algorithm among the ones selected in Section 3.1, according to the metrics presented in Section 4.2.2. The first and the second case studies assume that there are no queuing effects, i.e., that delayed requests are dropped, and that all requests are short and homogeneous, i.e., they require one time unit to be processed. As a result, $q(\cdot)=0$ and $C_{\text{lrr}}(\cdot)=0$ in (3). The difference between the first and the third case studies is in the workload used during the experiments as we will discuss later. The second case study, considers the case when delayed requests are queued in an infinite buffer, and the requests are non homogeneous with some short requests and some long running requests that can take up to 60 time units to process.

Long running requests and requests that can be buffered form a significant percentage of cloud workloads [57]. While there are no infinite buffers, assuming an infinite buffer in our simulations enables us to reason on the algorithm behavior and detect possible software bugs in the implementation of the algorithms, e.g., both PLBS and Agile exhibited very bad performance due to queuing effects, leading all the considered metrics to diverge to infinity in the second case study. After careful analysis, we found that for some abrupt workload changes, Agile was predicting negative values for the workload. We have added a check on the predictions from Agile to make sure that when the predictor predicts a negative value, the predictions are discarded. We were not able to spot the error in PLBS but we suspect that it is again a logical error. Spotting the problem with Agile was simplified due to the use of PEAS as the framework pointed us to where the errors occur in the predictions.

5.1 Solving the CCP

We assume that an acceptable risk is $\varepsilon = 0.05$, meaning that the probability that the chance constraints are not satisfied is 5%. Since each experiment considers a workload over almost six years, accepting that 5% of the time we might not able to fulfill (18), is a reasonable choice for a regular web service. Among all the experiments, we can choose a fraction $\eta \in [0, \varepsilon)$ that can be discarded without affecting the feasibility of the solution. We here set $\eta = 0.01$. As for the choice of the confidence level $1 - \beta$ that we have in the obtained result, as discussed in Section 4.2.3, we can take $\beta = 10^{-10}$, without affecting too much the number of experiments required for the evaluation.

Having chosen these parameters, it is possible to compute N such that it satisfies inequality (17), yielding $N = 796$. It is thus trivial to compute the number $\kappa = \lfloor \eta N \rfloor = 7$ of scenarios that can be discarded according to Algorithm 1. Publicly available workloads are scarce [11], and finding $N = 796$ workload traces to solve the SP might not be always possible. However, one can easily adapt the above parameters to fit the available dataset, yet keeping similar probabilistic guarantees. For example, choosing $\eta = 0$ and $\beta = 10^{-6}$, but keeping $\varepsilon = 0.05$, one requires $N = 270$ workloads traces, with no discarded trace. In order to have a high confidence in the feasibility of the result, and in order to be able to discard possible traces containing sporadic events that leads to bad performance, we used the parameters leading to $N = 796$.

5.2 The workloads

Since artificially generated workloads may not be a good representation of real workloads and would thus affect the results of our evaluation, we choose to perform the evaluations with real workloads. Given the scarcity of publicly available workloads, it is challenging to obtain 796 real commercial workload streams. On the other hand, the sixth most popular website on the World Wide Web is Wikipedia the open and

collaborative online encyclopedia, according to Alexa [4]. Wikipedia is hosted on the servers of the Wikimedia foundation which hosts other projects such as Wikiquotes and Wikibooks. The Wikipedia foundation provides publicly available workload traces that logs the per page aggregate number of requests per hour to all services hosted by the foundation [5, 8].

This workload is interesting since the traces can be separated into independent streams with each stream representing a project and a language, e.g., one stream can have the requests to the German Wikipedia pages, another can be for the Swedish Wikitionary project, and a third can be for the Zulu Wikibooks project, and so on. In total, the Wikimedia foundation hosts over 888 projects and language combinations. Many of these project streams can be divided even further into, for example, load from Mobile users, load generated by editors, load generated on "talk pages" and so on.

We downloaded the Wikimedia foundation traces for the period between the 9^{th} of December, 2007 and the 16^{th} of October, 2013. The traces were analyzed and all the different streams that are present in the load during this period of almost six years were extracted. The extraction yielded more than 3000 different workload streams. We chose $N = 796$ streams to be used to validate our framework. The chosen streams are mostly of long running projects spanning the whole period of the trace or are streams which have high request arrival rates. We have performed a correlation analysis on the selected workloads, and we found that they are practically not correlated. Further details on the workloads used and the correlation analysis are given in Appendix B and Appendix C respectively.

5.3 Simulating the cloud

Since it is not feasible to run 796 experiments per auto-scaler on a real testbed – meaning 4776 experiments in total – each of them using a workload spanning around 6 years with millions of requests per hour, we decided to use a simulator which we parameterize instead with some real experiments. A summary of some of the main decisions in our experiments follows.

1. We use a discrete event simulator that we parameterize using real server experiments on our server testbed. The experiments use one real application and one benchmark.

2. Many of the techniques we test have more components other than auto-scaling and capacity prediction algorithms for other functionalities. For example, ConPaas does not just predict the workload and the required capacity, but also tries to optimize the choice of the size of the VMs deployed, Agile also has components for VM cloning and migration. Since our main target is to evaluate the

accuracy of the prediction of the capacity required, when needed we have only focused on evaluating this part of the system as it is the core part of auto-scaling.

3. The traces we have are logged on hourly bases with no smaller logging granularity available. Using the data as is with the auto-scaling algorithms can undermine the fairness of the experiments since, with the exception of Hist, the algorithms are designed to operate on a seconds to few minutes interval. We have considered two main approaches to use the traces in our experiments. The first one is to interpolate the values for smaller time granularities based on either another workload that we have with a finer logging granularity or some other method such as regression. This approach is similar to the approach taken by Malkowski et al. [45]. The second approach is to scale down the workload intensity by dividing the number of requests by a factor, e.g., scaling down the workload by a factor of 3600 gives for each hour the average arrival rate per second.

   The first approach has a clear disadvantage since the interpolation can distort the actual workload and thus give wrong performance results. The second approach has the disadvantage of not capturing the transient workload effects if the workload is scaled down by a large factor, i.e., by dividing the workload by a factor, the effect of some extreme spikes that happened at a smaller time granularity can be reduced due to the down scaling of the workload. Since we did not want to introduce any artificial behavior in our evaluation, we opted for the second solution, down sizing the workload by a factor. To show that the results we obtain are still valid for bursty workloads, we choose two factors to scale down the workload, with one of the two factors small to capture the case of bursty workloads, as discussed later. Scaling-down a workload is an approach taken typically to compensate for the limitations of testbeds as done by Gandhi et al. [27] and Urgaonkar et al. [63].

To parameterize the simulator, we performed some experiments using C-Mart, an open-source cloud computing benchmark designed by Turner et al. [61] that emulates modern cloud applications, and using a MediaWiki installation on which we replicated the Spanish Wikipedia pages [8].

C-Mart is chosen since it represents a modern and up-to-date web application. Turner et al. compare C-Mart to RUBIS and TPC-W. They show through experimental evaluation that their benchmark reflects a more realistic cloud application compared to many benchmarks used in the literature. We have also used another widely used web application, Mediawiki, to complement our results. There are a few motivations behind using Mediawiki with a full replica of the Spanish Wikipedia beside the usage of C-Mart to parameterize the simulator. First, the workloads we are using are workloads directed to web services using Mediawiki, e.g., Wikipedia, Wiki-

books and Wikitionary. In addition, this application has recently gained popularity in the literature as a modern, real, and representative application of cloud and web services [9, 18, 22, 25, 36]. C-Mart contains both stateful and stateless applications. Some setups of MediaWiki can also be stateful. Since many of the algorithms tested and most of the algorithms found in the literature consider stateless applications, we choose to consider only the stateless case.

The service rate of a VM is modeled to be a Poisson process. Using a set of experiments where we have run multiple instances of both the chosen applications, we found that the average number of requests that a VM can serve per second varies considerably with the workload mix used in the experiment. For our experiments we set the number of requests that a VM can serve per second to 22 requests for a VM with one core assigned and two Gigabytes of RAM. Appendix A describes the experiments done to obtain the average number of requests a VM can serve per second.

To capture the effect of the spikes in the traces, two different factors are used in simulations for scaling down the workload. We choose the first factor to scale down the workloads to be 3600 , i.e, the number of seconds in one hour. This is used in the first case study and in the second case study. The resulting traces thus correspond to the average request arrival rate per second for each hour. This way we mainly capture the long term workload dynamic changes in our evaluation. In the second set of experiments, we scale down the workload by a factor of 60 only.

We performed additional experiments in order to estimate the startup time and shutdown time of a VM. The average startup time resulted in 29.7s, with a very low standard deviation of 0.30s, while the average shutdown time resulted in 6.72s, with a standard deviation of 0.45s. In both cases the time required to startup or shutdown a VM is fairly contained, and always less than 1 minute, i.e., the time unit considered in the simulator. Therefore, without loss of generality, the simulator considers 1 time unit of delay for the startup of a VM, and 0 time units of delay for shutting down a VM. All the tested algorithms were tuned according to this information. A more detailed analysis of the conducted experiments can be found in Appendix D. These experiments are then used to parametrize a discrete-event python-based simulator to simulate the cloud infrastructure[2]. All the auto-scalers are then evaluated using the simulator instead of the actual application deployments.

When parameterizing the auto-scalers, we followed any guidelines set by the authors in their published papers or any tips we received through direct communication with the authors. If the authors did not provide such guidelines or tips, we performed some simple parameter sweeping and sensitivity analysis for the algorithms, and tuned the algorithms accordingly. The sheer volume of the experiments we conducted

[2]The code of the simulator will be open-sourced in case this manuscript is accepted. Relevant technical details of the simulator, however, can be found in Appendix A.

and the length of the workloads allowed us to find logical errors in the codes we obtained or implemented, such as not handling division by zero. Whenever we found such a problem, we tried to fix it, leaving the original algorithm untouched.

5.4 Experimental results

In the experimental evaluation, recall that we are considering distances with respect to a desired behavior. Therefore, in all the presented results, the lower the value of the distance, the better the algorithm is performing. The following figures (Figures 4 to 7) have on the x-axis the considered algorithms, and on the y-axis the value $\hat{\rho}^\star$ obtained as a solution of the SP (16), when considering the specified distance.

The solution of the SP (16), when considering distance (4) for the different considered auto-scaling algorithms and for the three case studies is presented in Figure 4. As

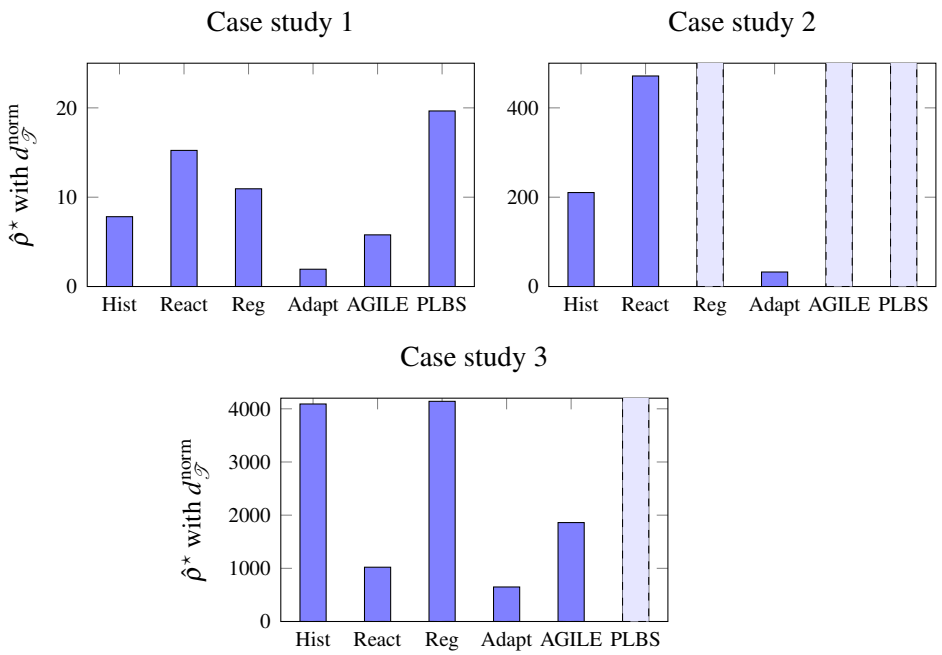


Figure 4: Results of the scenario approach with distance (4).

discussed above, this norm distance accounts for several aspects, i.e., over- and under-provisioning, how much the assigned capacity oscillates around the ideal one, and also for how long the auto-scaler is not behaving in an ideal way. As a result, $d_{\mathscr{T}}^{\text{norm}}$ gives a general idea of the overall average performance. Adapt shows better performance than the other algorithms in all three case studies, providing significantly lower values

of the considered distance. We note the difference in performance between case study 1 and case study 3. Since the scale down factor for the workloads is lower for case study 3, the workloads are much more bursty. While the performance of all algorithms for the bursty workloads is much worse, the performance of both Reg and Hist has relatively decreased more than the other algorithms while the performance of React has relatively improved, i.e., the ordering of the performance of the algorithms. Reg and Hist both depend on the regularity in the workload pattern in their predictions. For very bursty workloads with a lot of unpredictable spikes, React performs well since the predictions follow the spikes as they occur. Note the lighter blue columns in the graphs are used to indicate that the value for this algorithm is quite high compared to the other values.

However, from the service provider viewpoint, the information about the norm is difficult to interpret, and can only be used for ranking the algorithms. Therefore, other quantities need to be used for a more interesting and informative evaluation. Indeed, the service provider wants to know quantitatively how close to optimal the auto-scaling algorithm is from assigning the ideal amount of resources. This information can be obtained through distances (6), (7), and (8). If these distances are adopted for the solution of the SP (16), the service provider can guarantee what will be the maximum over- and under-provisioning, with a probability of $1-\varepsilon$.

Figure 5 shows the obtained results with these three distances, and for the three case studies. For the sake of presentation, we used $-\hat{\rho}^{\star}$ for the under-provisioning case, indicating that the allocated resources are below $y^{\circ}(\cdot)$. The computed values can thus be used similarly to confidence interval extremes.

Analyzing the obtained results, in the first case study Adapt is able to keep the capacity fairly close to its ideal value, with a maximum bound of 4 servers more, and of 13 server less than actually needed. On the other hand, in the second case study, React is the algorithm that exhibits better performance, with a maximum bound of 51 servers more, and of 9 server less than actually needed. Reg and AGILE have very high values of instantaneous over- and under-provisioning. The third case is similar to the first two cases except for React which again performs relatively much better than the first case. Dashed bars represent distances that are significantly higher than the other methodologies, and for improving the readability of the graph, were left out. For further details on the raw numbers see Table 1. Notice that using PEAS, we can guarantee that these bounds hold with a probability $1-\varepsilon$ with a confidence that is practically 1.

Whereas maximum over- and under-provisioning is of extreme importance, there is however one aspect that must be considered, i.e., how long an auto-scaling technique spends in an over- or under-provisioning state. This aspect is somehow orthogonal to the one discussed above. Indeed, there may be situations in which, in every time instant, the auto-scaling technique is close to the ideal behavior, but this is never

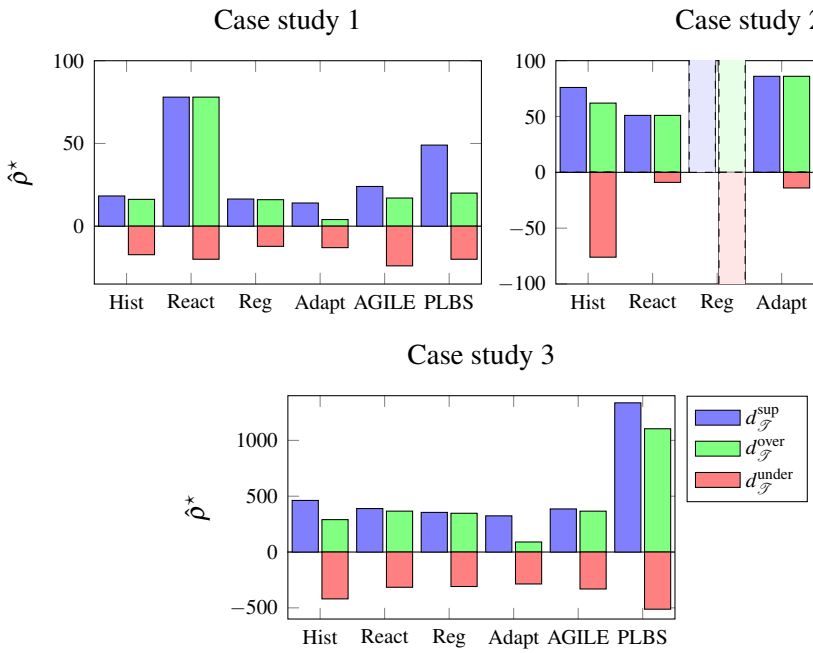


Figure 5: Estimate of the maximum Over-Provisioning (OP) and Under-Provisioning (UP).

reached, resulting in losing revenue (in the case of under-provisioning), or wasted energy and resources (in the case of over-provisioning). On the other hand, a technique may have some high yet short "spikes" in the capacity allocation, but generally being extremely close to the ideal allocation. This last case presents poor performance, for example, with respect to distance (6), while generally behaving in a very nice way.

