Modeling and Control of Server-based Systems

Dellkrantz, Manfred

2016

Document Version:
Publisher's PDF, also known as Version of record

Modeling and Control of Server-based Systems

Manfred Dellkrantz

LUND
UNIVERSITY

Department of Automatic Control

Lic. Tech. Thesis
ISRN LUTFD2/TFRT--3269--SE
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Holmbergs i Malmö AB.
Lund 2016

Abstract

When deploying networked computing-based applications, proper resource management of the server-side resources is essential for maintaining quality of service and cost efficiency. The work presented in this thesis is based on six papers, all investigating problems that relate to resource management of server-based systems. Using a queueing system approach we model the performance of a database system being subjected to write-heavy traffic. We then evaluate the model using simulations and validate that it accurately mimics the behavior of a real test bed. In collaboration with Ericsson we model and design a per-request admission control scheme for a Mobile Service Support System (MSS). The model is then validated and the control scheme is evaluated in a test bed. Also, we investigate the feasibility to estimate the state of a server in an MSS using an event-based Extended Kalman Filter. In the brownout paradigm of server resource management, the amount of work required to serve a client is adjusted to compensate for temporary resource shortages. In this thesis we investigate how to perform load balancing over self-adaptive server instances. The load balancing schemes are evaluated in both simulations and test bed experiments. Further, we investigate how to employ delay-compensated feedback control to automatically adjust the amount of resources to deploy to a cloud application in the presence of a large, stochastic delay. The delay-compensated control scheme is evaluated in simulations and the conclusion is that it can be made fast and responsive compared to an industry-standard solution.

Acknowledgements

First of all I would like to thank my children, Elias and Tilda, for constantly making my life a little less predictable, more interesting and fun, and of course their mother Hanna, for supporting me in my work and for being an awesome partner in parenthood. I also want to express my gratitude towards my parents Elisabeth and Lennart and my brother Gustaf for their support and for introducing me to the wonderful world of computers and coding.

I would also like to thank my supervisors Prof. Anders Robertsson, Prof. Maria Kihl and Prof. Karl-Erik Årzén. Your passion for the research problems and ability to navigate the academic jungle is really inspiring and I have learned a lot.

Many thanks also to my many coauthors to the papers we've produced over the years. Among them I would like to send a special thanks to my colleague Jonas Dürango who has spent countless hours with me, discussing flow models, control strategies, cloud applications, academics, TV series and video games. We've also shown empirically a number of times that it is possible for colleagues to share very small hotel rooms without getting on each other's nerves.

Thanks also to the LCCC research group Cloud/Embedded and the people in the Cloud group at Umeå University for many hours of interesting discussions and inspiring research presentations.

Of course, this work wouldn't have been possible without the help of the people that keep the department running; Pontus Andersson, Anders Blomdell, Lizette Borgeram, Anders Nilsson, Ingrid Nilsson, Mika Nishimura, Monika Rasmusson and Eva Westin. A special thanks to Leif Andersson, without your LaTeX wizardry the content of this book would have looked a lot worse. Of course I would also like to send thanks to the other people at the department, both seniors and PhD students, postdocs and guests, for making it an inspiring, interesting and fun work place.

Financial Support This work has been partly funded by the Swedish Research Council project "Cloud Control" VR CLOUD 2012-5908 and the Swedish Research Council grant VR-621-2010-5864. The author is also part of the LCCC Linnaeus and ELLIIT Excellence Centers.

Contents

1

Introduction

Resource management of computer systems, which has gained increased attention during the last few decades, was explored already in the late sixties. It is an essential mechanism to handle load disturbances such as resource outages, traffic surges and changes in user behavior. In this thesis we explore resource management of large computer systems that are at least indirectly facing human users.

The load such systems are subjected to is stochastic and varies on a few different time scales. On the very large time scale there is the long term evolution of the usage, for example, a specific service gradually growing in popularity or users that are following a trend in changing their usage pattern. The load typically also experiences seasonal variations, such as lows during the night, highs during the day and peaks around holidays. These variations are typically possible to predict to some extent.

But there are also changes in load which are impossible to predict, such as the traffic surge in a cellular network after a natural disaster or the rush to a news service after some political event. On June 25, 2009 CNN reported a five-fold increase in traffic and Twitter was forced to disable parts of its services due to the death of Michael Jackson[1]. A resource shortage can also have the opposite cause, for example that we suddenly lose parts of our resources due to, for example, a power outage.

Poorly managed resources can severely degrade the performance of a system with potentially large financial consequences. Resource management of these systems, based on measurements of the system states such as actual utilization and response times, is crucial for the optimization of operation cost and the guarantee of service level agreements during load surges, for example during marketing campaigns or various events. In the last decades the environmental effects of underutilizing computing resources have also come into the focus of research, since lowering the resource-demand lowers the power-consumption which in turn reduces the environmental impact.

[1] `http://web.archive.org/web/20160305092105/http://edition.cnn.com/2009/TECH/06/26/michael.jackson.internet/`

Therefore, the challenge is how to maintain adequate system performance while providing guarantees on system convergence and disturbance rejection. As for many other areas where control is used, resource management of computing systems comes down to collecting measurements, making educated estimations of what is causing these values and then acting to get the system to behave as desired.

2

Resource management of computer systems

The field of control of large computer systems comes with many challenges. In this chapter the challenges addressed in this thesis will be discussed. Each section ends with references to the papers in which the challenge is addressed.

2.1 Performance Models

Typically when employing control-schemes optimizing resource utilization, a model describing the entity that is to be controlled can provide useful insights into the properties of the system. Models can for example tell you what kind of control action is needed, how aggressively a controller can be tuned or where in the system a potential bottleneck can occur. In the case of resource management for computer systems providing some kind of service, you are typically interested in some sort of performance model. Such a model should capture the important aspects regarding the system considered, such as user behavior, application resource demands and the influence of the underlying infrastructure. Based on these factors, the model should give as output some measure of the performance of the system, such as user-perceived Quality of Service (QoS) or service revenue.

Common in the control of computer systems community is to use models based on queueing systems [Kleinrock, 1975]. Incoming work in the form of jobs are put in a queue awaiting to be serviced. A server takes jobs from the queue in some order, takes some time to process each job, and then sends a response. Since queueing systems are, in many aspects, hard to deal with analytically it is often considered enough to use continuous time, approximative fluid-models [Agnew, 1976; Rider, 1976; Wang et al., 1996]. These result in first-order nonlinear ODEs describing the relation from incoming load to queue length to response latency. Response latency (response time) is a commonly used metric for QoS.

In Paper I modeling of disk write-intensive database-traffic is investigated. In Paper II a queueing system-based model of a Mobile Service Support System (MSS)

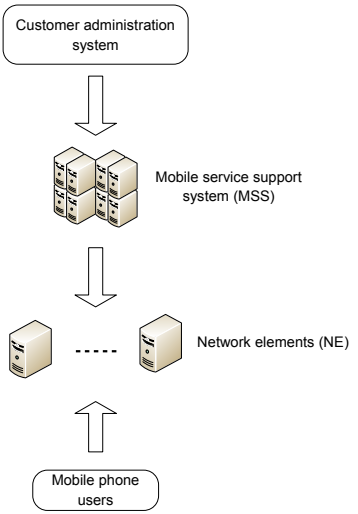

Figure 2.1 Illustration of the challenge in an MSS. NEs needs to handle loads from both the MSS and from actual users and can become overloaded.

is developed. In Paper III estimation of the queue length of a queueing system using an Extended Kalman Filter (EKF) is explored. In Paper VI a queueing system-based simulation model is used, and a delay-compensator using a fluid-approximation model is designed.

2.2 Mobile Service Support System

Parts of the work in this thesis is aimed at a commercial Mobile Service Support System (MSS) developed and produced by Ericsson AB. MSSs are used by network operators for all processing regarding new subscribers and services in the network. Each new subscriber or service requires processing and data storage in several Network Elements (NEs). Not only are the NEs geographically distributed, with different roles, but they typically also have different owners and manufacturers, making the network highly heterogeneous. The MSS presents to the operator and its business support systems a unified middleware where complex functions can be easily invoked. The software architecture is complex with several layers and distributed infrastructures, which means that specific parts of the system will not have complete knowledge of the interactions among other parts of the system.

The system architecture is illustrated in Figure 2.1. One request to the MSS from an upstream system normally results in a number of requests downstream out on the mobile network to several different NEs. An NE is usually a database storing subscriber and service data, for example, the Home Location Register (HLR). A

user ID, which needs to be fetched from one database, needs to be supplied in a query to another database to get the system consistent. In parallel to the changes and setups that the MSS performs, the network is also used by the end users. Services being set up by the MSS are queried by base stations and other systems requiring that information. In respect to the MSS, this traffic can be considered as unknown background traffic, in contrast to the known traffic flowing through the MSS.

The experience from deployed Ericsson systems shows that there can be problems with overloads in the NEs. The measurable load arriving from the MSS and the unknown (not directly measurable) load arriving from mobile users may interfere with each other, creating a race for resources that may lead to overload in a NE. When one NE becomes overloaded and unresponsive, this may result in the entire transaction requiring rollback to avoid in-consistencies in the network. Such a rollback may require manual work which is of course costly for the operator. To protect against such situations, traffic monitoring and control are crucial.

In Paper II we investigate a few different control-related problems in MSSs. In Paper III we design an observer for the state of an MSS.

2.3 Cloud Computing

There are many ways of defining what cloud computing is. Most definitions agree that cloud computing is the business model where a service provider leases IT-related resources of some kind to a service consumer. National Institute of Standards and Technology (NIST) defines cloud computing by specifying the following five requirements [Mell and Grance, 2009]:

On-demand self-service
: By this they mean that the consumers should themselves be in control of the amount of acquired resources.

Broad network access
: The service should be accessible over the Internet using appropriate, connected devices and software.

Resource pooling
: The available resources should be pooled by the provider. The pool of resources should be available for the consumers. This also implies that all resources in some sense are both location- and infrastructure-unaware. You don't know on what kind of hardware you run, or exactly where in the data center.

Rapid elasticity
: It should be possible to rapidly let go of, and acquire more resources as the needs of the consumer change.

Measured service

The amount of used resources should be metered by the provider. This metering information can also be made available to the consumer.

Even though this list of requirements is not always fulfilled for services that still are referred to as cloud services, the requirements gives a good idea of what the core of the concept cloud computing is.