In order to take into account this aspect, we here consider distances (9), and (10), that measure the average over- and under-provisioning in a time unit. Figure 6 shows the obtained results with the different algorithms. Also in this case we considered $-\hat{\rho}^\star$ for distance (10) analogously to Figure 5. It is possible to see from Figure 6 that Adapt is the algorithm providing better performance in both the case studies. However, if one compares Figures 5 and 6, it is possible to see how AGILE might have bad instantaneous performance but, when time is taken into account, AGILE exhibits better performance in all the case studies.

It is worth noting that the majority of the algorithms tends to over-estimate the needed capacity, rather than under-estimate it. This behavior is desirable from an auto-scaler, since under-estimation could cause large losses for the service provider, especially if the auto-scaler keeps under-estimating for long periods. In this respect,

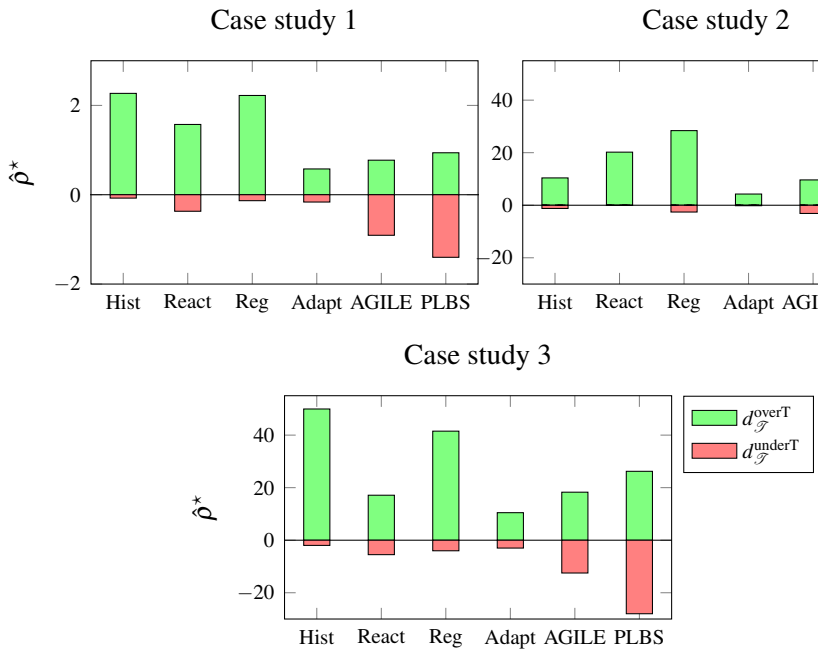


Figure 6: Worst case average Over-Provisioning (OP) and Under-Provisioning (UP) in a time unit.

React and Adapt show the best performance in the second case study, which is also the more realistic one.

As a last metric, we consider the normalized adapted ADI (14). Figure 7 shows the obtained results. As already mentioned, this metric is similar to the norm (4), since it considers the overall performance, accounting both for over- and under-provisioning, and possible oscillations around the desired value. The only qualitative difference is that it considers an interval of values around the real target, it penalizes only when the auto-scaler is outside that interval. We here considered $L = 0.9$ and $U = 1.1$, thus not penalizing auto-scalers that are committing an error below 10%. According to this metric, Adapt is the auto-scaler with the best performance in the first two case studies, while React is the best one in the third case study.

5.5 Discussion

The experimental results suggest that newly published algorithms do not perform particularly better than the older ones. React, which is one of the simplest auto-scalers possible, performs much better than many of the state-of-the-art algorithms

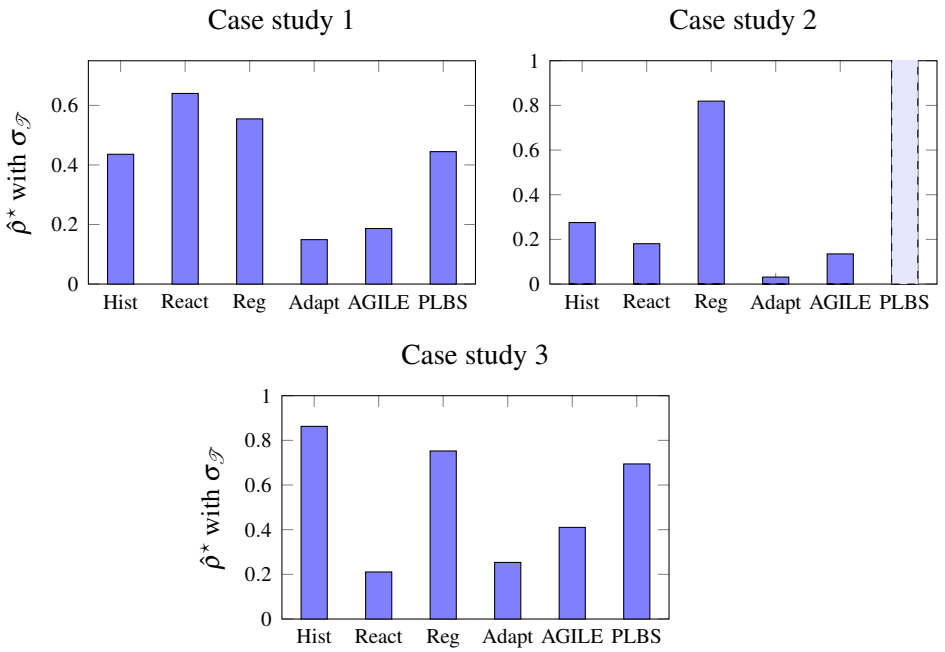


Figure 7: Normalized adapted ADI for the considered auto-scaling strategies.

and outperforms all of them in the second case study. These results hold for all the workloads similar to the ones we use. Since we considered 796 different workloads, each over about six years, the obtained can be generalized, and PEAS allows to quantify the confidence in the obtained performance. In particular, Adapt is the one that is almost invariantly better than the other ones both in the first and in the third case study. On the other hand, in the second case study, React is the auto-scaler showing better performance in terms of under-provisioning. Since this is a relevant aspect from the service provider viewpoint, this data would suggest that React could be a good alternative to Adapt. A summary of the obtained results is presented in Table 1.

While the chosen 6 controllers are not an exhaustive set of the algorithms proposed of the state-of-the-art, they are a representative set. The framework is very flexible in terms of what performance metrics can be integrated and the ease of use to test an algorithm that has not been tested yet. The framework enables the research community to compare the performance of published auto-scalers and the contribution

of newly proposed ones in the future.[3] We believe that this is of great value for testing, quantifying and comparing the performance of auto-scalers.

Table 1: *Summary of the obtained results. The best values are highlighted in bold.*

Case study 1						
$d^{(\cdot)}_{\mathscr{T}}$	**Hist**	**React**	**Reg**	**Adapt**	**AGILE**	**PLBS**
norm	7.82	15.2	10.9	**1.94**	5.78	19.7
sup	18.2	78	16.4	**14**	24	49
under	17.3	20	**12.2**	13	24	20
over	16.2	78	16	**4**	17	20
underT	**0.0757**	0.37	0.133	0.164	0.908	1.4
overT	2.27	1.57	2.22	**0.577**	0.773	0.939
$\sigma_{\mathscr{T}}$	0.436	0.64	0.555	**0.149**	0.186	0.445
Case study 2						
$d^{(\cdot)}_{\mathscr{T}}$	**Hist**	**React**	**Reg**	**Adapt**	**AGILE**	**PLBS**
norm	210	471	1.71×10^4	**32.5**	2.25×10^3	Inf
sup	76	**51**	5.08×10^3	86	2.49×10^3	Inf
under	76	**9**	4.24×10^3	14	2.49×10^3	Inf
over	62	**51**	5.08×10^3	86	514	Inf
underT	1.22	**0.0356**	2.59	0.181	3.13	Inf
overT	10.4	20.2	28.4	**4.26**	9.64	Inf
$\sigma_{\mathscr{T}}$	0.275	0.181	0.819	**0.0312**	0.135	Inf
Case study 3						
$d^{(\cdot)}_{\mathscr{T}}$	**Hist**	**React**	**Reg**	**Adapt**	**AGILE**	**PLBS**
norm	4.09×10^3	1.02×10^3	4.14×10^3	**649**	1.86×10^3	1.38×10^4
sup	462	389	355	**324**	386	1.34×10^3
under	420	316	309	**286**	331	512
over	290	366	347	**90**	366	1.1×10^3
underT	**1.98**	5.5	4.02	2.99	12.5	28
overT	49.9	17.1	41.5	**10.5**	18.3	26.2
$\sigma_{\mathscr{T}}$	0.863	**0.211**	0.752	0.253	0.41	0.694

[3]The code for the framework and for the simulator will be open-sourced.

6 Conclusion and future work

In this paper we propose PEAS, a framework for the evaluation of auto-scaling strategies that is able to provide probabilistic guarantees on the obtainable performance. In particular, we consider three case studies, where we evaluate six different algorithms and test them against 796 distinct real workload traces from projects hosted on the Wikimedia foundations' servers. The considered case studies show the need for a deeper evaluation of the auto-scaling techniques proposed in the literature. The results obtained using PEAS highlighted the problem of generalizing the conclusions of the original published studies. As future work, we are developing a more comprehensive comparative evaluation of auto-scaling strategies using PEAS including more algorithms.

On the other hand, although PEAS was conceived for evaluating auto-scaling strategies, the approach is quite general and can be applied to different resource management problems, especially when stochastic quantities hinder the possibility of providing performance guarantees for the considered methodologies. We envision that, in the near future, the proposed framework can be successfully applied also to a larger class of algorithms, by suitably defining application specific performance measurements and metrics.

APPENDIX

A Parametrization of the simulator

In order to validate and parametrize our simulator, we used C-Mart, a recently published cloud benchmark developed by Turner et al. [61]. The benchmark is publicly available [55]. C-Mart has been designed with cloud performance testing in mind. It utilizes modern web technologies, such as JavaScript, CSS, AJAX, HTML5, SQLite, MongoDB and Memcache DHT, in order to build cloud web applications.

We deployed the benchmark on part of our local 640 cores cluster to emulate a modern two tiered web service running an online auction and shopping website. The backend server runs MySQL while the frontend runs Tomcat application servers. The web application hosted utilizes technologies such as HTML5, AJAX, CSS, rich multimedia, and SQLite. The utilization of these technologies to build C-Mart is the main reason for selecting C-Mart over other widely used benchmarks such as RUBiS and TPC-W.

Both the frontend and backend servers are virtualized KVM images. We used the images provided by the benchmark authors as is, but modified the workload generator to accept the number of users from a trace file while keeping the average number of requests per users bounded. The workload generator run as a standalone application on a bare-metal server with 32 CPU cores and 56 GB of memory. To better investigate the tradeoffs between the average number of requests, the average latencies and the resources allocated to a machine, the database VM also runs on a separate physical server in order to maintain no interference between the database VM and the application VM. The database VM has 8 CPU cores and 10GB of memory, in order to ensure that the database tier is not the bottleneck.

For the frontend VMs, we vary their sizes to obtain the average number of requests a VM is able to serve per time unit. The load is mixed with 24 different pages a client can access with a mixture of images, CSS, video, and text. In the beginning, the size of the front-end machines was kept small, with one CPU and 1GB of memory. This configuration corresponds to a t2.micro Amazon EC2 instance.

We ran several experiments to measure r_{app} (see Section 4.2.1), the average number of requests one VM is capable of serving in an acceptable response time, i.e., below 500ms. This number changed based on the workload mix. For some experiments it was as high as 100 to 150 requests per second per VM, while for other experiments the number was as low as 5 to 10 requests per second per VM. These numbers conform with multiple measurements from online deployed services, for example, Justin.tv backend servers can handle 10 to 20 requests per second per VM on average [30]. Similar numbers were also obtained for the Play framework on EC2 [54].

We ran similar experiments using the MediaWiki server. In these experiments we built a workload generator which generate GET requests for different pages on the Spanish Wikipedia. Since the pages vary in size, we decided to run the experiment for the worst case scenario when all requests are directed only to the largest page on the Spanish Wikipedia, "Alsacia en 1789". We then stress the VMs to find the maximum number of requests that a VM can handle with an end-to-end response time – between a workload generator running on a machine in Lund and the VMs located in Umeå – of 4 seconds or less for the full page to load. The average number of requests per second varied between 20 and 25 requests per seconds in different experiments for a VM with 1 core and 2 GB of RAM. We thus decided to consider the average service rate of a simulated server to follow a Poisson process with an average of $r_{\text{app}} = 22$ requests per time unit per VM. In the simulations, the number of VMs provisioned for a service is then computed using Equation (3).

Table 2: *Description of some of the workloads used for the evaluation.*

Name	key	Language	Project
a	ar.q	Arabic	Wikiquote
b	de	German	Wikipedia
c	en	English	Wikipedia
d	es.mw	Spanish	Wikipedia Mobile
e	fr.d	French	Wiktionary
f	it	Italian	Wikipedia
g	jp	Japanese	Wikipedia
h	nl.q	Dutch	Wikiquote
i	pl	Polish	Wikipedia
j	pt.d	Portuguese	Wictionary
k	ru.mw	Russian	Wikipedia Mobile
l	sv.d	Swedish	Wiktionary
m	uz	Uzbek	Wikipedia
n	zh	Chinese	Wikipedia
o	zu.b	Zulu	Wikibooks

B Workload examples

Figure 8 shows some examples of workload streams from the ones used for the evaluation. The workloads are very diverse in nature, in evolution, in burstiness, and in magnitude. Almost all of the traces in the dataset show some form of diurnal pattern where the number of requests increase during some hours of the day. Since these workloads contain very regional workloads, e.g., the requests on the Vietnamese Wikipedia, the workloads also differ on when these diurnal patterns occur. The diversity in the workloads shows the importance of evaluating the performance of an auto-scaler using many different workloads. It also strengthen our argument for an evaluation framework that can provide probabilistic guarantees on the evaluation number. Table 2 gives more information about the workloads shown in Figure 8.

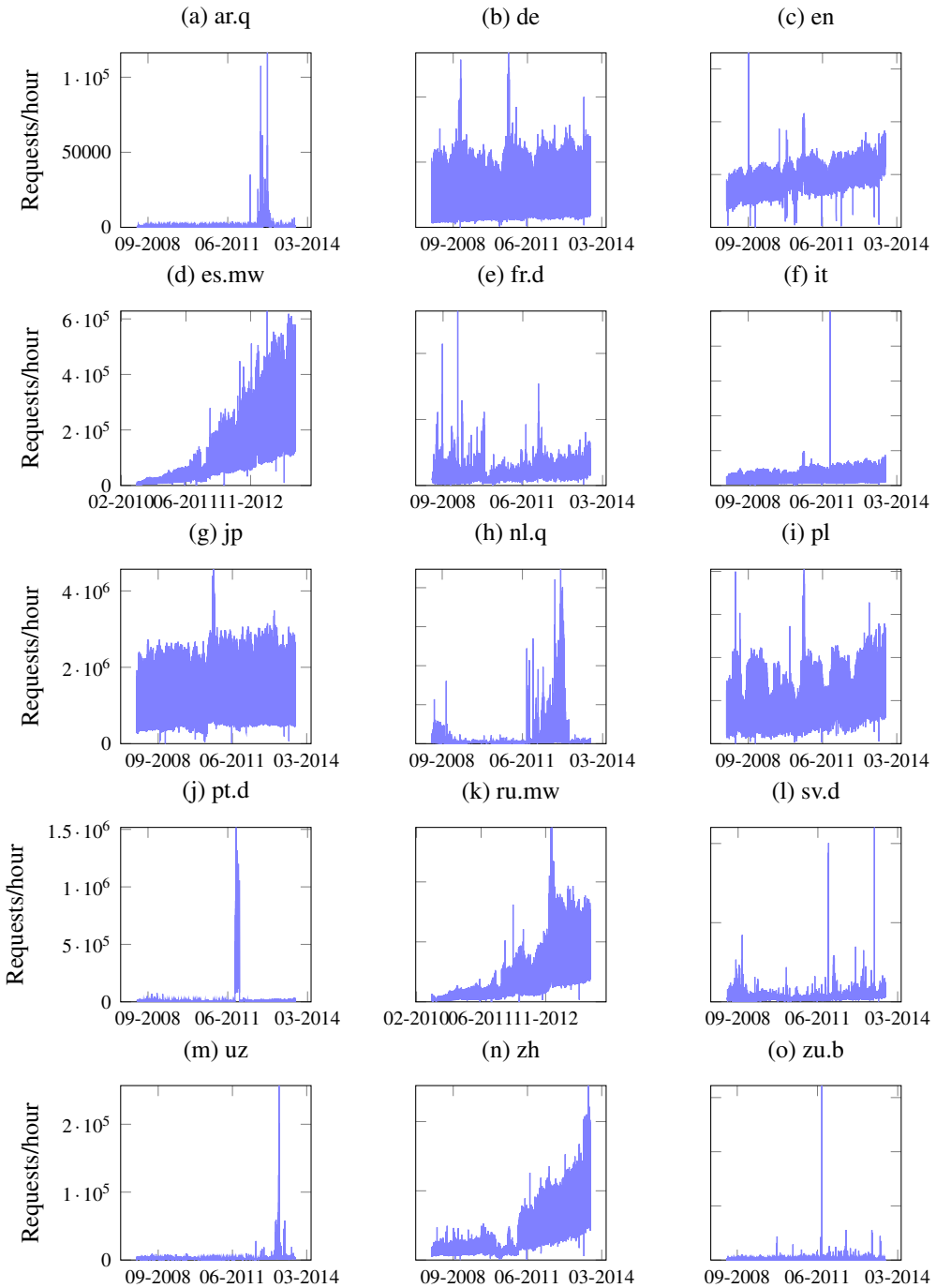


Figure 8: Some of the workloads used for the experiments

C Correlation analysis of the workloads

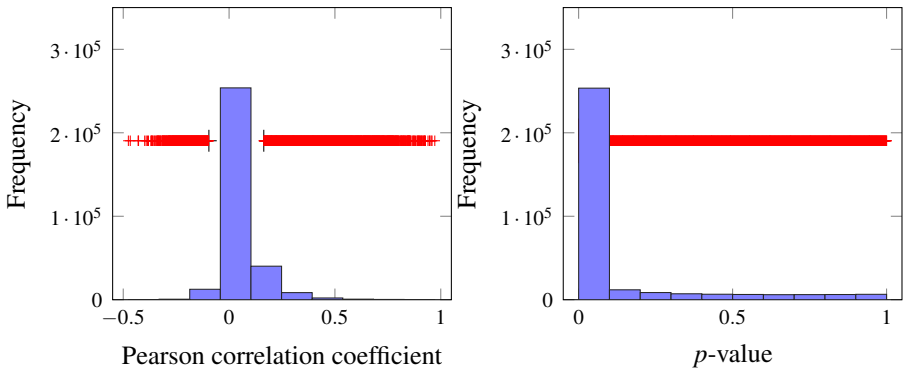


Figure 9: Distribution of the Pearson correlation coefficients (on the left), and of the p-values (on the right) for the correlation analysis.