Cloud computing is often divided into the three service models Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS). IaaS are services offering consumers on-demand computing infrastructure such as computers, data storage, memory or network. These resources are often virtualized. PaaS covers services which provide some sort of computing platform. This could for instance be programming language execution environments, databases or web servers. In the SaaS model application software is made available on-demand to its users. Examples of this includes Google Apps and Microsoft Office 365.

Commonly, the central part of the cloud services offered by providers is the virtual private servers. Examples of such services include Amazon EC2, Google Compute Engine and City Cloud. These services gives access to Virtual Machines (VMs), or "instances", which are run in a physical computer in a data center. Through either a web interface or an API, users can boot up new instances, and terminate those already running. New instances are of a user-specified instance type, each with its own amount of memory, CPU reservation and disk. The consumer is then billed for every machine time unit used, rates varying with the instance type and service provider.

Since the cloud computing model offers **On-demand self-service**, **Rapid elasticity** and **Resource pooling**, many SaaS services are built and deployed on top of IaaS or PaaS services. This allows for a start-up company launching a service to adjust their capacity with no up-front payment. And in the event the service grows in popularity, the capacity can be easily increased.

For example Netflix has been very open with their usage of the Amazon cloud computing platform for building their services and is therefore commonly used as an use case in the research community. They rent capacity from Amazon, both in form of VMs (through EC2) and storage space (through the block storage service S3). On this they deploy many smaller services which combine to form their SaaS service of streaming movies and TV series to users all over the world. Examples of these smaller services could be searching, storing ratings or computing recommendations based on previously watched content.

When a user makes a request to an online service, such as opening the front page in Netflix, there is a server responding from some data center. Typically, for large services, such a request involves a multitude of sub-services handling different parts of the presentation of the complete service. This puts some load on the resources running the service. If the service is already under heavy load, the request will be queued up for a long time, waiting to be served. Long delays in responses (response

times) will lower the user-perceived QoS, in other words customers that have to wait gets unhappy [Nah, 2004].

The flexible resource model in cloud computing pose a challenge in adapting the right amount of resources, so as to avoid wastefulness as well as decreased QoS. *Autoscaling* refers to making resource allocation decisions autonomically. Autoscaling has previously been approached with feedback control. However a lot of the previous work has ignored the fact that, even though we have "rapid elasticity", scaling up still typically takes in the order of minutes [Mao and Humphrey, 2012], which can cause problems for feedback controllers. Autoscalers used in production typically uses threshold controllers with a cooldown period [Amazon, 2014; Google, 2014; Rackspace, 2014]. Threshold controllers observe a metric over time and if it has been over a predefined threshold for some amount of time the controller acts. After the controller has acted it gets put in a cooldown state for one cooldown period in which it is prevented from acting. This is to prevent the controller from taking more decisions while the previous decision has not yet come into effect.

Paper VI presents a delay-compensating controller able to control virtual machine deployment even in the presence of long, asymmetric, stochastic resource reconfiguration delays.

2.4 Brownout

Since deploying applications on a very large scale requires a lot of hardware, virtual or physical, such applications must be able to handle hardware failures. Single computers rarely fail, but in a ten thousand node cluster, failure is something that happens daily. Therefore software built for running on this scale is typically built from the beginning to withstand sudden loss of parts of the hardware set [Hamilton et al., 2007]. However failures on this scale can often also be correlated in both space and time, and this can lead to a temporarily under-provisioned system [Gallet et al., 2010; Yigitbasi et al., 2010].

A *flash-crowd* is another way a service can be temporarily under-provisioned. A flash-crowd is when a sudden burst of traffic comes to a certain service. Flash crowds can have many different causes, but typically it happens when the victim service is mentioned or linked to in some other relatively popular channel, generating a flood of visitors. A special case of flash-crowd is referred to the Slashdot effect, which got its name from the website Slashdot which posts news about science and technology. The effect is the overload that happens to small independent websites when they get linked to in a post on Slashdot.

One attempt to remedy temporary under-provisioning was introduced in brownout [Klein et al., 2014; Maggio et al., 2014]. The essence of brownout is that a service is extended with capabilities to serve content in two different modes. In one mode it serves only service content which is marked as mandatory while in the second mode it serves both the mandatory and some content marked as optional. The

system then measures the response times of the users and adjusts what percentage of the requests should be served both mandatory and optional content. The idea is that the system should be able to greatly lower its resource requirement by serving less optional content, and thereby handling a temporary capacity shortage.

As an example we can consider the product page on an e-commerce website. Here, the product description would have to be considered mandatory content, since a visitor wouldn't consider the response complete and useful without the description of the product. However, a section describing related products would be part of the optional content. The user doesn't necessarily need it, but it makes the other products on the web site more accessible to the user and increases revenue [Fleder et al., 2010].

The previous work done on brownout was focused on the single server case, where we only have one computer working to service users. When a service requires more resources than can be provided by a single server, you typically deploy multiple worker computers and use a load balancer to spread the load evenly among the servers. Many load balancing algorithms base their decision on the response times of the workers. Since the aim of brownout is to keep the response times at or below a certain set point, response time-based load balancers would not be able to see any difference between an overloaded worker, only serving mandatory content, and a fully loaded worker serving full content.

In Paper IV and Paper V load balancing algorithms for brownout-enabled applications are investigated.

3

Publications

This licentiate thesis is based on the following six papers. They are listed below with a description of their contribution to the research field and a description of the contributions of the author. The formatting of the papers have been adapted to the print format of the thesis.

Paper I

Dellkrantz, M., M. Kihl, and A. Robertsson (2012). "Performance modeling and analysis of a database server with write-heavy workload". In: *2nd European Conference on Service-Oriented and Cloud Computing*. Bertinoro, Italy.

In this paper we investigate the performance dynamics of a MySQL/InnoDB database server with write-heavy workload. The main objective of our investigation was to understand the system dynamics due to the buffering of disk operations that occurs in database servers with write-heavy workload. We characterize the traffic and its periodic anomalies caused by flushing of the buffer. Further, we present a performance model for the response time of the requests and show how this model can be configured to fit with actual database measurements.

The author came up with the model and how to find its parameters, set up the test bed, performed the experiments and was the main author of the manuscript.

Paper II

Amani, P., B. Aspernäs, K. J. Åström, M. Dellkrantz, M. Kihl, G. Radu, A. Robertsson, and A. Torstensson (2012). "Application of control theory to a commercial mobile service support system". *International Journal On Advances in Telecommunications* **5**:3&4. URL: `http://www.iariajournals.org/telecommunications/`.

In this paper we identify and explore some important control challenges for the Ericsson Mobile Service Support System which handles setup of new subscribers and services in a mobile network. Further, we present analysis and experiments showing some advantages of proposed solutions. First, we develop a load-dependent server model for the system, which is validated in testbed experiments. Further, we propose a control design based on the model, and a method for estimation of response times and arrival rates. The main contribution of this paper is that we show how control theory methods and analysis can be used for commercial telecom systems. Parts of our results have been implemented in commercial products, validating the strength of our work.

The author set up and operated the test bed on which the load-adaptive controller was tested. The author also ran simulations and collected data for the event-based state estimator.

Paper III

Dellkrantz, M., M. Kihl, A. Robertsson, and K. J. Åström (2012). "Event-based response time estimation". In: *7th International Workshop on Feedback Computing*. San Jose, California, USA.

In this paper estimation techniques, suitable for support systems for mobile phone systems, are explored. These systems are complex queueing systems with large databases. The traffic generated by users and system administrators changes rapidly, some loads can be measured while others cannot. Attempts to capture all the details of the system give models that are not suitable for on-line control. Estimators based on continuous flow models with event based measurements are designed using an EKF. The estimators are compared with simple-data based estimators.

The author ran the simulations and came up with the idea to consider the fundamental limitations of the estimation.

Paper IV

Dürango, J., M. Dellkrantz, M. Maggio, C. Klein, A. V. Papadopoulos, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén (2014). "Control-theoretical load-balancing for cloud applications with brownout". In: *53rd IEEE Conference on Decision and Control*. Los Angeles, CA.

In this paper, we present novel load-balancing strategies, specifically designed to support brownout-enabled workers. They base their decision not on response time, but on user experience degradation. We implemented our strategies in a self-adaptive application simulator, together with some state-of-the-art solutions. Results obtained in multiple scenarios show that the proposed strategies bring significant improvements when compared to the state-of-the-art ones.

The author implemented the equal-thetas algorithm and contributed equally to its idea and tuning together with J. Dürango.

Paper V

Klein, C., A. V. Papadopoulos, M. Dellkrantz, J. Dürango, M. Maggio, K.-E. Årzén, F. Hernández-Rodriguez, and E. Elmroth (2014). "Improving cloud service resilience using brownout-aware load-balancing". In: *33rd IEEE International Symposium on Reliable Distributed Systems*. Nara, Japan.

In this paper we propose two novel brownout-aware load balancing algorithms. To test their practical applicability, we extended the popular lighttpd web server and load-balancer, thus obtaining a production-ready implementation. Experimental evaluation shows that the approach enables cloud services to remain responsive despite cascading failures. Moreover, when compared to Shortest Queue First, believed to be near-optimal in the non-adaptive case, our algorithms improve user experience by 5%, with high statistical significance, while preserving response time predictability.

The author contributed equally to the idea of making the controllers event-based and queue-based together with J. Dürango. The author also implemented the equal-thetas algorithm in the lighttpd load balancer used in the conducted experiments.

Paper VI

Dellkrantz, M., J. Dürango, A. Robertsson, and M. Kihl (2015). "Model-based deadtime compensation of virtual machine startup times". In: *10th International Workshop on Feedback Computing*. Seattle, WA, USA.

This paper presents a delay-compensator inspired by the Smith predictor. The compensator allows one to close a simple feedback loop around a cloud application with a large, time-varying delay, preserving the stability of the controlled system. It also makes it possible for the closed-loop system to converge to a steady-state, even in presence of resource quantization. The presented approach is compared to a threshold-based controller with a cooldown period, that is typically adopted in industrial applications.

The author came up with the idea for the compensator structure, both for delay compensation and quantization compensation, wrote the simulation and was the main author of the manuscript.

Not included

The following publications co-authored by the author of this thesis covers related work but are not included in this thesis.

Kihl, M., P. Amani, A. Robertsson, G. Radu, M. Dellkrantz, and B. Aspernäs (2012). “Performance modeling of database servers in a telecommunication service management system”. In: *7th International Conference on Digital Telecommunications*, pp. 124–129.