To provide a better understanding of how the workload streams used in this study are related, we performed a correlation analysis on the 796 workload streams used. We calculated the Pearson correlation coefficient between all possible pairs of workload streams, e.g., (jp, pl), (de, en) , (de, es) and so on. Figure 10 shows how the correlation coefficients, and the corresponding p-values are distributed. Each of the figures shows the distribution of correlations as a histogram and on top of the histogram a box-plot representing the spread of the obtained values. The figures indicate that the correlation coefficients are mostly concentrated close to 0, which is highlighted also by the box-plots. The p-values confirm that there is not statistical evidence of correlation between the different workloads, since they are mostly concentrated close to zero: The 75th percentile is 0.034. When analyzing the correlations more in detail, there are surprising results. For example, The load on the German Wikipedia is highly correlated with the load on the Spanish Wikipedia (Correlation coefficient greater than 0.7) while both are not correlated with almost any other load on any European language Wikipedia such as the load on the French Wikipedia, the load on the Italian Wikipeda, the load on the Norwegian Wikipedia, the load on the Portuguese Wikipedia, or the load on the Swedish Wikipedia.[4]

[4]The whole dataset related to the correlation analysis of the workloads will be made publicly available, in case this manuscript is accepted.

D Analysis of startup time and shutdown time

To evaluate the statistical characteristics of the startup and shutdown time of VMs, we have made an experiment where we installed MediaWiki [8] on an Ubuntu Linux server VM. The VM was assigned 4 cores and 10 GB of RAM on an HP Elitedesk G1 SFF machine with a quad-core hyper-threaded Intel Core I7 processor. The VM is controlled using Vagrant [52] running on Virtualbox. The VM is started and stopped 200 times and the VM startup and shutdown times are recorded from the time the startup command is issued till the MediaWiki application is responsive. To make the experiments realistic, the physical machine hosting the MediaWiki VM has been injected with varying background load. The frequency of the measured startup

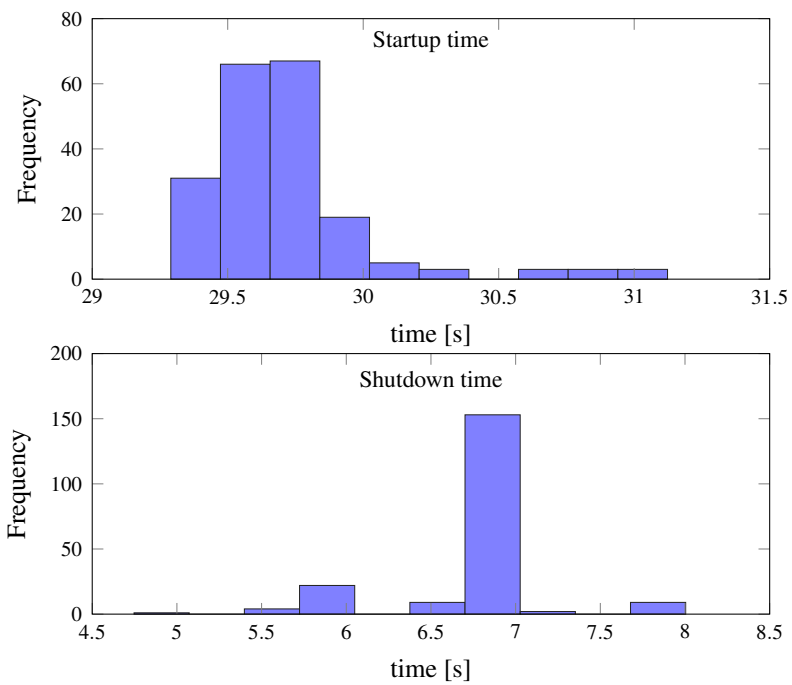


Figure 10: Histograms for startup and shutdown time.

and shutdown times are shown in Figure 10. The average and standard deviation for the measured startup times are 29.712s, and 0.310s respectively. The average and standard deviation for the measured shutdown times are 6.721s, and 0.455s respectively. By inspecting Figure 10, neither the startup nor the shutdown times appear to be normally distributed. In both cases performing a normality test rejected the hypothesis that the data is normally distributed. Figure 11 shows the empirical

Cumulative Distribution Function (CDF) and the standard normal CDF for the two quantities, indicating also in the top right corner the p-value of the test. Note that the x axis indicates the normalized time, i.e., the average have been subtracted to the data, and then divided by the standard deviation. We tried to fit the data to four other distributions, namely, power law, stretched exponential, exponential, and log-normal, but none of the tested probability distributions is able to describe these data. For the purpose of this work, the identification of the right probabilistic distribution is not relevant, since both the startup and shutdown times are below the time scale in which the auto-scaler takes its decisions. However, a probabilistic characterization of the startup and shutdown times deserves deeper investigation, which is left to future work.

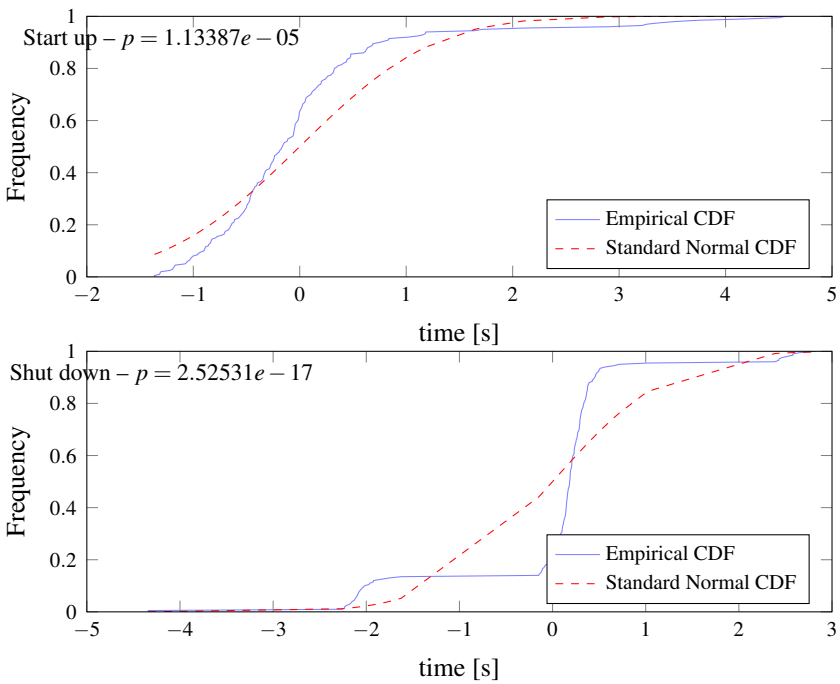


Figure 11: Empirical cumulative distribution functions versus standard normal distribution.

References

[1] A. Abate and M. Prandini. Approximate abstractions of stochastic systems: a randomized method. In *Proc. 50th IEEE Conf. on Decision and Control and*

European Control Conf. (CDC-ECC), pages 4861–4866. IEEE, Dec 2011.

[2] A. Al-Shishtawy and V. Vlassov. ElastMan: Autonomic elasticity manager for cloud-based key-value stores. In *Proc. 22nd Int. Symposium on High-performance Parallel and Distributed Computing*, HPDC 13, pages 115–116, New York, NY, USA, 2013. ACM.

[3] T. Alamo, R. Tempo, and A. Luque. On the sample complexity of probabilistic analysis and design methods. In J. Willems, S. Hara, Y. Ohta, and H. Fujioka, editors, *Perspectives in Mathematical System Theory, Control, and Signal Processing*, volume 398 of *Lecture Notes in Control and Information Sciences*, pages 39–50. Springer Berlin Heidelberg, 2010.

[4] Alexa. The top 500 sites on the web., 2015. `http://www.alexa.com/topsites`[Online; accessed 2015-04-09].

[5] A. Ali-Eldin, A. Rezaie, A. Mehta, S. Razroev, S. Sjöstedt-de Luna, O. Seleznjev, J. Tordsson, and E. Elmroth. How will your workload look like in 6 years? analyzing wikimedia's workload. In *Proc. IEEE Int. Conf. on Cloud Engineering*, IC2E 14, pages 349–354, Washington, DC, USA, 2014. IEEE Computer Society.

[6] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *IEEE Network Operations and Management Symposium*, NOMS 12, pages 204–212, April 2012.

[7] F. Almeida Morais, F. Vilar Brasileiro, R. Vigolvino Lopes, R. Araujo Santos, W. Satterfield, and L. Rosa. Autoflex: Service agnostic auto-scaling framework for IaaS deployment models. In *Proc. 13th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing*, CCGrid 13, pages 42–49, 2013.

[8] D. J. Barrett. *MediaWiki (Wikipedia and Beyond).* " O'Reilly Media, Inc.", 2008.

[9] S. Blagodurov, D. Gmach, M. Arlitt, Y. Chen, C. Hyser, and A. Fedorova. Maximizing server utilization while meeting critical slas via weight-based collocation management. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 277–285. IEEE, 2013.

[10] P. Bodik. *Automating Datacenter Operations Using Machine Learning*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2010.

[11] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proc. 1st ACM Symposium on Cloud Computing*, SoCC 10, pages 241–252, New York, NY, USA, 2010. ACM.

[12] S. D. Bopardikar, A. Borri, J. a. P. Hespanha, M. Prandini, and M. D. Di Benedetto. Randomized sampling for large zero-sum games. *Automatica*, 49(5):1184–1194, 2013.

[13] E. S. Buneci and D. A. Reed. Analysis of application heartbeats: Learning structural and temporal features in time series data for identification of performance problems. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 52. IEEE Press, 2008.

[14] G. Calafiore and M. Campi. Uncertain convex programs: randomized solutions and confidence levels. *Mathematical Programming*, 102(1):25–46, 2005.

[15] G. Calafiore and M. Campi. The scenario approach to robust control design. *IEEE Trans. on Automatic Control*, 51(5):742–753, May 2006.

[16] M. Campi and S. Garatti. A sampling-and-discarding approach to chance-constrained optimization: Feasibility and optimality. *Journal of Optimization Theory and Applications*, 148(2):257–280, 2011.

[17] M. C. Campi, S. Garatti, and M. Prandini. The scenario approach for systems and control design. *Annual Reviews in Control*, 33(2):149–157, 2009.

[18] E. Casalicchio and L. Silvestri. Mechanisms for sla provisioning in cloud-based service providers. *Computer Networks*, 57(3):795–810, 2013.

[19] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev.*, 35(5):103–116, Oct. 2001.

[20] T. Chieu, A. Mohindra, A. Karve, and A. Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *IEEE Int. Conf. on e-Business Engineering*, ICEBE 09, pages 281–286, Oct 2009.

[21] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[22] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[23] M. A. Dong and R. K. Treiber. Dynamic resource pool expansion and contraction in multiprocessing environments, Mar. 3 1992. US Patent 5,093,912.

[24] D. G. Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2014.

[25] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. In *IEEE Int. Conf. on Cloud Engineering*, IC2E 14, Boston, MA, United States, 2014.

[26] F. Ferraris, D. Franceschelli, M. Gioiosa, D. Lucia, D. Ardagna, E. Di Nitto, and T. Sharif. Evaluating the auto scaling performance of Flexiscale and Amazon EC2 clouds. In *Proc. 14th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC 12, pages 423–429, Sept 2012.

[27] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. AutoScale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, Nov. 2012.

[28] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proc. Int. Conf. on Network and Service Management*, CNSM 10, pages 9–16, Oct 2010.

[29] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *Proc. 4th ACM/SPEC Int. Conf. on Performance Engineering*, ICPE 13, pages 187–198, New York, NY, USA, 2013. ACM.

[30] T. Hoff. Justin.tv's live video broadcasting architecture, 2010. `http://highscalability.com/blog/2010/3/16/justintvs-live-video-broadcasting-architecture.html`[Online; accessed 2014-11-24].

[31] A. Iosup. IaaS cloud benchmarking: Approaches, challenges, and experience. In *Proc. 5th IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS 12, pages 1–8, New York, NY, USA, 2012. ACM.

[32] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Gener. Comput. Syst.*, 27(6):871–879, 2011.

[33] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1):155–162, 2012.

[34] E. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Min. Knowl. Discov.*, 7(4):349–371, Oct. 2003.

[35] H. Khazaei, J. Misic, and V. Misic. Performance analysis of cloud computing centers using m/g/m/m+r queuing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 23(5):936–943, May 2012.

[36] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz. Napsac: Design and implementation of a power-proportional web cluster. *ACM SIGCOMM computer communication review*, 41(1):102–108, 2011.

[37] P. Lama and X. Zhou. Efficient server provisioning with control for end-to-end response time guarantee on multitier clusters. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):78–86, Jan 2012.

[38] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proc. 10th ACM SIGCOMM Conf. on Internet Measurement*, IMC 10, pages 1–14, New York, NY, USA, 2010. ACM.

[39] H. Li and T. Yang. Queues with a variable number of servers. *European Journal of Operational Research*, 124(3):615–628, 2000.

[40] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, pages 1–10. ACM, 2010.

[41] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh. Automated control in cloud computing: Challenges and opportunities. In *Proc. 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC 09, pages 13–18, New York, NY, USA, 2009. ACM.

[42] H. T. Liu and J. A. Silvester. Dynamic resource allocation scheme for distributed heterogeneous computer systems, July 9 1991. US Patent 5,031,089.

[43] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, pages 1–34, 2014.

[44] A. H. Mahmud, Y. He, and S. Ren. Bats: Budget-constrained autoscaling for cloud performance optimization. *SIGMETRICS Perform. Eval. Rev.*, 42(1):563–564, June 2014.

[45] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 131–140. ACM, 2011.

[46] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Proc. 11th IEEE/ACM Int. Conf. on Grid Computing*, GRID 10, pages 41–48, Oct 2010.

[47] S. Meng, L. Liu, and V. Soundararajan. Tide: achieving self-scaling in virtualized datacenter management middleware. In *Proc. 11th Int. Middleware Conf. Industrial Track*, Middleware Industrial Track 10, pages 17–22, New York, NY, USA, 2010. ACM.

[48] D. Mituzas. Page view statistics for wikimedia projects, 2007. `http://dumps.wikimedia.org/other/pagecounts-raw/`[Online; accessed 2014-11-20].

[49] A. Nemirovski and A. Shapiro. Scenario approximations of chance constraints. In G. Calafiore and F. Dabbene, editors, *Probabilistic and Randomized Methods for Design under Uncertainty*, pages 3–47. Springer London, 2006.

[50] M. A. Netto, C. Cardonha, R. L. Cunha, and M. D. Assunção. Evaluating auto-scaling strategies for cloud computing environments. In *Proc. IEEE 21st Int. Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, MASCOTS 14, pages 1–10, 2014.

[51] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic distributed resource scaling for Infrastructure-as-a-Service. In *Proc. 10th Int. Conf. on Autonomic Computing*, ICAC 13, pages 69–82, San Jose, CA, 2013. USENIX.

[52] J. Palat. Introducing vagrant. *Linux Journal*, 2012(220):2, 2012.

[53] A. V. Papadopoulos and M. Prandini. Model reduction of switched affine systems: A method based on balanced truncation and randomized optimization. In *Proc. 17th Int. Conf. on Hybrid Systems: Computation and Control*, HSCC 14, pages 113–122, New York, NY, USA, 2014. ACM.

[54] C. Papauschek. Real-world performance of the play framework on ec2, 2013. `http://blog.papauschek.com/2013/04/real-world-performance-of-the-play-framework-on-ec2/`[Online; accessed 2014-11-24].

[55] J. Payne. C-mart:benchmarking the cloud, 2014. `http://theone.ece.cmu.edu/cmart/`[Online; accessed 2014-11-20].