Papadopoulos, A. V., C. Klein, M. Maggio, J. Dürango, M. Dellkrantz, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén (2016). “Control-based load-balancing techniques: analysis and performance evaluation via a randomized optimization approach”. *Control Engineering Practice*. Accepted for publication.

Bibliography

Here follows the bibliography for Chapter 2. The bibliographies of the included papers can be found at the end of their respective re-print.

Agnew, C. E. (1976). "Dynamic modeling and control of congestion-prone systems". *Operations research* **24**:3, pp. 400–419.

Amazon (2014). *Auto scaling concepts — Amazon Web Services documentation*. Accessed: 2014-08-27. URL: https://web.archive.org/web/20140729191545/http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/AS_Concepts.html.

Fleder, D. M., K. Hosanagar, and A. Buja (2010). "Recommender systems and their effects on consumers: the fragmentation debate". *EC* **229**, p. 230.

Gallet, M., N. Yigitbasi, B. Javadi, D. Kondo, A. Iosup, and D. H. J. Epema (2010). "A model for space-correlated failures in large-scale distributed systems". In: *Euro-Par*. DOI: 10.1007/978-3-642-15277-1_10.

Google (2014). *Google compute engine autoscaler — Google Cloud Platform Documentation*. Accessed: 2014-12-01. URL: https://web.archive.org/web/20141201094332/https://cloud.google.com/compute/docs/autoscaler/.

Hamilton, J. R. et al. (2007). "On designing and deploying Internet-scale services." In: *LISA*. Vol. 7, pp. 1–12.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014). "Brownout: building more robust cloud applications". In: *36th International Conference on Software Engineering (ICSE)*. Hyderabad, India.

Kleinrock, L. (1975). *Theory, Volume 1, Queueing Systems*. Wiley-Interscience. ISBN: 0471491101.

Maggio, M., C. Klein, and K.-E. Årzén (2014). "Control strategies for predictable brownouts in cloud computing". In: *19th IFAC World Congress*. Cape Town, South Africa.

Mao, M. and M. Humphrey (2012). "A performance study on the VM startup time in the cloud". In: *IEEE 5th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 423–430.

Mell, P. and T. Grance (2009). "The NIST definition of cloud computing". *National Institute of Standards and Technology* **53**:6, p. 50.

Nah, F. F.-H. (2004). "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour & Information Technology* **23**:3, pp. 153–163.

Rackspace (2014). *How auto scale cooldowns work — Rackspace Knowledge Center*. Accessed: 2014-11-17. URL: https://web.archive.org/web/20141117122211/http://www.rackspace.com/knowledge_center/article/how-auto-scale-cooldowns-work.

Rider, K. L. (1976). "A simple approximation to the average queue size in the time-dependent M/M/1 queue". *Journal of the ACM (JACM)* **23**:2, pp. 361–367.

Wang, W.-P., D. Tipper, and S. Banerjee (1996). "A simple approximation for modeling nonstationary queues". In: *15th Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation*. Vol. 1. IEEE, pp. 255–262.

Yigitbasi, N., M. Gallet, D. Kondo, A. Iosup, and D. H. J. Epema (2010). "Analysis and modeling of time-correlated failures in large-scale distributed systems". In: *GRID*. DOI: 10.1109/GRID.2010.5697961.

Paper I

Performance Modeling and Analysis of a Database Server with Write-Heavy Workload

Manfred Dellkrantz Maria Kihl Anders Robertsson

Abstract

Resource-optimization of the infrastructure for service oriented applications require accurate performance models. In this paper we investigate the performance dynamics of a MySQL/InnoDB database server with write-heavy workload. The main objective of our investigation was to understand the system dynamics due to the buffering of disk operations that occurs in database servers with write-heavy workload. In the paper, we characterize the traffic and its periodic anomalies caused by flushing of the buffer. Further, we present a performance model for the response time of the requests and show how this model can be configured to fit with actual database measurements.

Originally published in 2nd European Conference on Service-Oriented and Cloud Computing, Bertinoro, Italy, 2012. Reprinted with permission of Springer.

1. Introduction

The processing and control of service-oriented applications, as web applications, mobile service management systems, media distribution applications, etc., are usually deployed on an infrastructure of server clusters. The rate at which the requests arrive can vary heavily both during a single day and during longer periods, due to user behavior patterns. Scaling for the worst traffic peaks can be expensive though and will result in most of the capacity being unused most of the time. Capacity planning and resource optimization is therefore needed, which require the design of accurate performance models that capture the system dynamics in high loads.

Previous work on control systems for service-oriented applications and systems has mainly focused on applications with CPU-intensive workload, for example web server systems and databases with read-only requests. For CPU-intensive workloads previous work has shown that the performance dynamics are accurately captured by a single server queue model, see for example, [Cao et al., 2003] and [Kihl et al., 2011]. However, for applications including large databases (too large to store in main memory), hard drive dynamics will influence the performance dynamics in high loads. Typically the application will need to read data from disk on every database read. Write operations are however often buffered in the server to make them more efficient. For example, in [Rago et al., 2013] the authors examine different buffering/caching techniques for use with NFS (Network File System).

Writing to persistent media is often a slow process which should be avoided if possible. Further, writing performance is also affected by certain rules of locality. For example, writing sequential data to a hard drive can be many times faster than writing to random sectors. By buffering writes and completing them in sequential order, the writes are executed more efficiently. However, when the buffered writes are actually flushed to disk, the response times of the normal flow of traffic are heavily influenced. The server becomes occupied by work other than that of the normal flow of requests. Therefore, the system dynamics of server systems with write-heavy workload cannot be captured with the single server queueing models proposed for CPU-intensive workload.

In this paper we examine write-heavy workload on a MySQL database server using the engine InnoDB. The database is stored on a magnetic hard drive which results in the database server having to employ heavy buffering to speed up the writes. In the paper, the characteristics of write-heavy workload is examined. We develop a model and configure the model parameters using experiments in our testbed. We show in experiments that the model accurately captures the periodic anomalies that occur when the system needs to empty the buffer.

In Section 2 we present the lab environment used for the database measurements. In Section 3 we characterize the database traffic. In Section 4 we present the model developed for the traffic and discuss how to configure it. We also validate the model with lab measurements.

2. System Description

In this paper, we investigate the dynamics of database servers with write-heavy workload. The models and methods proposed in the paper are based on the results from experiments in our testbed. In this section, we first give an introduction to dirty page caching, which is used in many operative systems and database systems to improve the latencies when writing to disk. Further, we describe our testbed.

2.1 Page Cache

One common way to implement write-buffering is using a page cache. The storage is divided into fixed size pages. When data is written, the page being written to is first read from storage and then changed in memory and marked as dirty. Dirty pages are then kept in memory for some time before it is written back to disk and marked clean.

MySQL has several different storage engines, among them MyISAM and InnoDB. MyISAM has no built in cache for data. Instead it relies on the page caching features of the operating system. In this paper, we have used the storage engine InnoDB, which has its own system of pages which are buffered in the so called *buffer pool*. Pages are written to and read from disk directly using one of several methods for directly accessing the block storage device, bypassing the operating system page cache. The InnoDB engine tries to estimate the speed of the block device and the rate at which new pages are made dirty and from that it calculates how often and how many dirty pages need to be written to disk.

2.2 Testbed

We have used an experimental testbed. The testbed consists of one computer acting as traffic generator, and one database server. The computers were connected with a standard Ethernet switch.

The traffic generator was executed on an AMD Phenom II X6 1055T at 2.8 GHz with 4 GB main memory. The operating system was 64-bit Ubuntu 10.04.4 LTS. The traffic generator was implemented in Java, using the JDBC MySQL connector. The traffic generator used 200 working threads and generated MySQL queries according to a Poisson process with average rate λ queries per second. The behavior of the traffic generator was validated in order to guarantee that it was not a bottleneck in the experiments.

The database server had a 2.0 GHz Celeron processor and 256 MB main memory. The database files are on the system disk which is a standard 3.5" hard drive. It runs the 32-bit version of Ubuntu 10.04.4 LTS (Linux 2.6) and MySQL Server 5.1.41. The InnoDB engine was configured with 16 MB of buffer pool.

The structure of the relations in the database comes from the scalable Wisconsin Benchmark [DeWitt, 1991] and it has $n = 10^7$ tuples. The structure of the queries used all follow the following pattern:

```
UPDATE <relation> SET unique3=? WHERE unique1=?;
```

The question marks are replaced with uniformly distributed pseudo-random integers in the interval $[0, n[$. This query changes the value of one of the integer attributes of a random tuple.

3. Performance Characterization

In order to investigate the dynamics of a database server with write-heavy workload, we performed a series of experiments in our testbed presented in Section 2. In all the experiments, all requests included a MySQL UPDATE query, causing the system to write one database element to disk. Figure 1 illustrates the system behavior during an experiment where the average arrival rate, λ, was 25 requests per second. The figure shows that the system periodically have to pause the normal work and instead focus on the buffered dirty pages for some time. While the normal response times are below 0.2 seconds, response times of up to one second occur, because of these pauses. The number of requests that have these high response times are affected by the fact that requests are sent and queued up, even when the server is busy with the dirty pages.

The average response time as a function of the number of concurrent jobs is from now on referred to as the N/T graph. The throughput as a function of the number of concurrent jobs inside the server at all times is from now on referred to as the N/P graph. The N/T and N/P graphs for our system are shown in Figure 2.

It can be seen in the N/P graph that for a very small number of concurrent requests (up to 10), the throughput is much lower than for a higher number of concurrent requests. This is likely (to some extent, at least) because of network delays and buffering in lower protocol layers.

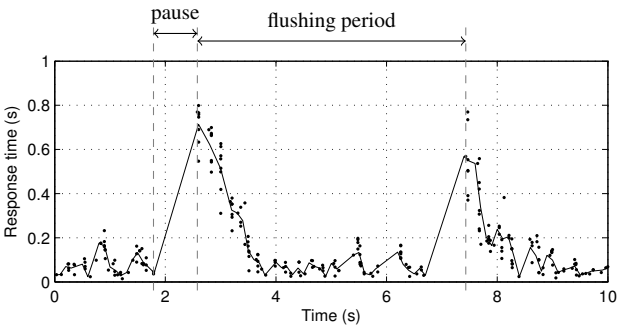

Figure 1. Response time graph of the InnoDB system, UPDATEs only, with constant Poisson-traffic, $25/s$

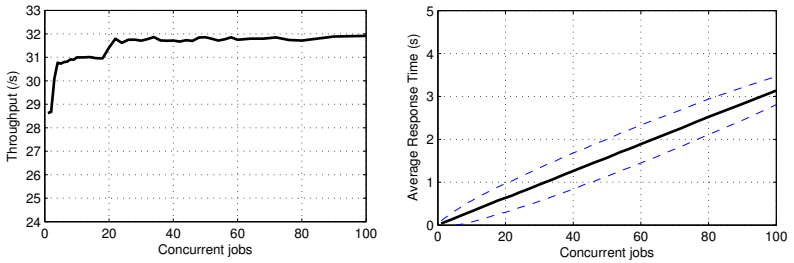

Figure 2. N/P graph (left) and N/T graph (right) of the InnoDB system, UPDATEs only. Every point was run for 900 seconds.

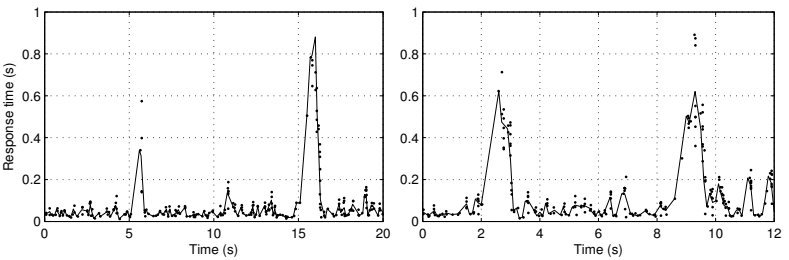

Figure 3. Response time graph of the InnoDB system, UPDATEs only, with constant Poisson-traffic, $12.5/s$ (left), $18.75/s$ (right)

During high loads, the dirty page cache will be written to disk periodically. The period between two occurrences of disk writing, called the flushing period, depends on the arrival rate. As can be seen in Figure 1, an arrival rate of 25 requests per second results in a flushing period of approximately 5 seconds. Figure 3 shows the response times during an experiment with an average arrival rate of 12.5 requests per second, which results in a flushing period of approximately 10 seconds and an experiment with an average arrival rate of 18.75 requests per second, which results in a flushing period of approximately 7 seconds. These experiments show that the period between flushes of the buffer is inversely proportional to the arrival rate, since

$$25 \cdot 5 \approx 12.5 \cdot 10 \approx 18.75 \cdot 7 \tag{1}$$

4. Performance Model

In this section, we describe our proposed performance model, which captures the dynamics of our system.

4.1 Model Description

We propose a queuing network model shown in Figure 4. The model consists of three parts, a *Network delay (ND)*, a *Job queue (JQ)*, and a *Dirty page buffer (DPB)*. The ND is used to model the reduced throughput at very low numbers of concurrent requests. After passing the ND, requests enter the JQ. As requests are processed by the server, the user is acknowledged and one dirty page equivalent is placed in the DPB. The DPB has a fixed maximum size and when that is reached the server stops processing requests in the JQ and starts to process dirty pages from the DPB instead until the DPB is empty. When the DPB is empty, the server switches back and continues to work on the JQ.

As a request enters the server it is assigned a processing time. Our experiments have shown that the processing time for a request in the JQ and the processing time for one dirty page equivalent (T_{proc} and T_{dp}, respectively) can be modeled by an exponential distribution. Further, the time each request spends in the ND (T_{nd}) can be modeled as a sum of a constant and an exponentially distributed random number.

Further, the maximum size of the DPB is denoted DPB_{max} and it is a constant integer number. The maximum length of the DPB and one dirty page equivalent per request determine the inverse proportionality between flush period time and arrival rate shown in Equation (1).

4.2 Parameter Configuration

The model has the following parameters which must be configured:

$$T_{nd} \text{ distribution}, \mathrm{E}\left[T_{proc}\right], \mathrm{E}\left[T_{dp}\right], DPB_{max}$$

The maximum capacity of the DPB can be determined by measuring the period of flushes, p, for some high traffic with throughput P. Since p determines how often the DPB needs to be flushed and P determines how fast new dirty pages are put into the DPB, the max length of the DPB is $DPB_{max} = P \cdot p$.

By examining some experiment with high number of concurrent requests, a lower limit on the duration of the pause in processing ($\min(T_{pause})$) can be determined. By measuring the time between request departures and filtering out those that are $> \min(T_{pause})$, an average on the pause duration (T_{pause}) can be estimated. From these results, the mean of the dirty page processing time is given by $\mathrm{E}\left[T_{dp}\right] = T_{pause} \cdot {DPB_{max}}^{-1}$.

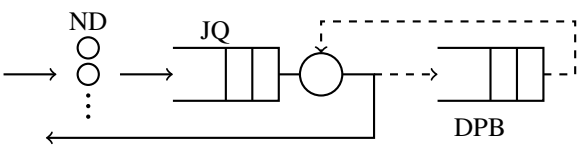

Figure 4. The Model

Table 1. Fitted model parameters

T_{proc}	$\text{Exp}(0.0269)$
T_{nd}	$0.0025 + \text{Exp}(0.00049)$
DPB_{max}	111
T_{dp}	$\text{Exp}(0.00433)$

With the knowledge of T_{dp} and the throughput (P) when keeping high number of concurrent requests, the average processing time T_{proc} can be determined. By assuming that the server is always busy, the throughput can be assumed to be inversely proportional to the total processing time spent on every request. Since the server spends a total of $T_{proc} + T_{dp}$ time on every request, the average for the processing time is given by $\text{E}[T_{proc}] = P^{-1} - \text{E}\left[T_{dp}\right]$.

The distributions used for the network delay (T_{nd}) are determined by performing an experiment keeping one concurrent request in the system. The response times T are measured. Since the total response time of one single request is the sum of the network delay, the processing time plus that it has a probability of ${DPB_{max}}^{-1}$ to get $DPB_{max} \cdot T_{dp}$ added, the average network delay is given by $\text{E}[T_{nd}] = \text{E}[T] - \text{E}[T_{proc}] - \text{E}\left[T_{dp}\right]$.

4.3 Model Validation

In order to validate the proposed model, we developed a discrete-event simulation program, written in Java. By using the configuration method described in Section 4.2, we can conclude that the values in Table 1 make a good fit for our database server described in Section 2.

In Figure 5, the cumulative distribution function of the response times from an experiment with arrivals following the Poisson process with an average rate of

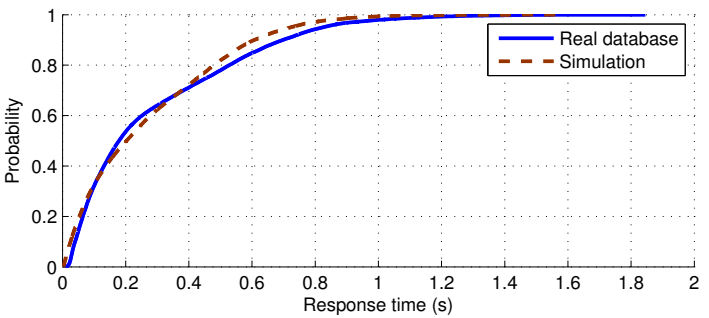

Figure 5. Cumulative distribution function of response times from InnoDB database system, and the proposed model. Traffic is generated with a Poisson process with average 28 requests per second.

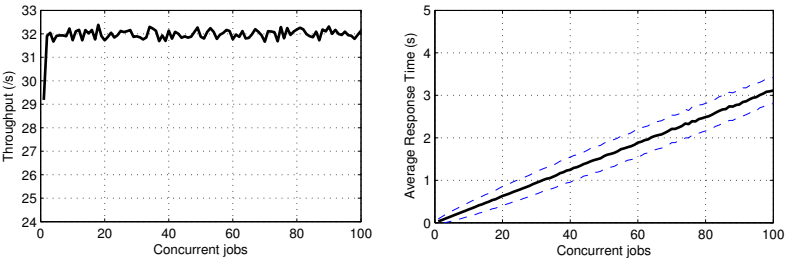

Figure 6. N/T (top) and N/P (bottom) graph from the simulation of the model. Every number of parallel jobs was run for 900 seconds.

28 requests per second, are shown. One graph shows the results from a testbed experiment and one graph shows the results from the discrete-event simulation of the model. As can be seen in the graphs, the distribution of response times in the model fits accurately with the database experiment.

Further, the N/T and N/P graphs for the simulation is shown in Figure 6. These graphs can be compared with the graphs of the corresponding experiments, shown in Figure 2. The graphs show that the proposed model fits well with the real system.

5. Conclusions

Many service-oriented applications use database servers for storing data. When the applications have a workload that writes to a database stored on hard drives, disk writing optimizations introduce performance dynamics that may be difficult to monitor and control. Traditional queuing system models do not suffice when the response times show these periodic anomalies. In this paper, we have developed a performance model based on queueing systems for database servers with write-heavy workload. We validate our model using experiments in a testbed.

Acknowledgment

This work has been partly funded by the Lund Center for Control of Complex Engineering Systems (LCCC) and the Swedish Research Council grant VR 2010-5864.

References

Cao, J., M. Andersson, C. Nyberg, and M. Kihl (2003). “Web server performance modeling using an M/G/1/K* PS queue”. In: *10th International Conference on Telecommunications*. Vol. 2. IEEE, pp. 1501–1506.

DeWitt, D. J. (1991). “The Wisconsin benchmark: past, present, and future”. In: *The Benchmark Handbook*, pp. 119–165.

Hsu, W. W., A. J. Smith, and H. C. Young (2001). “I/O reference behavior of production database workloads and the TPC benchmarks — an analysis at the logical level”. *ACM Transactions on Database Systems* **26**:1, pp. 96–143.

Kamra, A., V. Misra, and E. M. Nahum (2004). “Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites”. In: *12th IEEE International Workshop on Quality of Service*. IEEE, pp. 47–56.

Kihl, M., A. Robertsson, M. Andersson, and B. Wittenmark (2008). “Control-theoretic analysis of admission control mechanisms for web server systems”. *World Wide Web* **11**:1, pp. 93–116.

Kihl, M., G. Cedersjö, A. Robertsson, and B. Aspernäs (2011). “Performance measurements and modeling of database servers”. In: *6th International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*.

Kleinrock, L. (1975). *Theory, Volume 1, Queueing Systems*. Wiley-Interscience. ISBN: 0471491101.

Liu, X., J. Heo, L. Sha, and X. Zhu (2006). “Adaptive control of multi-tiered web applications using queueing predictor”. In: *10th IEEE/IFIP Network Operations and Management Symposium*, pp. 106–114.