[56] A. Prékopa. Probabilistic programming. In A. Ruszczyǹski and A. Shapiro, editors, *Stochastic Programming*, volume 10 of *Handbooks in Operations Research and Management Science*, pages 267–351, London, UK, 2003. Elsevier.

[57] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[58] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. 2011 IEEE 4th Int. Conf. on Cloud Computing*, CLOUD 11, pages 500–507, Washington, DC, USA, 2011. IEEE Computer Society.

[59] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. 7th Int. Conf. on Autonomic Computing*, ICAC 10, pages 21–30, New York, NY, USA, 2010. ACM.

[60] R. Sturm, W. Morris, and M. Jander. *Foundations of Service Level Management*. Sams Professionals. SAMS, 2000.

[61] A. Turner, A. Fox, J. Payne, and H. S. Kim. C-mart: Benchmarking the cloud. *IEEE Trans. on Parallel and Distributed Systems*, 24(6):1256–1266, 2013.

[62] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Proc. 2nd Int. Conf. on Autonomic Computing*, ICAC 05, pages 217–228, 2005.

[63] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1:1–1:39, 2008.

[64] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: operating system services for wide area applications. In *Proc. 7th Int. Symposium on High Performance Distributed Computing*, pages 52–63, Jul 1998.

[65] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup. An analysis of provisioning and allocation policies for Infrastructure-as-a-Service clouds. In *Proc. 12th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing*, CCGRID 12, pages 612–619, Washington, DC, USA, 2012. IEEE Computer Society.

[66] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 31–40. IEEE, 2013.

[67] J. Wilkes. More Google cluster data, 2011. http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html[Online; accessed 2014-10-30].

Paper VII

WAC: A Workload Analysis and Classification Tool for Automatic Selection of Cloud Auto-scaling Methods.

A. Ali-Eldin, J. Tordsson, E. Elmroth, and M. Kihl

To be Submitted for Journal publication

WAC: A Workload Analysis and Classification Tool for Automatic Selection of Cloud Auto-scaling Methods

Ahmed Ali-Eldin*, Johan Tordsson*, Erik Elmroth*, and Maria Kihl†

Abstract

Autoscaling algorithms for elastic cloud infrastructures dynamically change the amount of resources allocated to a service according to the current and predicted future load. Since there are no perfect predictors, no single autoscaling algorithm is suitable for accurate predictions of all workloads. To improve the quality of workload predictions and increase the Quality-of-Service (QoS) guarantees of a cloud service, multiple autoscalers suitable for different workload classes need to be used. In this work, we introduce WAC, a Workload Analysis and Classification tool that assigns workloads to the most suitable elasticity autoscaler out of a set of pre-deployed autoscalers. The workload assignment is based on the workload characteristics and a set of user-defined Business-Level-Objectives (BLO). We describe the tool design and its main components. We implement WAC and evaluate its precision using various workloads, BLO combinations, and state-of-the-art autoscalers. Our experiments show that, when the classifier is tuned carefully, WAC assigns between 87% and 98.3% of the workloads to the most suitable elasticity autoscaler.

1 Introduction

Elasticity can be defined as the ability of a cloud infrastructure to dynamically change the amount of resources allocated to a running application [6]. Resources should be allocated according to the changing load in order to preserve QoS requirements at a reduced cost. Typically, a variety of different applications with different workloads

*Department of Computing Science, Umeå University, Sweden, email: {ahmeda, tordsson, elmroth}@cs.umu.se

†Department of Electrical and Information Technology, Lund University, Sweden, email: {Maria.Kihl}@eit.lth.se

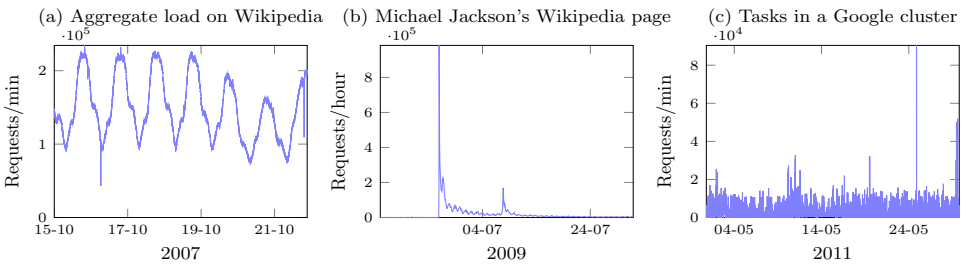


Figure 1: Different workloads have different burstiness.

and different user behaviors run in a cloud [1]. Some workloads have repetitive patterns, e.g., the Wikipedia workload shown in Figure 1(a) has diurnal patterns. Some uncorrelated spikes and bursts can occur in a workload due to an unusual event [4, 34], e.g., Figure 1(b) shows the load on Michael Jackson's Wikipedia page before and after his death with two visible significant spikes that occurred after his deaths and during his memorial service. On the other hand, some workloads have some weak patterns or no patterns at all, e.g., the workload shown in Figure 1(c) for tasks submitted to a Google cluster [53]. Most often, the characteristics of the workloads of new applications are unknown in advance.

There are tradeoffs between using different auto-scalers in terms of over-provisioning, under-provisioning, the oscillations in resource allocation and the time and resources required to compute the required capacity. To understand these tradeoffs, we implemented four autoscalers proposed in the literature. Figure 2 shows the output of the four implemented controllers, [6] in blue, [15] in red, [26] in green, and [51] in cyan, when used to predict the number of servers required to serve the load on the English Wikipedia pages between October 19th, 2010 and October 28th, 2010. The figure also shows the theoretical optimal number of servers required to serve such a workload (shown as the black line).[1] During this period, the load was unstable with huge bursty period. The first three peaks of the load are part of the bursty period. The rest is when the load started to settle down. One can see that during this transient period, the four auto-scalers show very different behavior and tradeoffs.

Since there are no perfect controllers [38], designing an autoscaler that produces accurate predictions for all possible datacenter workloads and applications is infeasible [7]. Elasticity autoscalers' performance will always vary with different workloads and changing system dynamics. Thus, workload characterization is an important first step towards choosing the most appropriate elasticity autoscaler for a workload.

[1]To calculate the theoretical desired capacity we used an approach similar to the approach described in Agile [21] to calculate the number of required servers to serve the requests in the workload

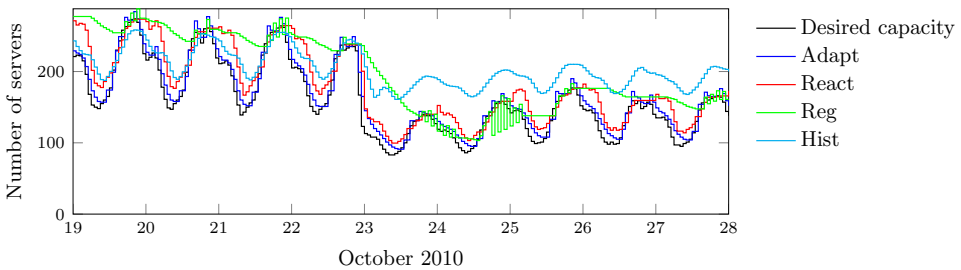


Figure 2: There are tradeoffs when choosing which auto-scaler to use in order to control server provisioning.

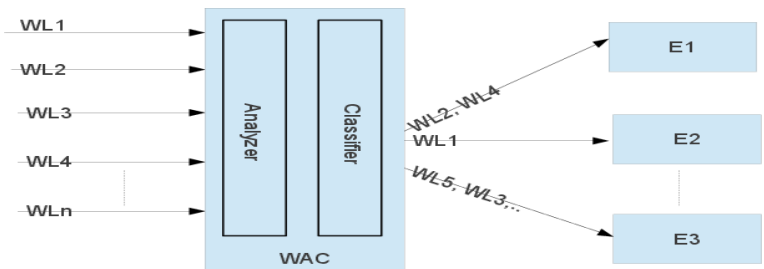


Figure 3: WAC: A Workload Analysis and Classification Tool.

An autoscaler can cause instability and extreme oscillations in resource allocations if the system dynamics change leading to wrong capacity provisioning decisions [38]. There is a cost if an autoscaler adds more resources to a service than required by the actual load as these extra resources are underutilized and wasted. Similarly, if the resources provisioned for a service are not sufficient for its workload, the service performance deteriorates and parts of the workload may get dropped. The service can become unresponsive or crash.

This work presents *WAC, a Workload Analysis and Classification tool* for cloud infrastructures. The tool analyzes the history of the running/new workloads and extracts some characteristics. Workloads are then classified and assigned to the most suitable available elasticity autoscaler based on the extracted characteristics, reducing the risk of bad predictions and improving the provided QoS. Figure 3 shows the two main components of WAC; the analyzer and the classifier. The analyzer extracts and quantifies two characteristics from the workload traces, *periodicity and burstiness*. Periodicity describes the similarity between the value of a workload at any point in time to the workload's history. Burstiness measures load spikes in the system, when a running service suddenly experiences huge increases in the number of requests. The classifier uses these two characteristics and a set of user-defined BLOs to classify the

workloads and assigns each workload to a suitable elasticity autoscaler. We discuss our choice of the two properties and the methods used to quantify them in more details in Section 2.

Classification is a form of supervised learning used to predict the class of an input data object based on a group of features [18]. A data object can be anything from a network request to a data stream. For WAC, the data object is a workload, the classification features are the workload's periodicity, burstiness and the predefined BLOs. The classes are the different elasticity autoscalers. Supervised learning algorithms require training using labeled training data, i.e., objects with known classes. The training data should be sufficient to enable the classifier to generalize to future inputs that arrive after training, i.e., new applications and workloads which will start running on the infrastructure in the future. To construct our training set, we collect and analyze three sets of workloads. The first set contains 14 real workloads from a diverse set of real applications. The second set consists of almost six years of workload traces from Wikimedia foundation with 798 (independent) workload streams for different Wikimedia projects in different languages. For the third set, we generate 55 synthetic workloads that have different characteristics compared to the real workloads. Workload analysis and generation are discussed in Section 3 and Appendix B.

Each class maps to an available elasticity autoscaler implemented and provided by the cloud provider. This enables a cloud provider or a cloud user to choose the right autoscaler for a certain workload reducing the aggregate prediction errors and allowing an improvement of the QoS. Many designs for elasticity autoscalers are suggested in the literature using different approaches such as control theory [36], neural networks [28], second order regression [26], histograms [51] and time-series models [7, 22]. Section 4 discusses our selected and implemented autoscalers to use with WAC.

Plenty of classifiers have been suggested in the machine learning literature, out of these Support Vector Machines (SVMs) and k-Nearest-Neighbors (kNN) are very popular [18]. kNN is chosen as the WAC classifier algorithm. We discuss our choice of kNN over SVMs, the training of the classifier, kNN accuracy and discuss our results in Section 5.

2 Workload Characterization Techniques

2.1 Why Periodicity and Burstiness?

Various characteristics can be used to characterize cloud workloads [17]. With respect to elasticity, the main characteristics are those that affect predictions of the future values of a workload accurately in order to provision enough resources to maintain

acceptable QoS. The predictability of a workload is dependent on how much its future is correlated with its past. For both clusters and grids, Li [35] shows that statistical measures based on inter-arrivals are of limited usefulness for finding such correlations and that count based measures, e.g. arrival rates, should be used. He identifies three main classes for workloads which correspond to workloads with periodicity, workloads with long-range dependence but not necessarily periodic, and non-periodic workloads. These findings are not special for grids since most workload predictors base predictions on either the short-term or the long-term dependence of the load.

Another aspect that affects the quality of predictions for a workload is the burstiness of the workload. A burst can cause severe system performance issues [11]. From a workload prediction point of view, some of these events are completely unpredictable with respect to both their time of occurrence and their magnitude, e.g., the death of Michael Jackson. Such events require a robust autoscaler that can handle substantial changes in the system dynamics. Other spikes are predictable with respect to their time of occurrence but not in magnitude, e.g, during the FIFA 1998 World Cup, spikes occurred around the times the games were played.

While there might be other useful features for workload classification, We choose periodicity and burstiness as the two main characteristics to classify a workload as they can be calculated for any workload. From our experience, they both have the ability to capture most of the required characteristics for designing an elasticity autoscaler and are thus enough to build a proof-of-concept prototype of the tool without having to deal with the curse of dimensionality [23].

2.2 Measuring Periodicity

A common method to measure periodicity is to use autocorrelation. Autocorrelation is the cross-correlation of a signal with a time-shifted version of itself [19]. A perfectly random signal has an autocorrelation of 0 at all time lags. An autocorrelation of 1 at a time lag of τ means that the signal strength is equal at this time lag while an autocorrelation near -1 means that the signal has an opposite direction of influence. The autocorrelation function (ACF) describes the autocorrelation as a function of τ. We use the average ACF as a measure of workload periodicity. While there are other methods for measuring periodicity such as the Fast-Fourier Transform (FFT), we chose the ACF because it is a time-domain method. FFT on the other hand converts the signal to the frequency domain and thus lose all time related information.

2.3 Measuring Burstiness

There are many proposed methods to measure burstiness, none of them prevalent though [5, 37]. We propose the use of Sample Entropy (SampEn) as a measure of

burstiness [46]. Sample Entropy is defined as *"the negative natural logarithm of the (empirical) conditional probability that sequences of length m similar point-wise within a tolerance r are also similar at the next point"*. If SampEn is equal to 0 then the workload has no bursts. The higher the value for SampEn, the more bursty the workload is. It has been used successfully to classify abnormal physiological signals for over a decade [46]. There are two main advantages of SampEn over other methods in the literature, i) SampEn can operate on short data sequences and, ii) it takes into account gradual workload increases and periodic bursts.

One main limitation of using SampEn is that it is computationally expensive to calculate both CPU-wise and memory-wise. The computational complexity (in both time and memory) of SampEn is $O(n^2)$ where n is the number of points in the trace. In addition, workload characteristics can change during operation. If workload burstiness is characterized for the full history of the workload, recent changes in the workload dynamics become harder to characterize. To overcome these limitations, we have modified the original SampEn algorithm to make it better suited for cloud workloads [5]. In the modified algorithm, *AvgSampEn*, the trace is divided into smaller equal sub-traces each with a length of one day. SampEn is then calculated for each sub-trace. A moving average is then calculated for the obtained values for the subtraces, giving higher weight to more recent SampEn values. By reducing n, the speed of the algorithm is significantly improved. [2] Appendix A discusses the implementation of the modified algorithm, *AvgSampEn*.

3 Workloads

Typically, cloud providers such as Amazon EC2 host many services with different workloads. The hosted services use anything from a couple of virtual machines to a few thousand machines [1]. With the exception of a recently released workload trace from a production cluster at Google [53], no cloud provider provides publicly available cloud workloads [11]. On the other hand, there are a few publicly available workloads for various types of Internet applications. We use some of these workloads in addition to the Google cluster workload. We believe that these workloads are representative for many of the workloads running on typical clouds since they include major web sites and core Internet technologies.

Recently, cloud providers began to host high performance computing applications [44]. Many similar workloads are publicly available of which we use the DAS-2 system workload, the Grid-5000 workload, NorduGrid traces, SHARCNET traces, Large hadron collider Computing Grid (LCG) traces and the AuverGrid traces, avail-

[2] We have also shown in [5] that the modified version of SampEn, *AvgSampEn*, is stable over a wide choice of parameters. Due to space constraints, we omit the more detailed explanation about *AvgSampEn* since it is out of our current scope. The interested reader is kindly referred to [5].

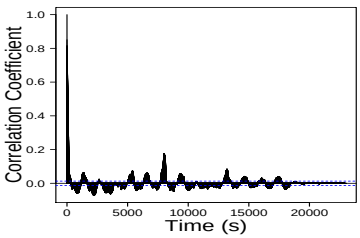
able from the grid workload archive [25]. Caching services are another type of applications running on the cloud [1]. We use traces from the IRCache [27] project as representative workloads for this class of applications. The IRCache traces maintain traces from different caching service sites. We only use traces from 4 sites. In addition, we use a one month trace of PubMed server usage [39], traces from the FIFA 1998 World Cup [8], and traces from Wikipedia on a minutes time scale [4].

In addition to the above workloads, the Wikimedia foundation [10] – which hosts Wikipedia, the sixth most popular website on the Internet [3] – provides publicly available workload traces that logs the per page aggregate number of requests per hour for all hosted projects [2]. These projects include, for example, Wikipedia, Wikibooks and Wikitionary in various languages. This workload is interesting since the load can be separated into independent streams with each stream representing a project and a language, e.g., one stream can have the requests to the English Wikipedia pages, another can be for the Swedish Wikitionary project, and a third can be for the French Wikibooks project, and so on. Many of these project streams can be even divided further into, for example, load from Mobile users and load from non-mobile. We use the Wikimedia hourly traces logged between the 9^{th} of December, 2007 and the 16^{th} of October, 2013 and extract 798 workload streams to be used to validate our tool. While there are more workload streams that we extracted, we choose workloads with larger dynamics since many of the extracted streams have very few requests, e.g., the Zulu language Wikipedia. We have performed a correlation analysis on the selected workloads, and we found that they are practically not correlated.

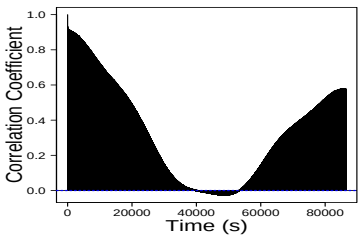
In addition to the real workloads, we have generated 55 synthetic workloads using a mixture of sinusoidal functions and probability distributions. Appendix B discusses some of the details of the the workload generation process used.