Rago, S., A. Bohra, and C. Ungureanu (2013). “Using eager strategies to improve NFS I/O performance”. *International Journal of Parallel, Emergent and Distributed Systems* **28**:2, pp. 134–158.

Paper II

Application of Control Theory to a Commercial Mobile Service Support System

Payam Amani Bertil Aspernäs Karl Johan Åström
Manfred Dellkrantz Maria Kihl Gabriela Radu
Anders Robertsson Andreas Torstensson

Abstract

The Mobile Service Support system (MSS), which Ericsson AB develops, handles the setup of new subscribers and services into a mobile network. Experience from deployed systems show that traffic monitoring and control of the system will be crucial for handling overload situations that may occur at sudden traffic surges. In this paper we identify and explore some important control challenges for this type of systems. Further, we present analysis and experiments showing some advantages of proposed solutions. First, we develop a load-dependent server model for the system, which is validated in testbed experiments. Further, we propose a control design based on the model, and a method for estimation of response times and arrival rates. The main contribution of this paper is that we show how control theory methods and analysis can be used for commercial telecom systems. Parts of our results have been implemented in commercial products, validating the strength of our work.

Originally published in International Journal On Advances in Telecommunications, vol.5, no. 3 & 4, pp. 204-215, Dec. 2012. Reprinted with permission.

1. Introduction

Resource management of computer systems, which has gained increased attention during recent years, was explored already in the late 60's [Brawn and Gustavson, 1968; Crocus, 1975]. It is an essential mechanism to handle load disturbances such as traffic surges and changes in user behavior. Poorly managed resources can severely degrade the performance of a system with potentially large financial consequences.

The work presented in this paper is motivated by a commercial Mobile Service Support System (MSS), developed and produced by Ericsson AB. Mobile Service Support Systems are used by the network operators for all processing regarding new subscribers and services in the network. Each new subscriber or service requires processing and data storage in several network nodes. The systems are in general multi-tier systems, implemented as distributed server clusters, where web and application servers process the incoming requests and database servers are used for data storage. The resource management of these systems, based on measurements of the system states such as actual utilization and response times, is crucial for the optimization of operation cost and the guarantee of service level agreements during load surges, for example during marketing campaigns or various events.

Therefore, the challenge is how to control system performance while providing guarantees on convergence and disturbance rejection. The solution is based on *dynamic control schemes*, which monitors the systems and provides actions when needed. Several types of resource–management mechanisms have been proposed and evaluated in the literature. In larger computer systems, *load balancing* is performed in order to distribute the demand for resources uniformly over a number of resource units (computers, CPUs, memory, etc.), thus avoiding the case that among the nodes with similar functionalities some are under-utilized while others are overloaded [Diao et al., 2005; Fu et al., 2006]. During overload periods, when more resources are requested than are available, *admission control* mechanisms reduce the load to the system by blocking or delaying some of the requests [Kihl et al., 2008; Chen et al., 2003; Liu et al., 2006; Liu et al., 2006]. For Internet applications, virtualized server systems can be used to divide physical resources into a number of separated platforms where different web applications are allowed to operate without affecting one another. *Dynamic resource allocation* between the virtualized platforms serves as a new and easy way to perform resource optimization on web server systems [Kjær et al., 2009; Xu et al., 2006; Wang et al., 2007]. In the last years, the field of *power and energy management* has become important. Large software systems have high energy consumption, which means that dynamic resource optimization of these systems may considerably lower the operating costs for the network operators [Bianchini and Rajamony, 2004; Claussen et al., 2009; Horvath et al., 2007; Elnozahy et al., 2003].

However, all optimization techniques require accurate performance models of the involved computing systems. The operation region is mainly high traffic load

scenarios, which means that the computing systems show non-linear dynamics that needs to be characterized accurately [Kihl et al., 2003]. A software system is basically a network of queues, as examples, the CPU ready queue, semaphore queues, socket queues, and I/O device queues, which store requests in waiting of service in the processors. Therefore, queuing models can be used when describing the dynamic behavior of server systems [Brawn and Gustavson, 1968; Dilley et al., 1998; Menascé and Almeida, 2002; Mei et al., 2001].

The concept of Load-Dependent Server (LDS) models, in which the response time of the jobs in the system is a function of the service time of the jobs and current number of jobs waiting to be served has, to the best of our knowledge, firstly been introduced in [Perros et al., 1992]. In [Perros et al., 1992; Rak and Sgueglia, 2010; Curiel and Puigjaner, 2001], standard benchmarks were used for workload generation and also regression models to capture the system dynamics. In [Mathur and Apte, 2004], a queuing network model which represents the load dependent behavior of the LDS was presented and validated with simulations. In [Leung, 2002], a theoretical analysis of the D/G/1 and M/G/1 models with load dependency assumptions was presented.

In this paper, we investigate solutions to some important control challenges identified for the commercial MSS developed by Ericsson AB. We present a load-dependent server model, which is validated in experiments. The model has been previously published in [Kihl et al., 2012]. Further, we extend [Kihl et al., 2012] by proposing and validating an admission control mechanism based on a load-adaptive controller. A modified version of the controller has been implemented in the Ericsson product. Finally, we show how extended Kalman filters can be used for estimating the response times and arrival rates in the system.

The paper is organized as follows. In Section 2, the Ericsson product is described and the control challenges identified for the system are presented. In Section 3, the testbed used for some of the experiments is described. In Section 4, the load-dependent server model is presented and validated. In Section 5, the load-adaptive controller is presented and experiments validating its performance are described. In Section 6, our work on response time estimation based on extended Kalman filters is presented. Finally, in Section 7, some conclusions are presented.

2. System and Problem Description

The Mobile Service Support system (MSS), which Ericsson AB develops, handles the setup of new subscribers and services into a mobile network. It presents to the operator and its business support systems a unified middleware where complex functions, such as setting up a new subscriber or modifying services for an existing subscriber, can be easily invoked. The software architecture is complex with several layers and distributed infrastructures, which means that specific parts of the system will not have complete knowledge of the interactions among other parts of the system.

2.1 System architecture

The system architecture is illustrated in Figure 1. One request to the MSS from an upstream system normally results in a number of requests downstream out on the mobile network to several different network elements (NEs). A network element is usually a database storing subscriber and service data, for example, the Home Location Register (HLR). A user id, which needs to be fetched from one database, needs to be supplied in a query to another database to get the system consistent.

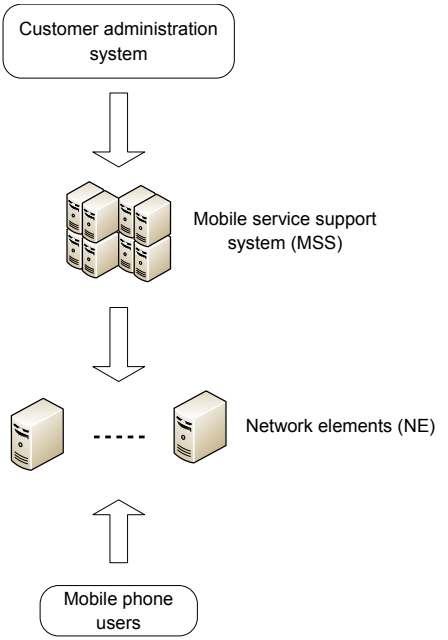

Figure 1. Mobile service support system (MSS).

In parallel to the changes and setups that the MSS performs, the network is also used by the end users. Services being set up by the MSS are queried by base stations and other systems requiring that information. In respect to the MSS, this traffic can be considered as unknown background traffic, in contrast to the known traffic flowing through the MSS.

2.2 Control challenges

The experience from deployed Ericsson systems shows that there can be problems with overload in the NEs. The measurable load arriving from the MSS and the unknown (not directly measurable) load arriving from mobile users may interfere with each other, creating a race for resources that may lead to overload in a NE. When

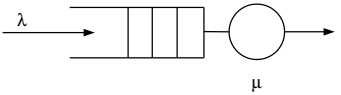

Figure 2. M/M/1 model.

one NE becomes overloaded and unresponsive, this may result in the entire transaction requiring rollback to avoid in-consistencies in the network. Such a rollback may require manual work which is of course costly for the operator.

To protect against such situations, traffic monitoring and control are crucial. In cooperation with Ericsson AB, some important control challenges have been identified for this type of system. These challenges are described below. In the following sections our collaborative work on these challenges will be presented. The models and control designs are based on response times, as this metric is rather easily measurable in the real system and because the response times can be mapped to the load status of the controlled system using the proposed model.

Performance models The first challenge is to design a performance model for the NEs, since good control designs are based on sufficiently accurate system models. The model should capture the dominant load dynamics of the NEs. Most service performance metrics such as response times and service rates depend on queue state dynamics, which means that queue models are suitable for these systems.

For the objective of performance control, simple models, such as single server queues, are often preferred. The model should only capture the dominating load dynamics of the system, since a well-designed control system can handle many model uncertainties [Åström and Wittenmark, 1997].

The classical M/M/1 model, where a single-server queue processes requests that arrive according to a Poisson process with exponential distributed service times, see Figure 2, has been shown to accurately capture the response time dynamics of a web server system [Cao et al., 2003]. However, experience from deployed systems and lab measurements have shown that databases may not have M/M/1 dynamics [Kihl et al., 2011]. Therefore, other models are required that more accurately captures the dynamics of database servers.

Admission control in MSS The NEs are loaded by two traffic sources, the measurable traffic coming to the MSS and the unknown (unmeasurable) traffic coming from the mobile users, as illustrated in Figure 3. The average arrival rates can be denoted as λ for the measurable traffic and λ_u for the unknown traffic. Overload in the NEs can be detected by monitoring the response time of requests sent to each node. When the average requests' response times exceed some threshold, the MSS can classify the involved NE as overloaded and thereby start actions to lower the arrival rate to that particular NE, in order to achieve an acceptable arrival rate, denoted as λ_c. Therefore, the second control challenge is to design an admission control scheme that can handle the unknown traffic at the NEs and further can handle the

time varying mean measured traffic rates experienced in the systems.

Monitoring and estimation One of the problems when designing control mechanisms in these types of systems is the lack of performance information. The designed protocols basically provide no means of control communication between the MSS and the NEs that can be used by a control system. Therefore, the third control challenge that has been identified is the design of monitoring and estimation mechanisms that could help in the design of, for example, an admission control scheme. The estimation scheme can be used as feed-forward control in the control system, and thereby improving the performance of the control system compared to when only using feedback control. In collaboration with Ericsson AB, some preliminary work on the application of extended Kalman filters for load estimation have been started for systems as in Figure 3.

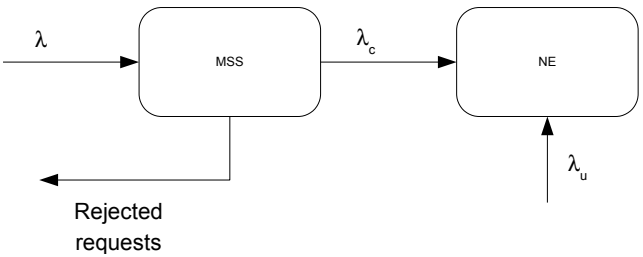

Figure 3. Load at the NEs.

3. Testbed

To validate some of the proposed solutions, we have performed a series of experiments in our server lab. We developed a MSS testbed with two traffic generators, one for the measurable traffic and one for the unknown traffic, and a MySQL 5.1.41 database server as depicted in Figure 4. The computers were connected to a local 100 Mbit/s Ethernet network.

The traffic generators were implemented in Java, using the JDBC MySQL connector, and they were executed on computers with an AMD Phenom II X6 1055T Processor at 2.8 GHz and 4 GB main memory. The operating system was Ubuntu 10.04.2 LTS. The traffic generators use 200 working threads and generate MySQL queries according to a Poisson process with average rate λ and λ_u queries per second. Both traffic generators were validated in order to guarantee that they were not a bottleneck in the experiments.

The database server has several relations with the same structure but with different number of tuples. The maximum number of allowed concurrent connections is

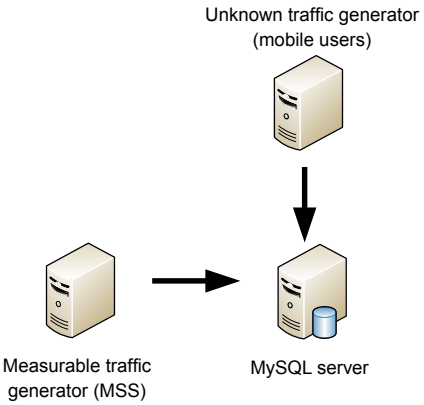

Figure 4. Testbed for the experiment.

set to 100. The structure of the relations comes from the Scalable Wisconsin Benchmark [DeWitt, 1991] with 10 million tuples. Two basic types of queries are used, `SELECT` (read) and `UPDATE` (write).

The queries look like this:

```
SELECT * FROM <relation> WHERE unique1=?;
UPDATE <relation> SET unique2=? WHERE unique1=?;
```

The question marks are replaced with uniformly distributed random numbers from zero to ten million.

4. Performance Models

In this section, we focus on the modeling aspects of database servers. The objective is to develop a performance model for the database server that captures the dynamics during high loads. The performance model can be used in resource optimization schemes, as admission control systems, in order to maximize the throughput of the database server, while keeping some latency constraints. One of the challenges for these database servers is that they have a write-heavy workload, which means that the CPU is not the bottleneck during high loads. This means that previous work on performance modeling of server systems may not be applicable since they assume CPU-intensive workload.

4.1 M/M/m model with load dependency (M/M/m-LDS)

We propose to add load-dependency to an M/M/m system. In all load-dependent server models, the service time for a request will be dependent on the number of concurrent requests in the system. This load-dependency will model effects of the operating system, memory use, etc., which may cause service degradation when

there are many concurrent jobs in a computing system [Curiel and Puigjaner, 2001]. In the experiment section, we will show that the M/M/m-LDS model accurately captures the behavior of various database workload.

The properties of the load dependent M/M/m model (*M/M/m-LDS*) are set by an exponential distributed base processing time, $x_{base} = 1/\mu$ and a dependency factor, f. When a request enters the system, it gets the base processing time x_{base} assigned to it. A single request in the system will always have a processing time of x_{base}. Each additional request inside the system increases the residual work for all requests inside the system (including itself) by a percentage equal to the dependency factor f. When a request leaves the system all other requests have their residual work decreased by f percent again. This means that if n concurrent requests enter the system at the same point, they will all have a processing time of

$$x_s(n) = x_{base} \cdot (1+f)^{n-1} \tag{1}$$

A special case is when $f = 0$. It means that there is no load dependency, and all requests will have processing time x_{base}.

The system can process a maximum of m concurrent requests at each time instance. Any additional request will have to wait in the queue. New requests arrive according to a Poisson process with average rate λ.

Therefore, the system can be modeled as a Markov chain as illustrated in Figure 5.

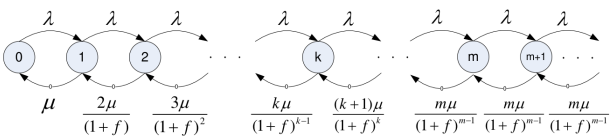

Figure 5. Illustration of M/M/m-LDS model as a Markov chain.

The average service rate of the system depends on the number of concurrent requests in the system, k, derived as follows:

$$\mu_k = \begin{cases} \dfrac{k\mu}{(1+f)^{k-1}} & \text{if } 0 < k < m \\ \dfrac{m\mu}{(1+f)^{m-1}} & \text{if } k \geq m \end{cases} \tag{2}$$

By solving the balance equations, stationary probability distribution of existence of k concurrent requests in the system is calculated as below:

$$\pi_k = \begin{cases} \frac{\left(\frac{\lambda}{\mu}\right)^k}{k!}(1+f)^{\frac{k(k-1)}{2}}\pi_0 & \text{if } 0 < k < m \\ \frac{\left(\frac{\lambda}{\mu}\right)^k}{m^{k-m}\cdot m!}(1+f)^{(m-1)(k-\frac{m}{2})}\pi_0 & \text{if } k \geq m \end{cases} \tag{3}$$

As the sum of the probabilities of all possible states equals to one, π_0 can be derived as follows:

$$\sum_{k=0}^{\infty}\pi_k = 1 \rightarrow$$

$$\pi_0 = \frac{1}{1+\sum_{k=1}^{m-1}\frac{\left(\frac{\lambda}{\mu}\right)^k}{k!}(1+f)^{\frac{k(k-1)}{2}} + \frac{\mu\left(\frac{\lambda}{\mu}\right)^m(1+f)^{\frac{m(m-1)}{2}}}{(m-1)!(\mu m-\lambda(1+f)^{m-1})}} \tag{4}$$

The stability condition in this case is:

$$\frac{\lambda}{\mu m}(1+f)^{m-1} < 1 \tag{5}$$

The average number of requests in the system, N, can be calculated as below:

$$\begin{aligned} N &= \sum_{k=1}^{\infty} k\cdot\pi_k = N_1 + N_2 \\ N_1 &= \sum_{k=0}^{m-1}\frac{\left(\frac{\lambda}{\mu}\right)^k(1+f)^{\frac{k(k-1)}{2}}}{(k-1)!}\pi_0 \\ N_2 &= \frac{\left(\frac{\lambda}{\mu}\right)^m(1+f)^{\frac{m(m-1)}{2}}(\mu m^2-\lambda(m-1)(1+f)^{m-1})\mu}{(m-1)!(m\mu-\lambda(1+f)^{m-1})^2}\pi_0 \end{aligned} \tag{6}$$

Finally by means of Little's theorem [Kleinrock, 1975], the average time each request spends in the system, T, can be derived as follows.

$$T = \frac{N}{\lambda} \tag{7}$$

4.2 M/M/m/n model with load dependency (M/M/m/n-LDS)

In case that the queue is limited to n positions, the probability for an empty system, π_0, can be determined as follows. This queuing system is named as *M/M/m/n-LDS*.

$$
\begin{aligned}
\pi_0 &= \frac{1}{I+II+III} \\
I &= 1+\sum_{k=1}^{m-1}\frac{\left(\frac{\lambda}{\mu}\right)^k (1+f)^{\frac{1}{2}k(k-1)}}{k!} \\
II &= \frac{(1+f)^{\frac{1}{2}m^2+\frac{1}{2}m+mn-n-1}\lambda^{n+m+1}}{m^n\mu^{n+m}m!(\lambda(1+f)^{m-1}-\mu m)} \\
III &= -\frac{(1+f)^{\frac{1}{2}m(m-1)}\lambda^m}{\mu^{m-1}(m-1)!(\lambda(1+f)^{m-1}-\mu m)}
\end{aligned}
\tag{8}
$$

Further, the average number of requests in the system is as follows:

$$
\begin{aligned}
N &= N_1 - N_2 \\
N_1 &= \sum_{k=0}^{m-1}\frac{k\left(\frac{\lambda}{\mu}\right)^k (1+f)^{\frac{1}{2}k(k-1)}\cdot\pi_0}{k!} \\
N_2 &= \frac{\mu(1+f)^{\frac{1}{2}m^2-\frac{1}{2}m-1}}{m^{m-1}\left(-\lambda(1+f)^{m-1}+\mu m\right)}\cdot\frac{N_{2_{n1}}+N_{2_{n2}}-N_{2_{n3}}}{N_{2_{D1}}+N_{2_{D2}}+N_{2_{D3}}} \\
N_{2_{n1}} &= -\lambda(n+m)(1+f)^{\left(\frac{1}{2}m^2+\frac{3}{2}m+mn-n-1\right)}\left(\frac{\lambda}{\mu}\right)^{n+m+1}\left(\frac{1}{m}\right)^{n+1} \\
N_{2_{n2}} &= \left(m(1+f)^m\mu(n+m+1)(1+f)^{\left(\frac{1}{2}m^2+\frac{1}{2}m+mn-n\right)}\right)\left(\frac{\lambda}{\mu}\right)^{n+m+1}\left(\frac{1}{m}\right)^{n+1} \\
N_{2_{n3}} &= \left(-\lambda(1+f)^m\mu(m-1)+(1+f)\mu m^2\right)\left(\frac{\lambda}{\mu}\right)^m(1+f)^{\frac{1}{2}m(m-1)} \\
N_{2_{D1}} &= \left(\frac{1}{m}\right)^m(1+f)^{\frac{1}{2}m(m-1)}m!\left(-\lambda(1+f)^{m-1}+\mu m\right)\left(\sum_{k=1}^{m-1}\frac{\left(\frac{\lambda}{\mu}\right)^k(1+f)^{\frac{1}{2}k(k-1)}}{k!}\right) \\
N_{2_{D2}} &= \left(\frac{-\lambda(1+f)^{m^2+mn-n-1}}{(\mu m)^{n+m}}\right)+\left(\frac{1}{m}\right)^m(1+f)^{\frac{1}{2}m(m-1)}m!\left(-\lambda(1+f)^{m-1}+\mu m\right) \\
N_{2_{D3}} &= \mu m\left(\frac{\lambda(1+f)^{m-1}}{\mu m}\right)^m
\end{aligned}
\tag{9}
$$

Finally, the average response time for a request can be derived using Little's theorem.