3.1 Periodicity of the Real Workloads

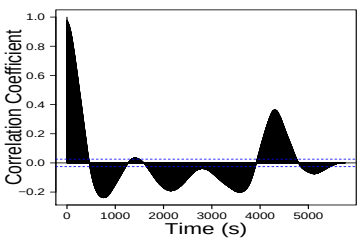
The coefficient of determination (R^2) is defined as the squared value of the correlation coefficient at a certain lag. It measures the proportion of variance in a dependent variable that can be explained by the independent variable, i.e., the dependence of the future data on the historical data of the workload [47]. Figure 4 shows the correlograms of some of the analyzed workloads. Figure 4(a) shows high ACF for short lags in the PubMed traces meaning that the value of the workload depends on its value in the near past. Figures 4(c), 4(d), 4(e) and 4(b) show high autocorrelation coefficients meaning that their corresponding workloads have strong periodicity. Figure 4(f) shows a workload that is almost random. The second column in Table 1 describes the ACFs for some of the workloads qualitatively. To obtain a single number representing the periodicity of the workload instead of a vector of correlation coefficients that can bias the classifier, we use the average correlation coefficients per



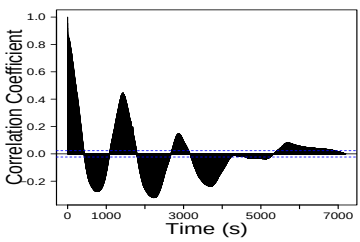
(a) IRCache service runnings at Boulder, Colorado (Bo).



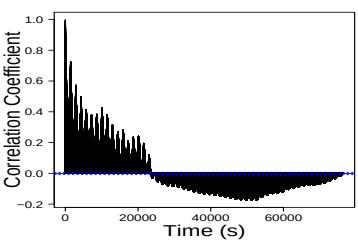
(b) PubMed access traces.



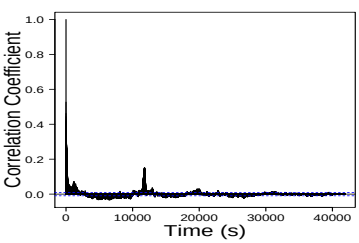
(c) IRCache service running at San Diego, California (SD).



(d) IRCache service running at Urbana-Champaign, Illinois (UC).



(e) The FIFA 1998 world cup server workload.



(f) A workload from a Google cluster.

Figure 4: Correlograms of non-grid workloads.

Table 1: *Workload analysis results.*

Workload	ACF	SampEn
Bo-IRCache	Very low	0.0
SV-IRCache	Very low	0.0017
SD-IRCache	High for small lags. Medium for large lags	1.062
UC-IRCache	High for small lags. Medium for large lags	0.0
DAS	Random signal	0.0
Grid5000	Random signal	236.16
LCG	Random signal	134.0
LPC	Random signal	2.827
NorduGrid	Random signal	8.43
SharcNet	Random signal	2.803
FIFA	High for small lags. Medium for large lags	0.0
Google	Very low	235.70
PubMed	High	0.0
Wikipedia	High	0.0014

workload for classification. This approach is similar to the approach taken by Rabiner and Wilpon [42].

3.2 Burstiness of the Real Workloads

There are two parameters needed to calculate $AvgSampEn$. The first parameter is the pattern length m, which is the size of the window in which the algorithm searches for repetitive bursty patterns. The second parameter is the deviation tolerance r which is the maximum increase in load between two consecutive time units before considering this increase as a burst. The pattern length m is set to 1 hour. The deviation tolerance r is set to 30% of the 70^{th} percentile of the workload values or to double the average number of requests served by a server per unit time for extremely low traffic, since 30% of the maximum workload can be less than the load handled by one server. The third column in Table 1 shows the average SampEn values computed for the different workloads. From the table we can see that workloads vary greatly in the degree of burstiness they have. With the exception of the DAS workload, workloads with ACFs around zero have significant burstiness as SampEn is greater than 1 for all of them, while the ones with high or medium ACFs do not show significant burstiness.

4 On the performance of different autoscalers

4.1 Chosen autoscalers

Since different autoscalers exhibit varying performance with different workloads, a number of autoscalers have to be selected for implementation. These autoscalers are the classes to which WAC assigns the workloads. For this work, we choose autoscalers that can be used on a wide range of scenarios with no human tuning. We implement four state-of-the-art autoscalers that fall in this criteria. In addition, we acquire the source codes of two other state-of-the-art autoscalers from their designer. The four implemented autoscalers are:

1. The first autoscaler implemented is the step controller as described by Chieu et. al. [15] (hereafter called *React*). It is a reactive autoscaler where the capacity provisioned follows the load changes when they happen with no predictions of future capacity. If the load is constant, resources are kept constant. If the load increases by $\Delta Load$, the autoscaler reacts to the load increase by provisioning S servers that can handle this load increase. Similarly if the load decreases by $\Delta Load$ which is less than a certain threshold, the autoscaler reacts by removing the extra machines. As the autoscaler computes the required capacity every time the load changes, it never allocates more resources than required as resources are only allocated when the load increases and are removed once the load decreases below a certain threshold.

2. The second implemented autoscaler is a histogram based controller as described by Urgaonkar et. al. [51] (henceforth called *Hist*). This autoscaler builds histograms for the workload history observed within a defined time interval (e.g. between 9:00 and 10:00 on Monday). The autoscaler estimates the peak demand of the workload to occur within the defined time interval before that interval begins, as follows. For each period there is a corresponding histogram that can be used to find the probability distribution of the arrival rate for that period. The autoscaler can predict the capacity required based on a higher percentile of the distribution tail, e.g., the 99^{th} percentile. This design allows the autoscaler to capture seasonal patterns such as diurnal or weekly patterns. The period length chosen for the histogram is one hour. The autoscaler has a correction mechanism in case the provisioned capacity is less than the actual required capacity. When the service load becomes more than the provisioned capacity can handle, the correction mechanism provisions more resources to the service.

3. The third implemented autoscaler is the regression controller (hereafter called *Reg*) described by Iqbal et. al. [26]. The autoscaler has a reactive component for scale-up decisions and a predictive component for scale-down decisions. When

the capacity is less than the load, a scale up decision is taken and new VMs are added to the service in a way similar to *React*. For scale down, the predictive component uses second order regression to predict the future workload. The regression model is recomputed using the complete history of the workload when a new measurement is available. If the current load is less than the provisioned capacity, a scale down decision is taken using the regression model.

4. The last implemented autoscaler is an adaptive hybrid controller (hereafter called *Adapt*) described in our previous work [6] that uses the workload slope to predict the future workload. The autoscaler has two components, a reactive component similar to *React* and a predictive component that predicts the future workload from the rate of change of the workload. The slope calculations are adaptive depending on the rate of change of the workload. If the load is increasing or decreasing rapidly and thus the rate of change is high, then the rate at which the autoscaler makes decisions is increased in order to adapt to rapid workload changes.

The two autoscalers for which we got the source codes are:

1. *AGILE*, proposed by Nguyen et al. [40] a wavelet-based algorithm to provide a medium-term resource demand prediction with a lead time of about 2 minutes, a time that allows AGILE to possibly start up new application server instances before performance falls short. We obtained the original source code from the authors for our experiments.

2. *ConPaaS*, proposed by Fernandez et al. [20]. The algorithm scales a web application in response to changes in throughput at fixed intervals of 10 minutes. The predictor forecasts the future service demand using standard time-series analysis techniques, e.g., Linear Regression, Auto Regressive Moving Average (ARMA), etc. The code for this auto-scaler is open-sourced. We downloaded the authors' original implementation.

4.2 How to Compare Elasticity autoscalers' Performance?

Many proposed state-of-the-art auto-scalers use the desired response time or service latency as a reference signal to a controller. A proposed approach then works if it meets the experiments' required response times, mostly set by the authors based on some QoS studies. Bodik [11] argues that this is not the correct reference an autoscaler should use. Even for fixed workloads, response time is noisy and non-linear with very high variance [12, 52]. This nonlinear relationship between resources provisioned and response time has been observed in large-scale distributed systems [16]. Latency does not also show under-provisioning and over-provisioning levels which is even

more important to compare the performance of different controllers. Two controllers might be achieving the required latency but one of them uses on average one tenth of the resources used by the other controller. Similarly, one controller can be dropping, on average, fewer requests than another one.

We add to Bodik's analysis that different applications have different processing requirements and a cloud infrastructure provider should not make any assumptions on how the resources are used, e.g., serving a static webpage hosted on the cloud is different from sorting a terabyte of data. In addition, any change in the workload mix changes the achievable latency [49]. The cloud paradigm is centered around resources on demand, so the measure of performance should be resource usage, e.g., average CPU and memory consumption or network utilization, or a metric that can be mapped to resource usage, which is not the case for latency. These metrics enable the provider to be as generic as possible when specifying the QoS in the Service Level Agreements (SLAs) with the customers.

To compare the performance of the autoscalers with the different workloads, we use the following cost metrics:

- Average Overprovisioning ($\overline{OP}$) is the average number of overprovisioned VMs by the autoscaler per unit time. It is calculated by summing the number of overprovisioned VMs over time (OP) and dividing the number by the total time for which the autoscaler was running. A machine is considered overprovisioned if it is of no use for the next 10 minutes. This time window reduces the penalty if an algorithm predicts the future workload well in advance.

- Average Underprovisioning ($\overline{UP}$) is the average number of underprovisioned VMs by the autoscaler per unit time. It is calculated by summing the number of underprovisioned VMs over time (UP) and dividing the number by the total time for which the autoscaler was running. Underprovisioning means that the autoscaler failed to provision the resources required to serve all requests on time.

- Average number of Oscillations ($\overline{O}$) which is the average number of VMs started or shut-down (O) per unit time. The reason to consider ($\overline{O}$) as an important parameter is the cost of starting/stopping a VM. From our experience, starting a machine (physical or virtual) takes from one minute up to several minutes depending on the application running (almost 20 minutes for an ERP application server). This time does not include the time required to transfer the images and any data needed but is rather the time for (virtual) machine boot, network setup and application initiation. Similar time may be required when a machine is shutdown for workload migration and load balancer reconfiguration.

Table 2 shows the cost metrics' values for the French Wikipedia workload using the six autoscalers with the average number of servers each autoscaler will deploy. These

Table 2: *The difference between the best performing autoscaler and the worst performing autoscaler can be quite large. For the French Wikipedia workload, when looking at $\overline{UP}$, the worst performing autoscaler is 28 folds worse than the best performing one.*

	$\overline{OP}$	$\overline{UP}$	$\overline{O}$
React	0.8	-0.53	0.04
Reg	1.26	-0.56	0.03
Adapt	0.7	-0.18	0.047
Hist	1.08	-0.28	0.04
AGILE	1.4	-3.5	0.065
ConPaaS	1.07	-5.1	0.038

numbers are obtained for a server system similar to the system described by Gandhi et al. [21]. The table shows clearly that the best performing server with respect to one metric can be multiple times better than the worst performing one. A service provider who deploys only one autoscaler can thus be doing orders of magnitude worse than others with a different autoscaler for a given workload. Since different applications have different requirements, application owners will give different priorities to the three parameters. For example, a traditional database system might be affected more by $\overline{O}$ since each change in the number of machines results in a heavy penalty in order to preserve consistency. Other applications might be affected more by $\overline{UP}$. Therefore, when comparing the autoscalers' performance, the values of $\overline{OP}$, $\overline{UP}$ and $\overline{O}$ are weighted by multiplying them by weighting factors α, β and γ respectively. In real deployments, the weights should be set based on the BLOs [32] and the QoS requirements by application owner.

We refer to the set of implemented autoscalers as C_I. To choose the best performing autoscaler for a given workload and a given set of weights, the cost metrics have to be calculated for each implemented autoscaler X. The total cost for using that autoscaler is,

$$Cost_X = \alpha\overline{OP} + \beta\overline{UP} + \gamma\overline{O}. \quad (1)$$

The best autoscaler for the given setup C_Z is thus,

$$C_Z = \{\chi | \chi \in C_I \;\&\; Cost_\chi \leq Cost_Y \;\forall\; Y \in C_I\}. \quad (2)$$

4.3 Performance of the autoscalers

The WAC classes are the implemented autoscalers described in Section 4. In order to label each workload in the training set, we need to know the best performing autoscaler for that workload. Since it is infeasible to run the experiments using more than 800 workloads on a real testbed, we have built a discrete event simulator using

Python, numpy and scipy [41] to simulate a cloud provider and test the autoscalers' performance with the various workloads. The simulator was parameterized using real experiments on our local cluster. In the real experiments, we replicate the Spanish Wikipedia in a VM on our local testbed. In addition, we ran experiments using C-Mart, a Cloud benchmark that has been developed recently [50].[3]

We divide the workloads described earlier into four sets, the first one is the set of real workloads with fine grained monitoring, the second one is the set of generated workloads, the third set is a combination of the previous two sets and the fourth one is the set with the 798 Wikimedia streams. In order to test a wide set of BLOs and scenarios in our experiments, we set the parameters in Equation 1 such that α varies between 0 to 1 (with a step of 0.01) for the real workloads with fine grained monitoring, 0 to 0.5 (with a step of 0.01) for the generated workloads, and 0 to 10 (with a step of 0.5) for the Wikimedia projects workloads. β varies between 1 to 10 for the real workloads with fine grained monitoring, 1 to 20 for the generated workloads and 1 to 5 for the Wikimedia streams, all with a step of 1. γ varies between 0 to 10 for the real workloads with fine grained monitoring, 0 to 20 for the generated workloads (both with a step of 0.5) and 0 to 5 for the Wikimedia stream.

Note that, we choose larger weights for β and γ for the first two workload sets (and their combination) since typically $\overline{UP}$ and $\overline{O}$ are more costly for most applications, the reason why almost all services were in the past run with static over-provisioning leading to low utilization on most datacenters. In order to show the wide applicability of the tool and that it is independent on the cost function, the fourth data set has cases with higher weights for α than the other two parameters.

The combinations of these values with the ACFs and the SampEn of the different workloads form the different scenarios with which we test WAC. For the real workloads with fine grained monitoring, there are in total $280,000$ scenarios, while for the generated workloads there are in total $2,090,000$ scenarios. For the Wikimedia streams there are 639200 scenarios in total.

Each combination is labeled with the best performing autoscaler. The second and third columns in Table 3 show the percentages of scenarios in which one of the four autoscaler we implemented outperforms the other three autoscalers for the real workloads with fine granularity of monitoring data and for the generated workloads. From the table we see that, *Reg* and *Adapt* outperform the other autoscalers considerably for both the real and the synthetic traces. *React* is the worst performing autoscaler for both workload types. Note that we have decided not to include the results for *AGILE* and *ConPaaS* for the first two sets in order to test the tool under different cases and with different autoscalers deployed. Similarly, the last column in Table 3 shows the percentages of scenarios in which one of five autoscalers outperforms the other four.

[3]Due to lack of space, we will not discuss the simulator here. The simulator source code along with all the tests will be open sourced.

Table 3: *Percentage of scenarios in which every autoscaler outperforms the other autoscalers for the different workload sets.*

autoscaler	Real workloads (fine grained)	Generated workloads	Wikimedia
React	6.55%	0.10%	NA
Reg	33.72%	61.33%	19.2%
Adapt	47.17%	34.30%	47%
Hist	12.56%	4.27%	29.5%
AGILE	NA	NA	1%
ConPaaS	NA	NA	3%

We attribute the performance of the autoscalers to the way they are designed. *Adapt* is designed with the aim of reducing $\overline{UP}$ and $\overline{O}$ at the cost of higher $\overline{OP}$. *Hist* is designed for workloads that have periodical patterns and low burstiness. *Hist* does not give any importance to $\overline{OP}$ as scale down mechanisms are absent in case the autoscaler provisions extra resources. *React* has no predictive component. It provisions resources after they are needed. This reduces the amount of $\overline{OP}$ at the cost of increased $\overline{UP}$ and $\overline{O}$. The modified *Reg* uses a second order regressor which is suitable for capturing patterns in the workload. *Reg* scales down resources faster than *Hist*, thus reducing $\overline{OP}$, but slower than *React*. Due to lack of space, we do not include deeper analysis or quantify $\overline{OP}$, $\overline{UP}$ and $\overline{O}$ values for different scenarios.

5 Workload Classification

5.1 Classifiers

The kNN and the SVM classifiers are among the most widely used classifiers [18,48]. SVM is a supervised learning method introduced by Boser et. al. [13]. It constructs separating hyperplanes to separate the training data so that the distance between the constructed hyperplanes and any training point is maximized. Training an SVM classifier and tuning its different parameters is complex [9]. We have tried using an SVM classifier but the training was very time consuming. We choose to use kNN instead since it is less complex, fast to train and fast to calculate an assignment once trained.

The kNN algorithm is quite simple. A training set is used to train the classifier. The training set is a set of samples for which the correct classification in known. When a new unknown sample arrives, a majority vote of its K-nearest neighbors is taken. The new sample is then assigned to the class with a majority vote. K is a configurable

parameter. The distance to a neighbor is usually the Euclidean distance between the sample and that point. There are two versions of the kNN algorithm. One version gives more weights to votes of closer neighbors while the second version gives equal weights to all k neighbors [18].