4.3 Parameter tuning

In a telecom system with latency constraints, the dominant dynamic of the system is often characterized by the average response time, T, when varying the average arrival rate, λ. Tuning of the parameters of the LDS model in a way that it fits the measured data from the actual server system is a necessary step in modeling of such systems. Assuming that λ and T are measureable, there are three main parameters for the M/M/m-LDS model, m, f and μ to tune in order to fit the model on the measured data. Further, for the M/M/m/n-LDS there is an extra parameter, n, to tune.

Therefore, in Figures 6-10, the effects of changing model parameters on dynamics of average response time versus mean arrival rate of queries are illustrated. In the rest of the paper, this graph will be called the λ/T graph. In each figure, it is assumed that two (three) of the parameters are fixed and the one that is mentioned is the variable. As the equations for calculating the mean response times are rather complex and the parameters are interdependent, more than one set of parameters can be fit on the measured data. Thus using these figures, a heuristic rule for tuning the parameters of the LDS model can be achieved.

In the cases where the M/M/m-LDS model is used, the first parameter to be tuned is the number of servers, m. As it can be seen in Figure 6, by increasing the maximum number of concurrent requests that can be processed in the system, the linear part of the λ/T graph will be shorter and the exponential rising rate of the graph is increased. In this case it is assumed that $(f,\mu) = (0.6, 22)$.

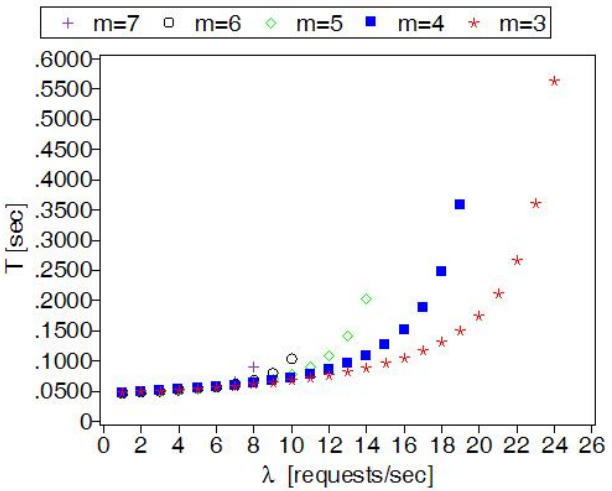

Figure 6. Variations of the λ/T graph for a special scenario with m as variable when $(f,\mu) = (0.7, 22)$.

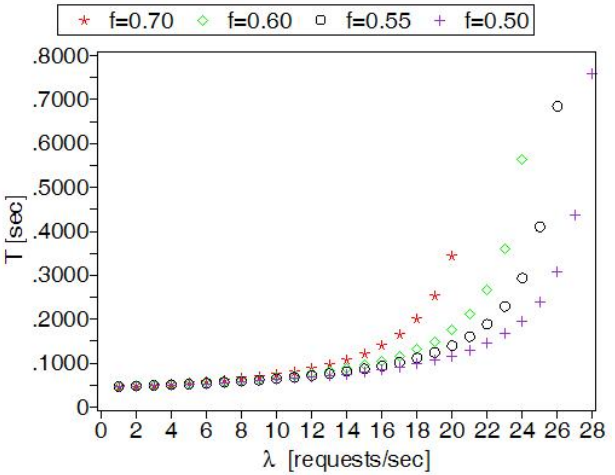

Figure 7. Variations of λ/T graph for a special scenario with f as variable when $(m,\mu) = (3,22)$.

The second parameter to be tuned is the dependency factor, f. As shown in Figure 7, by decreasing the dependency factor, the linear part of the λ/T graph is increased, however, the change is slower than in the case where m is decreased. On the other hand the exponential rising rate of the graph is increased in comparison with the case where m is decreased. Here, it is assumed that $(m,\mu) = (3,22)$.

The effects of changing μ on the λ/T graph while fixing the two other parameters is illustrated in Figure 8. As shown in the figure, by increasing μ in equal steps, the λ/T graph will be shifted to the right in equal steps. In this case, where $(m,f) = (3,0.6)$, the rate of rising of the graph is decreased.

In cases where the M/M/m/n-LDS model is used, there will be a saturation of the response times when the load is high enough to overload the queue. Here, it is assumed that the default values are $(m,n,f,\mu) = (4,15,0.6,22)$. Figure 9 and Figure 10 show the effects when varying m and f respectively. In each case, the values of the other three parameters are constant. The general effect of changing the parameters is similar as for the case with the infinite queue, with the difference that the response times saturate when the load is high.

4.4 Experiments

In order to validate the model, we have performed a series of experiments in our testbed, as described in Section 3. In this case, the arrival rate of the unknown traffic was set to zero. The dynamics of the database server highly depends on the mix of requests, since SELECT and UPDATE queries require different amount of server capacity. Therefore, experiments with varying workload mix have been performed.

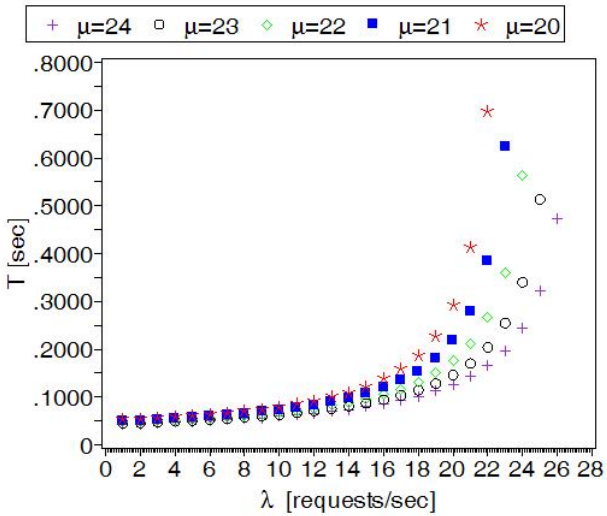

Figure 8. Variations of λ/T graph for a special scenario with μ as variable when $(m, f) = (3, 0.6)$.

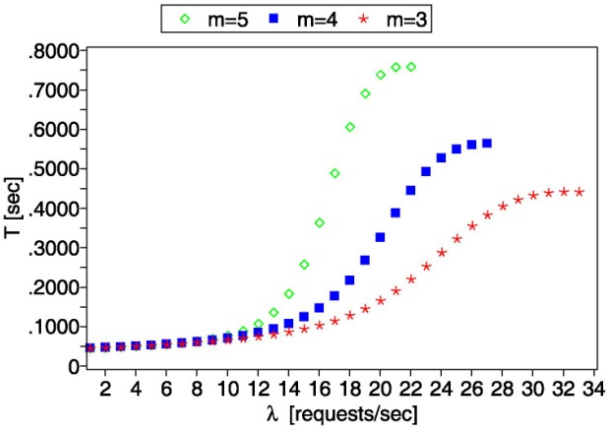

Figure 9. Variations of λ/T graph for a special scenario with m as variable when $(n, f, \mu) = (15, 0.6, 22)$.

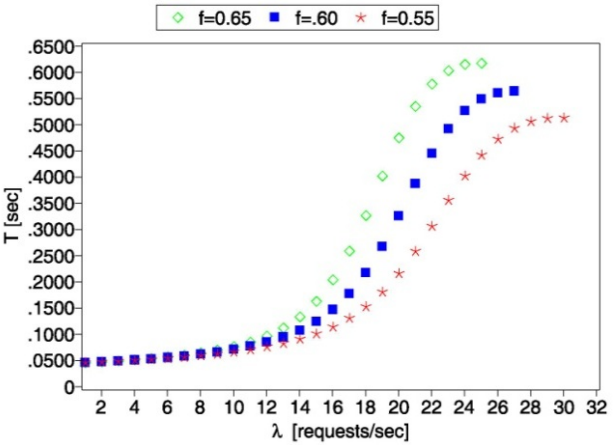

Figure 10. Variations of λ/T graph for a special scenario with f as variable when $(m,n,\mu) = (4,15,22)$.

Figure 11, Figure 12, and Figure 13 show the results from experiments where the arrival rate is varied from low load to high load. The graphs show the average response times of queries as a function of the arrival rate. We have fitted M/M/m/n-LDS models for the data using the tuning steps described in the previous section. In both scenarios, the CPU utilization was very low, also for high loads. The maximum CPU load was about 5%.

In order to model the network delays, we have added a bias of 0.023 seconds in the average response times of the proposed models.

In Figure 11, the workload is based on 100% UPDATE queries. The fitted model in this case has the following parameters $(m,n,f,\mu) = (3,81,0.75,37.1)$. Figure 12 depicts the same experiment setup when using a mix of 25% SELECT queries and 75% UPDATE queries. The fitted M/M/m/n-LDS model in this case has the following parameters $(m,n,f,\mu) = (6,73,0.44,35.2)$. In Figure 13 only SELECT queries are used. In this case the model parameters are $(m,n,f,\mu) = (6,240,1.39,38)$.

The results verify that the proposed model can represent the average dynamics of a database server with various workloads very well.

5. Admission Control

As part of the collaboration with Ericsson AB, we have designed an admission control mechanism for the measurable traffic to the NEs, as illustrated previously in Figure 3. As a direct effect of this work, a modified version of the control mechanism has been implemented in the Ericsson product. In this section, the controller design and its validation are described.

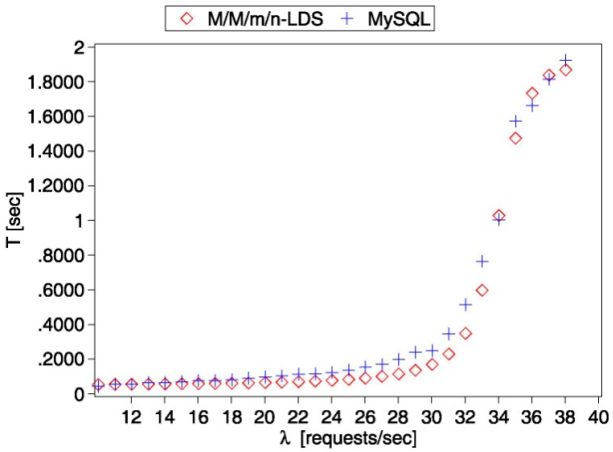

Figure 11. Performance of the M/M/m/n-LDS queuing model in modeling steady state dynamics of a MySQL database server using UPDATE queries.