5.2 Experiment Design

In the experiments, we consider the case when only two autoscalers deployed, the case when only two autoscalers, *Adapt* and *Reg*, are deployed, and the case when five autoscalers are deployed. *Adapt* and *Reg* perform better than *Hist* and *React* for more than 80% of the workload scenarios considered when looking at the fine grained real workloads set, the generated workloads set and the combination of these two sets. For each experiment, each scenario in the training set is labeled with the top performing autoscaler. The scenarios described in Section 4 are randomly split into a training set and a test set. In some experiments, the training set contains 75% of all the scenarios (for both training and cross-validation) while the test set has the remaining 25%. In other experiments 90% of the scenarios are used as a training set (for both training and cross-validation), while the remaining 10% is used for testing. This division is plausible since our data set is large enough [18]. We note that for both divisions, the accuracy of prediction results are very close. Both the training set and the test set have no single class dominating or skewing the training set.

Both sets are labeled with the right answer for each scenario. When the classifier assigns a workload to an elasticity autoscaler, the assignment is compared to the label. If the assignment matches the label, then the classification is correct. The accuracy of the classifier is calculated by calculating the ratio of correct classifications of the test set to the total size of the test set. We test the kNN classifier using both its versions, the one that gives equal weights to all K neighbors and the one that gives more weight to closer neighbors. For each version, we repeat the experiment for increasing K and calculate the classification accuracy.

5.3 Results

5.3.1 Classification of Real Workloads with fine grained monitoring

Figure 5(a) shows the classification accuracy of the classifier when there are only two classes, *Adapt* and *Reg*. The x axis of the figure is the value of K. The y axis of the figure is the accuracy of classification. The black squares shows the accuracy of the classifier with changing K when all k neighbors' votes are given equal weights. Significant oscillations occur in the accuracy as K increases until K is around 50. When closer neighbors' votes are given higher weight, oscillations are much smaller as the green triangles show where the accuracy stabilizes at around 0.92 when K is

equal to 25. The maximum accuracy achieved is 0.983 using equal weights for all neighbors for $K = 5$.

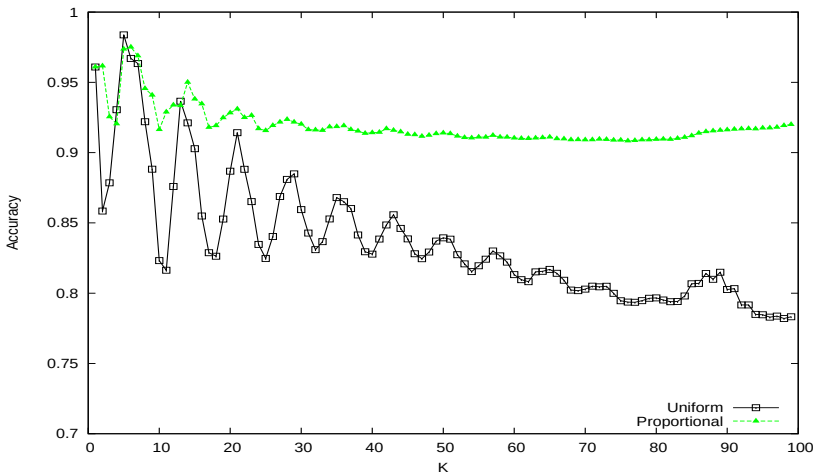
Figure 5(b) shows the classification accuracy of the classifier when there are four classes, *Adapt*, *React*, *Histo* and *Reg*, against K when K is varied from 1 to 200. The figure shows less oscillations compared to Figure 5(a). The maximum accuracy achieved is 0.91 using equal weights for all neighbors for $K = 4$. The accuracy of classification is reduced by more than 7% when more autoscalers are available. This might be due to not having enough training data in the training set for the classifier to be able to generalize for four classes. An accuracy of 91% is still considered good. For the service provider it means that instead of having just one autoscaler, the provider can have 4 autoscalers with only 9% of the workloads assigned in a non-optimal way. Since more training data, means more experience, the provider can further decrease the percentage of workloads assigned sup-optimally by constantly adding feedback to the classifier by adding more training points based on new workload behaviors observed. Figures 5(a) and 5(b) show that accuracy decreases with increasing K when all K neighbors' votes are given equal weights. When neighbors' votes are weighted such that closer neighbors have a higher weight, the accuracy stabilizes as K increases.

5.3.2 Classification of Generated Workloads

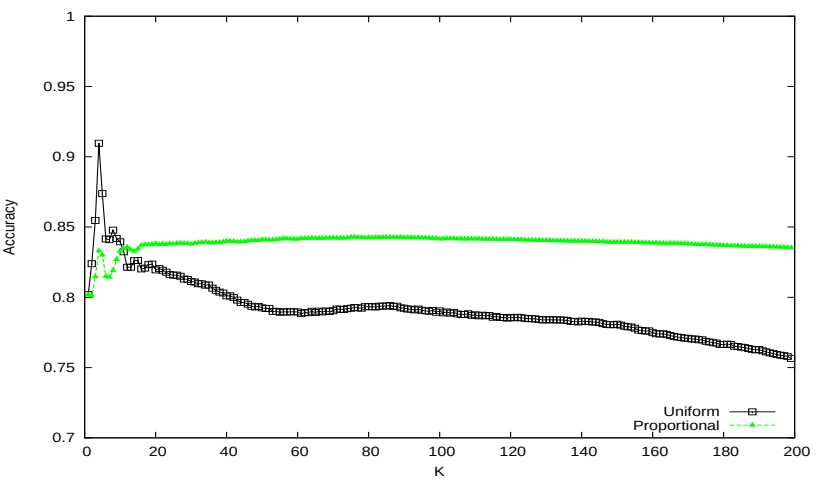
We repeat the tests described above using only the generated workloads. Figure 6(a) shows the classification accuracy of the classifier when there are only two classes, *Adapt* and *Reg*, against K. The maximum accuracy achieved is 0.92 using equal weights for all neighbors for $K = 3$. Figure 6(b) shows the classification accuracy of the classifier when there are four classes, *Adapt*, *React*, *Histo* and *Reg*, against K. The maximum accuracy achieved is 0.94 using equal weights for all neighbors for $K = 3$. Comparing Figure 5 to Figure 6, oscillations in the classifier's accuracy with increasing K is much lower when classifying synthetic workloads compared to when classifying real workloads.

5.3.3 Classification of Mixed Workloads

In our last experiment, we combine the scenarios using real and generated workloads in a single set. This set is again randomly shuffled and divided into a training set and a test set. Figure 7(a) shows the classification accuracy of the classifier when there are only two classes, *Adapt* and *Reg*, against K. The maximum accuracy achieved is 0.92 using any of the two versions of the kNN for $K = 5$. Figure 7(b) shows the classification accuracy of the classifier when there are four classes, *Adapt*, *React*, *Histo* and *Reg*, against K. The maximum accuracy achieved is 0.92 using equal weights for all neighbors for $K = 3$. We note that the oscillations in the classifier's

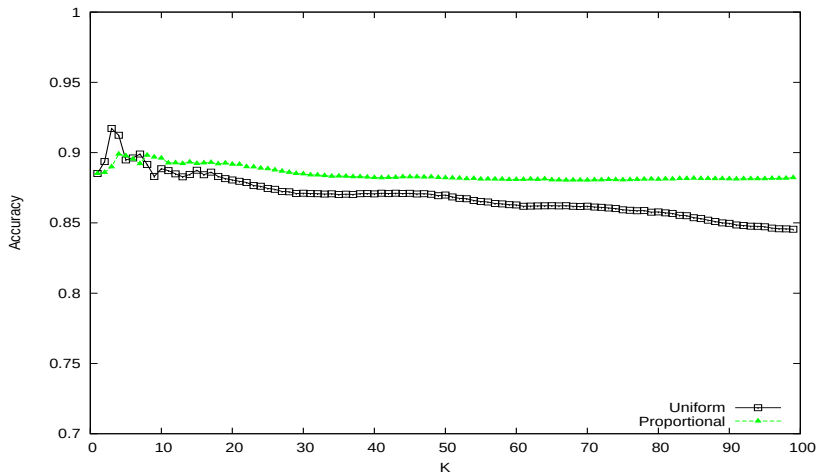


(a) Effect of changing K on the classifier accuracy with two autoscalers *Adapt* and *Reg* (real workloads).

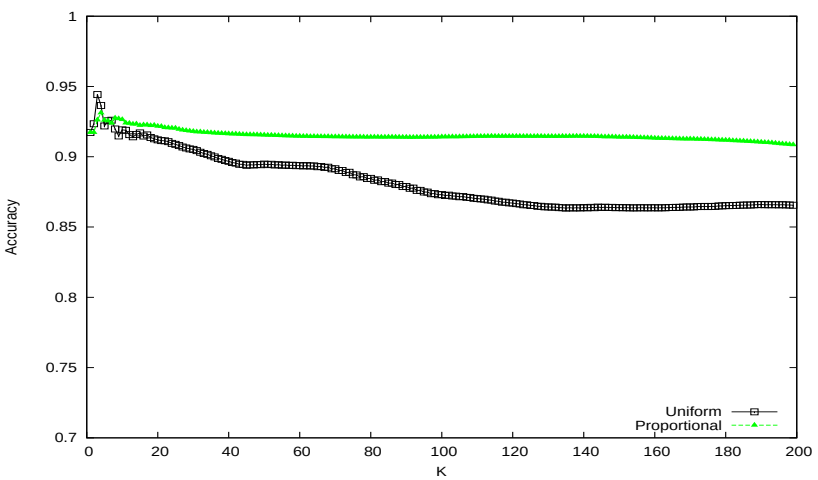


(b) Effect of changing K on the classifier accuracy with four autoscalers (real workloads).

Figure 5: Choosing K affects the accuracy of the classifier.



(a) Effect of changing K on the classifier accuracy with two autoscalers (generated workloads).



(b) Effect of changing K on the classifier accuracy with four autoscalers (generated workloads).

Figure 6: Classifier accuracy is lower for the generated workloads compared to the real workloads, but more stable with changing K.

accuracy with changing K in Figure 7 are not as severe as in the case of classifying real workloads.

5.3.4 Classification of the Wikimedia projects workloads

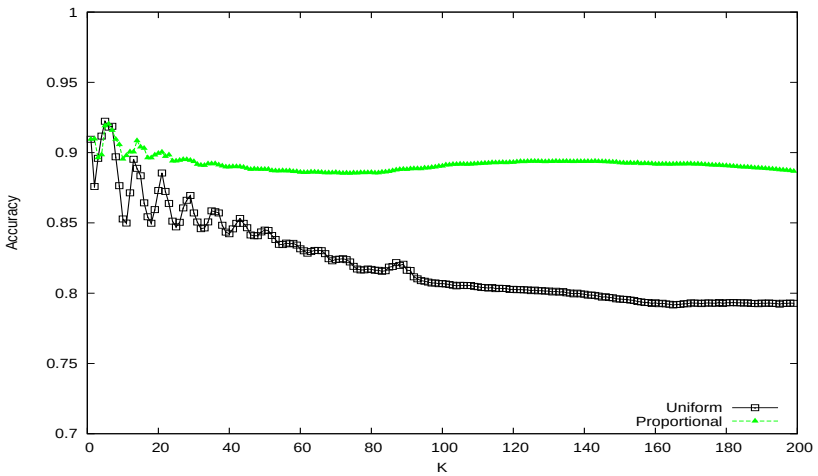
In our last experiment, we use five autoscalers, namely, *Reg*, *Adapt*, *Hist*, *ConPaaS*, and *AGILE*. The workload set used is the Wikimedia projects workloads. The set is again randomly shuffled and divided into a training set and a test set. Figure 8 shows the classification accuracy of the classifier against changing K. The maximum accuracy achieved is 0.87 using the proportional version of the kNN for $K = 5$. Since *AGILE* and *ConPaaS* are only performing well on a small fraction of this workload set, the classifier is unable in many cases to assign workloads to the two autoscalers. We have tried different parameterization of the autoscalers to improve their performance, but were unable to achieve any improvements.

5.4 Discussion of the Results

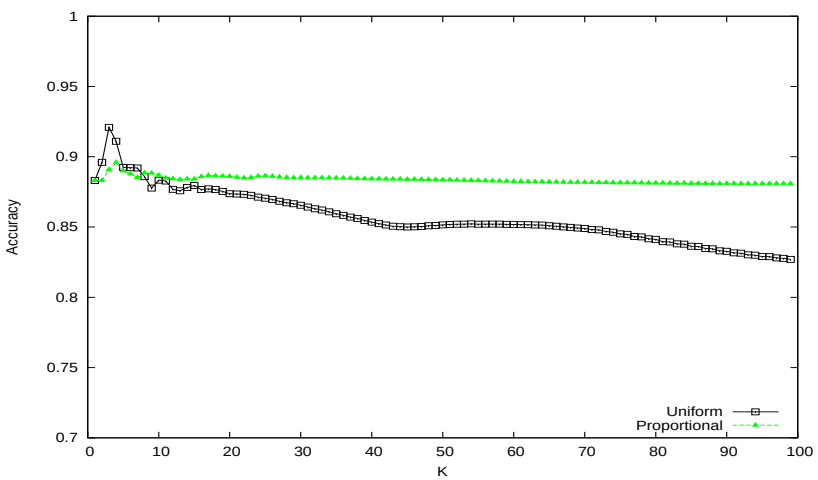
The oscillations in the figures (except for Figure 8) for lower values of K are attributed to the nature of the kNN algorithm. If F is the number of features used for classification, kNN constructs an F-dimensional space and places each training object in that space based on the object's features values. It divides the F-dimensional space into neighborhoods according to their classes. These neighborhoods can be scattered in the constructed space. Some of these neighborhoods can be small and surrounded completely by larger neighborhoods of a different class. When the number of workloads in the training set increase, e.g., in the Wikimedia streams case where we have more than 700 independent workloads, these oscillations disappear since the classifier is able to construct better classes.

Picking a small value for K results in correct classification of these smaller neighborhoods for the training data. On the other hand, it can result in over-fitting the training data and forming small neighborhoods laying completely surrounded within large neighborhoods of a different class. This makes the classifier sensitive to noise in the training set. A large value of K results in a less noise sensitive classifier and smoother decision boundaries but at the cost of increased training error, increased computational burden and loss of locality of the estimations since far away points affect the decision [29].

Oscillations are more obvious when the distance to all K neighbors have equal weights. When closer neighbors are given higher weight, the classification accuracy is more stable with increasing K. By looking at figs. 5 to 7, we observe that the accuracy of classification decreases with increasing K. The stabilization for the kNN version with weighted distance is faster and with a stable accuracy higher than the other version. When setting K in real life deployments, if the training set is comprehensive



(a) Effect of changing K on the classifier accuracy with two autoscalers (mixed workloads).



(b) Effect of changing K on the classifier accuracy with four autoscalers (mixed workloads).

Figure 7: Training the classifier with a mixed workload set of real and generated workloads makes the classification more robust with changing K compared to only training the classifier with real workloads.

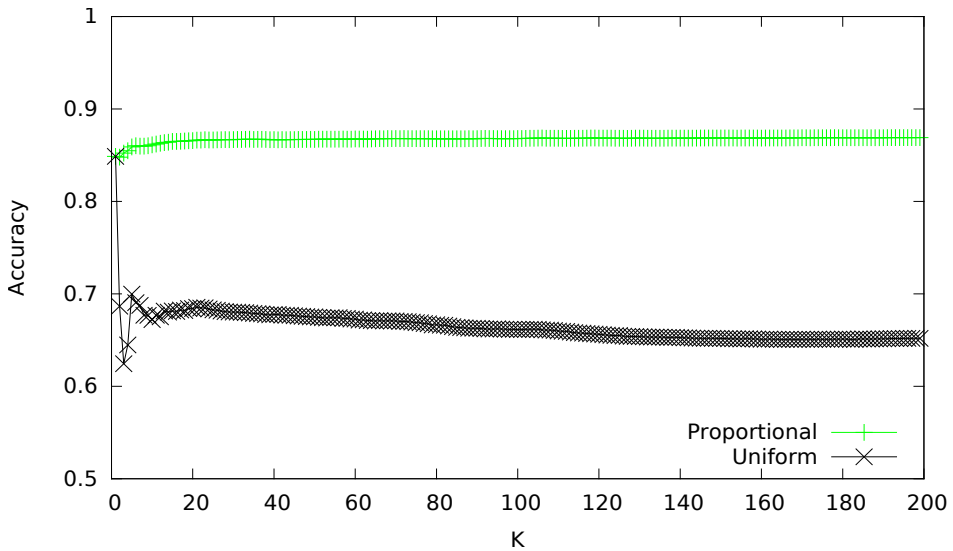


Figure 8: Effect of changing K on the classifier accuracy with five autoscalers and the Wikimedia streams.

with most workload variations expected, K can be chosen to be the one that gives best accuracy on the training set regardless of the stability. Otherwise, if the training set is not comprehensive K should be chosen to be the least K with the highest stable accuracy.

6 Related Work

AutoFlex is a framework for the implementation of auto-scaling services that follows both reactive and proactive approaches [7]. The proactive approach is based on the use of a set of predictors of the future demand of services deployed over IaaS resources, and a selection mechanism that chooses, over time, what is the best predictor to be used. while AutoFlex has some similarities to WAC, the systems are different in many ways. In principle, AutoFlex simulates the performance of the different predictors using the past two weeks of data and chooses the predictor with the least violations. WAC analyzes the historical workload and based on the workload characteristics, assigns the workload streams to autoscalers according to the analysis. There is no way in Autoflex to incorporate BLOs easily. In addition, we have tested WAC with elasticity autoscalers proposed in the literature while AutoFlex was tested using 6

of-the-shelf statistical predictors, most of them are not suitable for non-stationary workloads.

Previous work on workload characterization can be classified into two main directions. The first studies the methodologies used for workload characterization [14, 19, 35]. The second is the analysis and characterization of real workload traces [4, 8, 54]. Among the research done in the first direction is the work by Calzarossa and Serazzi [14] on the steps required to model a workload and the work by Downey and Feitelson [17] on modeling issues for parallel workloads that result in building unrealistic models.

Several workloads for various internet services have been analyzed previously. Arlit and Jin presented a detailed workload study of the 1998 FIFA World Cup website [8] between May 1st, 1998 until July 23rd, 1998 were analyzed. The logs contain 1.35 Billion requests and almost 5 TB of data were sent to the clients. Data was accessed by more than 2 million unique IP addresses. There has also been a rising interest to characterize Map-reduce workloads from production clusters [30, 43, 45] and grid workloads [24]. A long trace provided by Google spanning 29 days for a backend cluster was analyzed by Reiss et. al. [43]. The workload is dominated by 'normal production' tasks which have regular patterns. Lower priority tasks make up between 10% to 20% of the CPU usage in the cluster. These tasks have no regular patterns and are very bursty. Iosup and Epema analyze 15 different grid workloads [24]. They report that the average system utilization of some research grids was as low as 10 to 15% but considerably high on production grids. All 15 grids experienced overloads during some short term periods. They note that loosely coupled jobs are dominating most of the workloads. Loosely coupled jobs are suitable to run on the cloud. Similar findings are reported by Zhang et. al. [55]. Khan et. al. study workloads running on different servers in a cluster in order to find which servers frequently have correlated workload patterns [33]. They use a greedy algorithm to solve a computationally intractable problem to find the clusters that constitute correlated workloads on different servers.

Keogh and Kasetty discuss the published work on time-series data mining [31]. They show that much of the published work is of very little utility. They implement the contributions of more than two dozen papers and test them on 50 real workloads. For classification, they implement 11 published similarity measures and compare their performance to the Euclidean distance measure. None of the 11 proposed similarity measures is able to outperform the Euclidean distance.

7 Conclusions, Limitations and Future Work

In this work, we show the feasibility of designing and building an automated workload analysis and classification tool for cloud workloads where the tool assigns a workload

to the most suitable elasticity autoscaler based on a set of workload characteristics. While we only used two workload characteristics as features for the classification, the workload periodicity and burstiness, the tool (if tuned carefully) achieves an accuracy of assignment ranging between 0.87 up to 0.98, i.e., between 87% and 98% of the workloads are assigned to the most suitable elasticity autoscaler. Among the current limitations of our work is the way we choose the autoscalers. The six autoscalers we tested WAC with are chosen empirically. While the literature has a very rich selection of proposed elasticity autoscalers, there is a lack of understanding on how a designed autoscaler works with different workload scenarios. Most of the autoscalers are either designed ad-hoc or for a very special model/case. A related limitation is the lack of understanding of the cloud workload space. For future work, we intend to cover described limitations. In addition, we plan to design an online version of WAC able to change the assignment of the autoscaler to a workload whenever the workload characteristics change.

Acknowledgment

We would like to thank John Wilkes, Hiep Nguyen and Xiaohui Gu for providing us the source code for Agile. We would also like to thank Hector Fernandez, Guillaume Pierre and Thilo Kielmann for having the source code of ConPaaS open. We would like to thank the ASCI team at VU, Amsterdam, the Grid'5000 team, the AuverGrid team, John Morton and Clayton Chrusch, the e-Science-HEP at Imperial College London and the NorduGrid team for providing access to the Grid traces hosted on the Grid Workloads Archive.

Financial support for this work has been provided in part by the Swedish Government's strategic effort eSSENCE, the European Union's Seventh Framework Programme under grant agreement 610711 for the project CACTOS, and the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). Accessed: May, 2013.

[2] Bulgarian Wiktionary project. URL:http://bg.wiktionary.org/wiki/

[3] Top Sites. Accessed: November, 2013, URL: http://www.alexa.com/topsites.

[4] A. Ali-Eldin, A. Rezaie, A. Mehta, S. Razroev, S. Sjöstedt-de Luna, O. Seleznjev, J. Tordsson, and E. Elmroth. How will your workload look like in 6 years? Analyzing Wikimedia's workload. In *IC2E*. IEEE Computer Society, 2014.

[5] A. Ali-Eldin, O. Seleznjev, S. Sjöstedt-de Luna, J. Tordsson, and E. Elmroth. Measuring cloud workload burstiness. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 566–572. IEEE Computer Society, 2014.

[6] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *IEEE NOMS*, pages 204–212. IEEE, 2012.

[7] F. Almeida Morais, F. Vilar Brasileiro, R. Vigolvino Lopes, R. Araujo Santos, W. Satterfield, and L. Rosa. Autoflex: Service agnostic auto-scaling framework for IaaS deployment models. In *IEEE/ACM CCGrid*, pages 42–49, 2013.

[8] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, 2000.

[9] G. Bakir, L. Bottou, and J. Weston. Breaking SVM complexity with cross training. *NIPS*, 17:81–88, 2005.

[10] D. J. Barrett. *MediaWiki (Wikipedia and Beyond)*. " O'Reilly Media, Inc.", 2008.

[11] P. Bodik. *Automating Datacenter Operations Using Machine Learning*. PhD thesis, UC-Berkeley, 2010.

[12] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC*, pages 241–252. ACM, 2010.

[13] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *ACM COLT*, pages 144–152. ACM, 1992.

[14] M. Calzarossa and G. Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, 1993.

[15] T. Chieu, A. Mohindra, A. Karve, and A. Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *IEEE ICEBE*, pages 281 –286, 2009.

[16] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[17] A. Downey and D. Feitelson. The elusive goal of workload characterization. *ACM PER*, 26(4):14–29, 1999.

[18] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. John Wiley & Sons, 2012.

[19] D. G. Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2014.

[20] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. In *IEEE Int. Conf. on Cloud Engineering*, 2014.

[21] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. AutoScale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, Nov. 2012.

[22] N. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *ACM/SPEC ICPE*, pages 187–198. ACM, 2013.

[23] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[24] A. Iosup and D. Epema. Grid computing workloads. *IEEE Internet Computing*, 15(2):19–26, 2011.

[25] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. Epema. The grid workloads archive. *FGCS*, 24(7):672–686, 2008.

[26] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *FGCS*, 27(6):871–879, 2011.

[27] IRCache. Access to trace files. Accessed: May, 2013.

[28] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *FGCS*, 28(1):155–162, 2012.

[29] L. Kankanala and M. N. Murty. Hybrid approaches for clustering. In *PReMI*, pages 25–32. Springer, 2007.

[30] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *IEEE/ACM CCGrid*, pages 94–103, 2010.

[31] E. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. In *ACM SIGKDD KDD*, pages 102–111. ACM, 2002.

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

[33] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *IEEE NOMS*, pages 1287–1294. IEEE, 2012.

[34] M. Lassnig, T. Fahringer, V. Garonne, A. Molfetas, and M. Branco. Identification, modelling and prediction of non-periodic bursts in workloads. In *IEEE/ACM CCGrid*, pages 485–494, 2010.

[35] H. Li. Workload dynamics on clusters and grids. *The Journal of Supercomputing*, 47(1):1–20, 2009.

[36] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *IEEE/ACM ICAC*, pages 1–10. ACM, 2010.

[37] T. N. Minh, L. Wolters, and D. Epema. A realistic integrated model of parallel system workloads. In *IEEE CCGrid*, pages 464–473, 2010.

[38] M. Morari and E. Zafiriou. *Robust process control*. Morari, 1989.

[39] NCBI. PubMed. Accessed: May, 2013.

[40] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic distributed resource scaling for Infrastructure-as-a-Service. In *Proc. 10th Int. Conf. on Autonomic Computing*, pages 69–82, 2013.

[41] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[42] L. R. Rabiner and J. G. Wilpon. Considerations in applying clustering techniques to speaker-independent word recognition. *The Journal of the Acoustical Society of America*, 66(3):663–673, 1979.

[43] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ÙSoCC*, pages 7:1–7:13. ACM, 2012.

[44] Amazon Web Services. High performance computing (HPC) on AWS. Accessed: May, 2013.

[45] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou. Workload characterization on a production hadoop cluster: A case study on taobao. In *IISWC*, pages 3–13. IEEE, 2012.

[46] J. S. Richman, D. E. Lake, and J. R. Moorman. Sample entropy. *Methods in enzymology*, 384:172–184, 2004.

[47] A. Rubin. *Statistics for evidence-based practice and evaluation*. Brooks/Cole, 2012.

[48] B. Schölkopf and A. J. Smola. *Learning with kernels: support vector machines, regularization, optimization and beyond*. MIT Press, 2002.

[49] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *IEEE/ACM ICAC*, pages 21–30. ACM, 2010.

[50] A. Turner, A. Fox, J. Payne, and H. S. Kim. C-mart: Benchmarking the cloud. *IEEE Trans. on Parallel and Distributed Systems*, 24(6):1256–1266, 2013.

[51] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM TAAS*, 3(1):1, 2008.

[52] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 31–40. IEEE, 2013.

[53] J. Wilkes. More Google cluster data. Accessed: May, 2013.

[54] H. Xi, J. Zhan, Z. Jia, X. Hong, L. Wang, L. Zhang, N. Sun, and G. Lu. Characterization of real workloads of web search engines. In *IISWC*, pages 15–25. IEEE, 2011.

[55] Q. Zhang, J. Hellerstein, and R. Boutaba. Characterizing task usage shapes in Googleâs compute clusters. In *LADIS*, 2011.

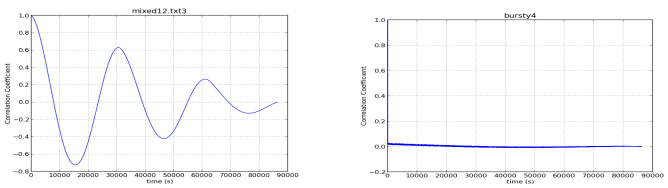
A Sample Entropy Algorithm

The sample entropy algorithm as shown in Algorithm 1. W is the workload for which SampEn is calculated. The first loop in the algorithm calculates $B^m(r)$, the probability that two sequences in the workload having m measurements do not have bursts. The second loop in the algorithm calculates $A^m(r)$, the probability that two sequences in the workload having $m+1$ measurements do not have bursts. Then SampEn is calculated as the negative logarithm of the conditional probability that two workloads sequences of length m that do not have bursts, have no bursts when the sequence length is increased by 1.

Data: r, m, T, L
Result: *AvgSampEn*

$N \leftarrow Length(L)$;
$P_{divided} \leftarrow \{T(L.k).....T(L.(k+1)\}\ \forall k \in \{0,N\}$;
$Tot_{SampEn} \leftarrow 0$;
for W *in* $P_{divided}$ **do**
  $n \leftarrow Length(W)$;
  $B_i \leftarrow 0$;
  $A_i \leftarrow 0$;
  $X_m \leftarrow \{X_m(i)|X_m(i) = [x(i),...x(i+m-1)]\ \forall\ 1 < i < n-m+1\}$;
  for $(X_m(i), X_m(j))$ *in* X_m*:* $i \neq j$ **do**
    Calculate $d[X_m(i), X_m(j)] = \max(|x(i+k) - x(j+k)|)\ \forall\ 0 \leq k < m$;
    if $d[X_m(i), X_m(j)] \leq r$ **then**
      $B_i \leftarrow B_i + 1$;
    end
  end
  $B^m(r) \leftarrow \frac{1}{n-m} \sum_{i=1}^{n-m} \frac{1}{n-m-1} B_i$;
  $m = m+1$
  $X_m \leftarrow \{X_m(i)|X_m(i) = [x(i),...x(i+m-1)]\ \forall\ 1 < i < n-m+1\}$;
  for $(X_m(i), X_m(j))$ *in* X_m*:* $i \neq j$ **do**
    Calculate $d[X_m(i), X_m(j)] = \max(|x(i+k) - x(j+k)|)\ \forall\ 0 \leq k < m$;
    if $d[X_m(i), X_m(j)] \leq r$ **then**
      $A_i \leftarrow A_i + 1$;
    end
  end
  $A^m(r) \leftarrow \frac{1}{n-m} \sum_{i=1}^{n-m} \frac{1}{n-m-1} A_i$;
  $Tot_{SampEn} \leftarrow |-\log[\frac{A^m(r)}{B^m(r)}]| + a \times Tot_{SampEn}$;
end
$AvgSampEn \leftarrow Tot_{SampEn}/N$;

Algorithm 1: The algorithm for calculating *AvgSampEn*.



(a) Sample generated workload WL1. (b) Sample generated workload WL2.

Figure 9: Correlograms of two of the generated workloads.

B Workload generation

Equation 3 shows how WL1 whose ACF is shown in Figure 9(a) is generated.

$$x(i) = \log(i+10^3)(|\sin(\frac{3i}{10^4})+100|)(|\sin(\frac{i}{10^4}))+2*10^4|) \\ +W(0.3), \quad \forall\, i \in [0,n], \tag{3}$$

where $x(i)$ is the workload value at time i, $W(0.3)$ is Weibull distribution with a shape parameter equal to 0.3 and n is the length of the synthetic workload required. The log function is used to simulate a workload evolving with time but having similar periodicity. The sinusoids are used to give the workload a pattern similar to diurnal patterns and the Weibull distribution adds some uncertainty in the workload. This is similar to the pattern seen in the Wikipedia workload trace. Figure 9(b) shows the correlograms for WL2 . The two workloads are generated with an equation similar to Equation 3, but with an added component that increases the uncertainty (SampEn) in the workload thus decreasing the ACF. Figure 9(b) shows that the ACFs for WL2 and WL4 resemble the ACFs of the DAS, Grid5000 and LCG real workloads respectively. Some of the generated workloads have ACFs different from the ones seen in the real workloads. The equations used to generate the 55 synthetic workloads have mostly similar structures, but using different periods for the sinusoids, different amplitudes and different probability distributions such as log normal distributions, gamma distributions and chi-squared distributions as sources of uncertainty. Table 4 shows the SampEn values for the six generated workloads above. Similar to the real workloads, the generated workloads have burstiness levels ranging from no burstiness

Table 4: *Average Sample Entropy computed for the sample generated workloads.*

Workload	WL1	WL2	WL3	WL4	WL5	WL6
SampEn	0.0	236.0	2.5	0.002	0.0014	0.069

to very bursty. Due to lack of space and since this is not the focus of this work, we omit further details on workload generation.

Appendix

Workloads and Controllers

Appendix

This appendix is divided into two sections. In the first section, Section A, we show the results of some of our real experiments used to parametrize the simulations. Since the Wikipedia workload is central to this thesis, we add some further analysis to the analysis done in Paper III, we extend the analysis in Paper III in Appendix B.

A Experiments

We have setup an experimental testbed with replicas of some Wikimedia services. In this Section, we use a setup with the German Wikipedia as our replicated application. [1] The German Wikipedia is the second largest language Wikipedia by number of views per hour coming only after the English Wikipedia [2]. It is also the third largest language considering the number of articles [2]. For the purpose of our experiments, we have designed a workload generator that randomly accesses a set of pages from the replicated German Wikipedia. The generator is able to generate synthetic traces in addition to replaying workload traces from the Wikipedia logs.

Figure 1 shows the results of an experiment where a single VM was loaded gradually using the workload generator. The Figure shows the trade-off between the QoS achieved in terms of response time and the load on the machine. Every 10 seconds, the workload generator generates a new batch of requests starting from 1 request to 100 requests with a 1 request increase in each following batch, i.e., when the experiment starts, the generator generates 1 request, waits for 10 seconds and generates 2 requests, until 100 requests are generated. If the machine is congested and any of the requests is delayed, the request is buffered for 10 seconds before it gets dropped. We choose this gap between request generation times to reduce any queuing effects in the system and thus measure the response time for the machine unloaded. We choose the 10 seconds period before dropping requests since many usability experts argue that users will lose interest or abandon a website if the response time is beyond 10 seconds [4, 7, 8].

In Figure 1, we plot the number of requests served before some requests started being dropped due to congestion and the maximum response time in any batch of

[1] For instructions on how to replicate any Wiki, please refer to the MediaWiki documentation and to [1]

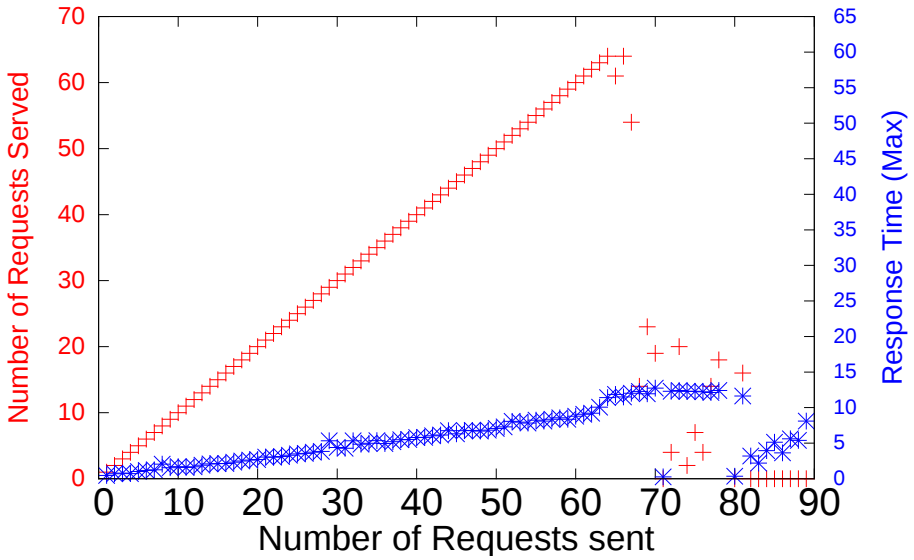


Figure 1: Response time and maximum number of requests a Virtual machine with 2 cores and 3 GB of memory can serve.

requests. As the number of requests sent increases, the response time increases until the VM becomes completely congested. When the machine becomes completely congested, most requests are dropped, and very few requests are served. We note that many of the autoscalers discussed in this thesis require as a parameter an average number of requests that a server can serve per unit time.

B Modeling a workload as a Markov Chain

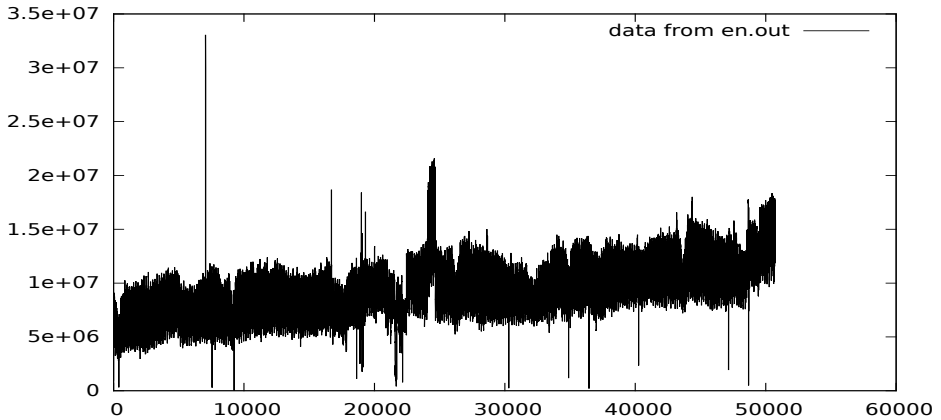
Markov Models are stochastic models that have been used for modeling systems that exhibit the *Markovian property* [10]. Given a system in a state X_n, the Markovian property states that the conditional distribution of any future state X_{n+1} depends only on the present state X_n with a certain probability $P(X_{n+1}|X_n)$ and not on any past state $X_{n-m} \forall m \in \{1,2,3,..\}$, i.e, $P(X_{n+1}|X_{n-m}) = 0$ [6]. Markov chains are stochastic processes exhibiting the Markovian property [6]. The transition probabilities between the current state and all possible future states are typically represented in a matrix form. The matrix is called the transition matrix [6]. Markov chains are widely used in system modeling, control, and optimization [9]. The Markovian property is usually stated as an assumption during system modeling. While there has been some work in the literature on testing for the property [3], these methods are seldom used.

Assuming the Markovian property holds for a workload, we model three major language request streams as Markov Chains, namely English, Swedish and Spanish. In this Appendix, we discuss only the results for the English and the Swedish request streams shown in Figure 2. The x-axis represents the time in hours while the y-axis represents the number of requests. We define a state of the system to be the percentile range in which the workload intensity is, i.e., if the y-axis is divided equally in 10 percentile ranges where the lowest range – State 1 – represents all times the load intensity falls within 10% of the maximum number of requests. Any spike occurring in the load means that the number of requests at the time the spike occurs falls within State 10, i.e., the number of requests is between 90% and 100% of the value of the maximum load seen. We have thus refined the 10th state into two more states, one representing the 95th percentile and one representing the 97th percentile of the maximum number of requests in the load. This approach is similar to the approach used by PRESS to predict the future workload [5].

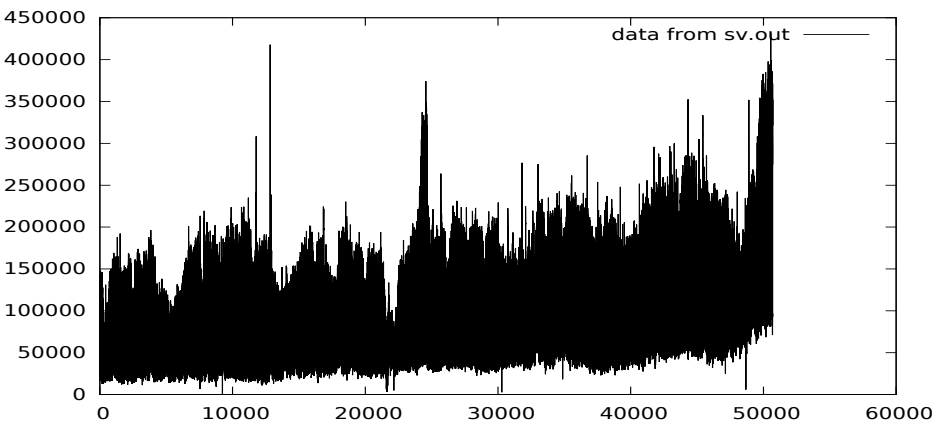
In order to build the transition matrix, using Python we use part of the data as a training set used build an initial transition matrix that gets updated as new monitoring data is available. For the training set, we use the first 1000 measurements in our trace which contains more than 50000 measurements. The trace is then replayed using the Python script and the transition matrix updated with each new measurement. Figure 3 shows the histograms for the transitions between the different states given a certain state for the English Stream. We note that, for any state, the workload stays in the same state with a higher probability. The probability to move from th current state to a state that is far away, i.e., from State 1 to State 9, is minimal. This is expected since the English Wikipedia workload is repetitive with very few abrupt changes in the trace. Designing a predictor for this workload is thus easies than a workload with high probabilities of transition between the different states.

To show an example of a workload with more abrupt changes, the histograms for the state transition probabilities for the Swedish Wikipedia workload is shown in Figure 4. Looking at the transition probabilities in the Figure, it is clear that the probability of transition from the current state to another far away state is higher than such a probability in the English workload. For example, Figures 3(g) and 4(g) show the transition probabilities form the 7th state to all other states. It is clear that moving from State 7 to State 4 is very unlikely in the English Wikipedia workload but likely in the Swedish Wikipedia Workload. The probability of moving from State 7 to State 6 or State 8 is more likely in the English Wikipedia workload than for the Swedish Wikipedia.

We have done this analysis on multiple workloads and have obtained similar results. These results can be further refined by increasing the number of states at the cost of complicating the model. These models give us a better understanding of the workloads. As future work, We are currently developing a tool for integrated service admission

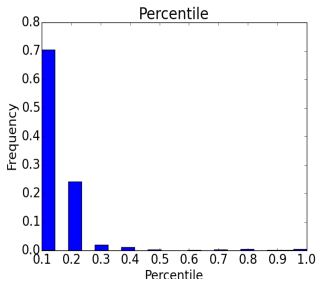


(a) English workload: Number of requests served per hour.

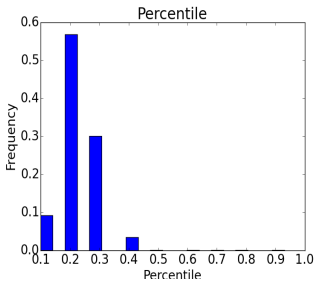


(b) Swedish workload: Number of requests served per hour.

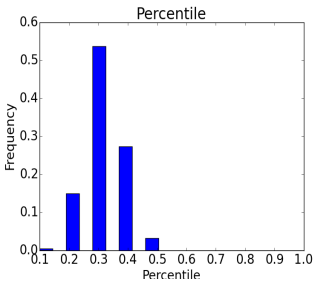
Figure 2: The English and Swedish Wikipedia Workload streams.



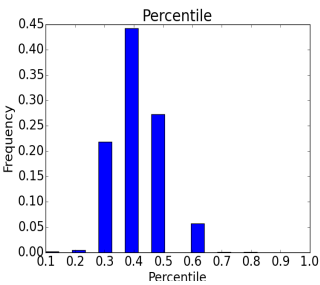
(a) State 1: 10th percentile.



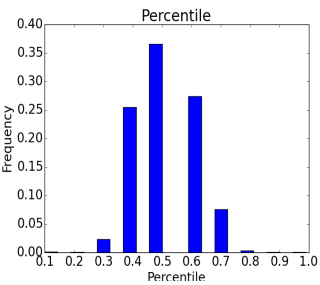
(b) State 2: 20th percentile.



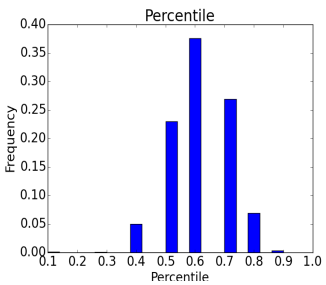
(c) State 3: 30th percentile.



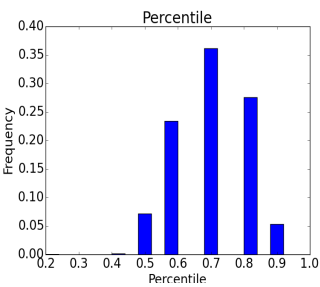
(d) State 4: 40th percentile.



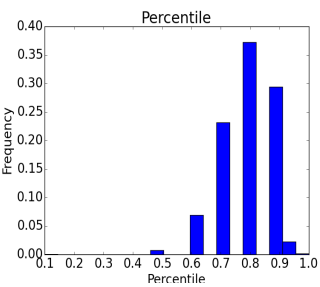
(e) State 5: 50th percentile.



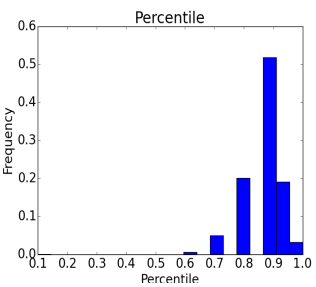
(f) State 6: 60th percentile.



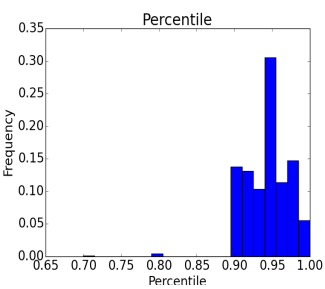
(g) State 7: 70th percentile.



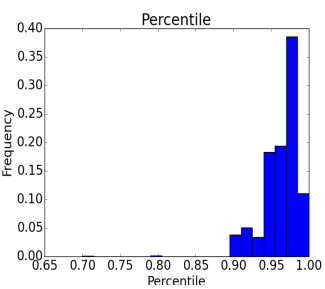
(h) State 8: 80th percentile.



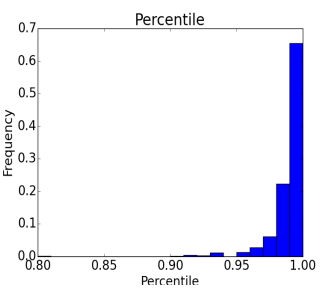
(i) State 9: 90th percentile



(j) State 10: 95th percentile.

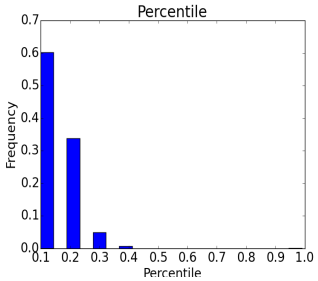


(k) State 11: 99th percentile.

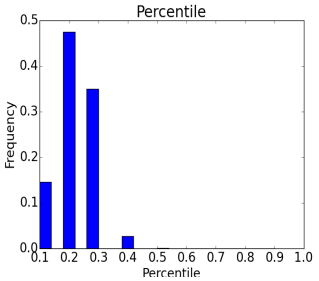


(l) State 1: 97th percentile.

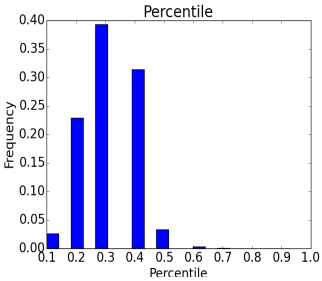
Figure 3: Transition probabilities for the English Wikipedia Workload.



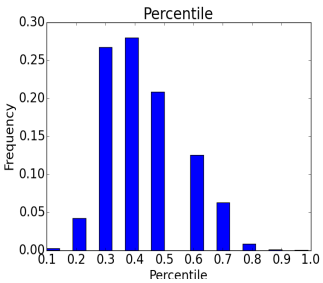
(a) State 1: 10th percentile.



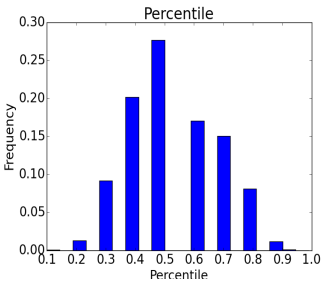
(b) State 2: 20th percentile.



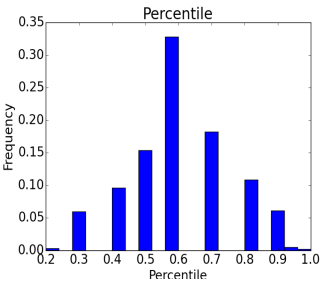
(c) State 3: 30th percentile.



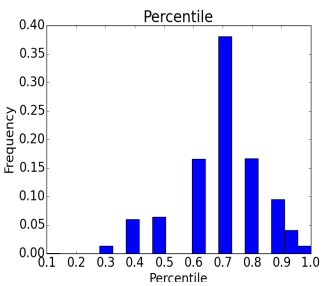
(d) State 4: 40th percentile.



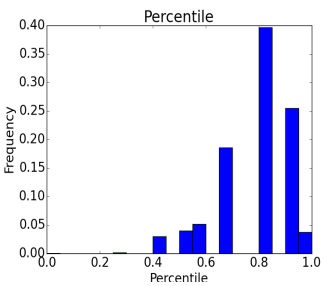
(e) State 5: 50th percentile.



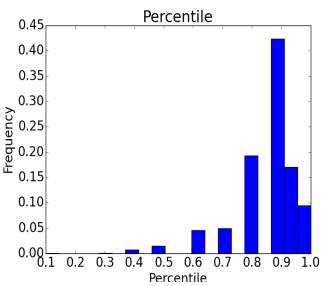
(f) State 6: 60th percentile.



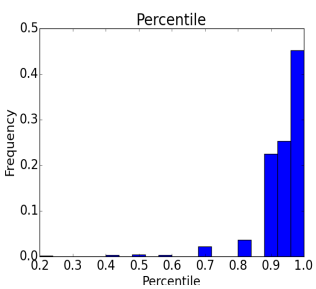
(g) State 7: 70th percentile.



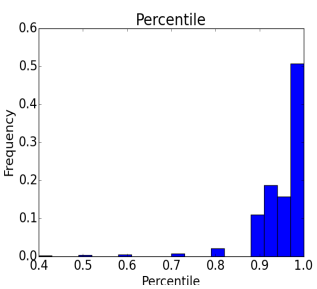
(h) State 8: 80th percentile.



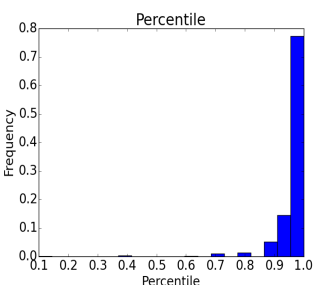
(i) State 9: 90th percentile



(j) State 10: 95th percentile.



(k) State 11: 99th percentile.



(l) State 1: 97th percentile.

Figure 4: Transition probabilities for the Swedish Wikipedia workload.

control and autoscaling using the Markov chain models discussed here and Markov Decision Process[2] [3].

References

[1] How to set up your own copy of wikipedia. Accessed: August, 2015, URL:http://www.extremetech.com/computing/114387-how-to-set-up-your-own-copy-of-wikipedia.

[2] Wikipedia statistics. Accessed: August, 2015, URL:http://stats.wikimedia.org/EN/Sitemap.htm.

[3] B. Chen and Y. Hong. Testing for the markov property in time series. *Econometric Theory*, 28(01):130–178, 2012.

[4] D. F. Galletta, R. M. Henry, S. McCoy, and P. Polak. When the wait isn't so bad: The interacting effects of website delay, familiarity, and breadth. *Information Systems Research*, 17(1):20–37, 2006.

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

[6] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[7] J. Nielsen. *Usability engineering*. Elsevier, 1994.

[8] J. Nielsen. Website Response Times. http://www.nngroup.com/articles/website-response-times/, 2010. [Online; accessed 19-August-2015].

[9] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[10] H. Tijms. *Understanding probability*. Cambridge University Press, 2012.

[2] This Section is part of a paper to be submitted

[3] Ahmed Ali-Eldin, Alessandro Papadopoulos, Karl-Johan Åstrom and Erik Elmroth, Admission Control of Cloud services based on Markov Decision processes (tentative title)