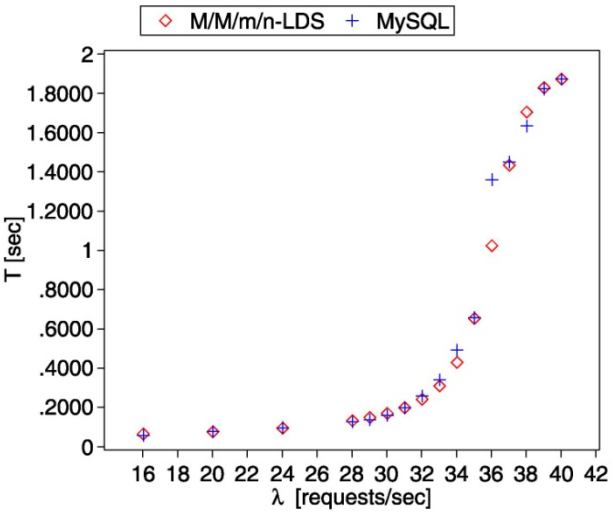

Figure 12. Performance of the M/M/m/n-LDS queuing model in modeling steady state dynamics of a MySQL database server using mixed queries.

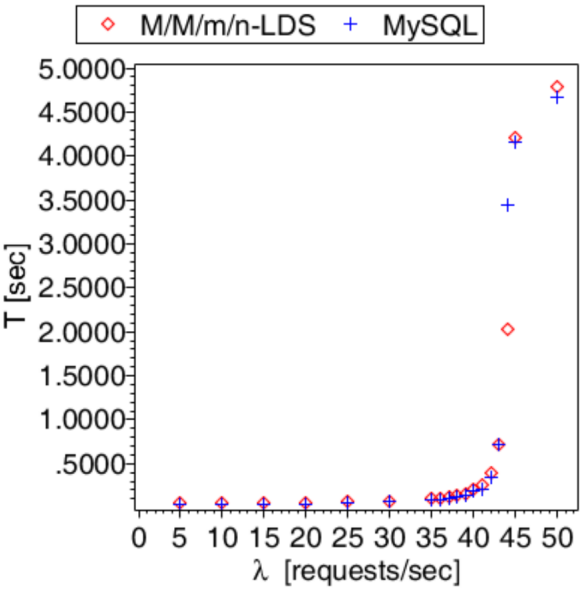

Figure 13. Performance of the M/M/m/n-LDS queuing model in modeling steady state dynamics of a MySQL database server using SELECT queries.

5.1 Control structure

The MSS includes a control system, as illustrated in Figure 14, which should ensure that the load on a specific NE is kept at an acceptable level. The control objective is to keep the mean response times of the NE queries below a desired value while maximizing the throughput. The control actions must be based on a limited amount of control information, due to the standardized protocols and the layered software architecture. The control system includes a controller and a gate.

The controller uses a response time reference value, T_{ref}, and measurements to determine an acceptable workload to the database server. The acceptable workload is defined by the normalized rate of admitted queries, λ_A, which corresponds to the ratio of the average arrival rate of the admitted requests over the higher bound of the average arrival rate of the requests. It is desired that the control system performs robustly in presence of fluctuations in the average arrival rate of the queries sent to the database. Therefore, the controller design is crucial for guaranteeing the control objectives.

The gate ensures the ratio λ_A of arriving queries is admitted to the database. In the experiments, the gate rejects requests that cannot be admitted. However, in the real product, this is not feasible. Instead, the real product has a traffic shaping mechanism that adds delays to the responses to the customer administration system.

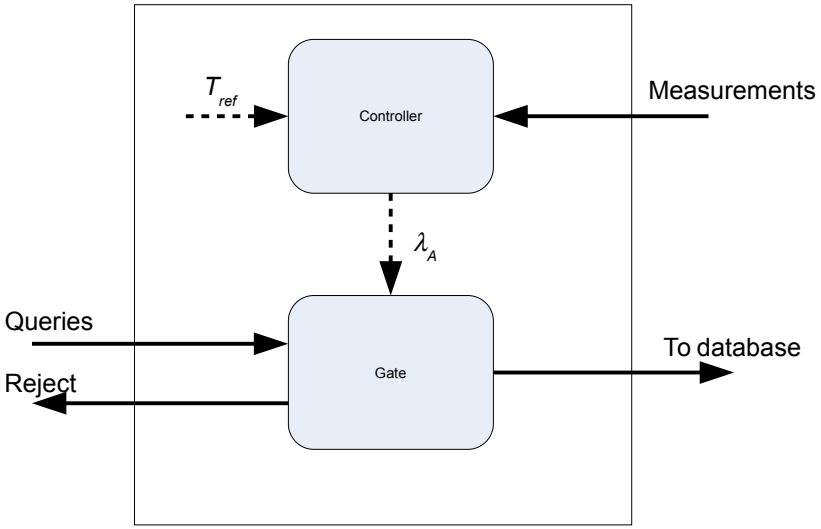

Figure 14. Control system.

Since the communication with the customer administration system is synchronous, adding delays to the responses will lower the arrival rate of requests.

In this paper, we focus on the controller performance. Therefore, the implementation of the gate is not the main focus as long as it can be assumed that the gate actuates the control signal accurately.

5.2 Controller design

We have designed a controller that can guarantee the control objectives for the system. The controller, called the Load-Adaptive Controller (LAC), only uses measurements of the query response times. A classical PID controller [Åström and Wittenmark, 1997] includes one Proportional part (P), one Integral part (I), and one Derivative part (D) that determines the control signal based on the deviation of the input signal from the reference value. For stochastic systems, the derivative part will amplify the effect of high frequency noise in the response time error and thus deteriorate the overall performance of the system.

Therefore, the LAC is based on a modified PI controller with anti-windup. The LAC adapts its proportional gain with the variations in the mean arrival rates of queries sent to the database. The structure of the modified PI controller is illustrated in Figure 15.

The total load of the NE is determined by the aggregated arrival rates of the

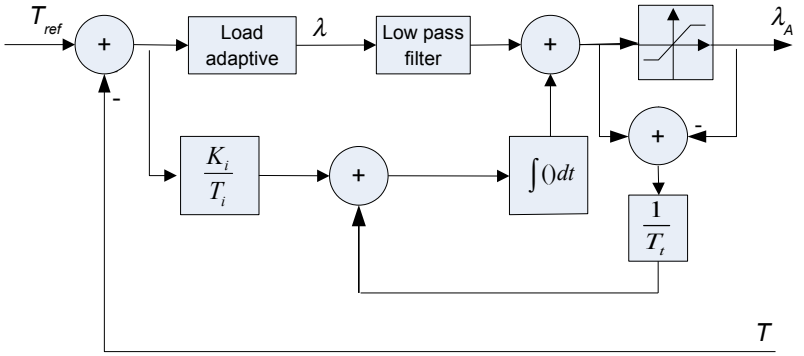

Figure 15. Load-adaptive controller (LAC).

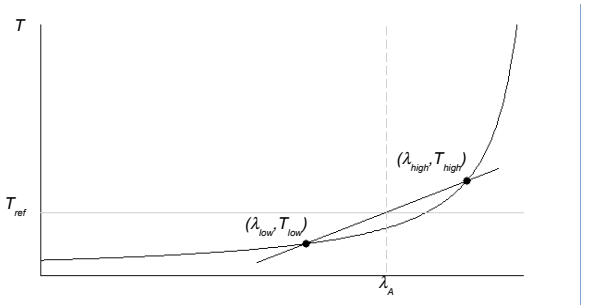

Figure 16. An illustration of the LAC calculations.

measurable and the unknown traffic streams. However, assuming that the unknown traffic is stationary during a limited time period and that the database server behaves as a conservative queuing system [Kleinrock, 1975], a specific admitted ratio of the traffic will correspond to a specific mean response time, as illustrated in Figure 16.

The controller continuously keeps track of two points in this graph, one low point, (λ_{low}, T_{low}), which is situated below the reference response time, T_{ref}, and one high point, $(\lambda_{high}, T_{high})$, which is situated above T_{ref}. As the control system operates only based on measured response times of NE queries, λ_{low} guarantees that those measurements exist for all sampling intervals. The upper limit for mean arrival rates of the queries processed by the NE while not overloading the database is represented by λ_{high}. The starting values for λ_{low} and λ_{high} are set to 5% and 100% respectively.

The admittance rate of the incoming queries is iteratively updated so that its corresponding response time meets the desired value. Every sampling time, the controller calculates the average response time, T, over the last period. If the av-

erage response time during sampling period k, T_k, is too high, $(T_k > T_{ref})$, the high point is updated as $(\lambda_{high}, T_{high}) = (\lambda_k, T_k)$ where λ_k is the normalized admitted arrival rate during interval k. If the average response time during interval k is too low, $(T_k < T_{ref})$, the low point is updated as $(\lambda_{low}, T_{low}) = (\lambda_k, T_k)$. It is now assumed that the optimal normalized arrival rate, λ_o, which gives a response time of exactly T_{ref} is in the interval $[\lambda_{low}, \lambda_{high}]$. Therefore, the next normalized admitted arrival rate, λ_{k+1}, can be interpolated from these points using classic geometry:

$$\lambda_{k+1} = \lambda_k + \frac{\lambda_{high} - \lambda_{low}}{T_{high} - T_{low}}(T_{ref} - T_k) \tag{10}$$

Therefore, the quotient $(\lambda_{high} - \lambda_{low})/(T_{high} - T_{low})$ is used as proportional gain in the P-part of the controller. The algorithm will converge to the desired response time value assuming that the arrival process is stationary or slowly changing. It is obvious that the control gate cannot admit more queries than the incoming ones. This upper limit will be noted in the calculations and treated as a saturation limit of the control signal.

The integral I-part of the controller is used when the P-part is not enough for keeping the steady state error to zero. The integral part uses a controller parameter, K_i, which in conventional PI controllers are equal to the proportional gain. However, in this case, as the proportional gain changes drastically due to the load-adaptive algorithm, using the conventional PI structure will lead to a reduced phase margin which will drive the system to unstable region. Therefore, K_i is chosen as a static gain and its suitable value is determined in tuning phase of the controller.

Further, the parameter T_i is the integration time constant and T_t is the integrator‘s reset time constant in the anti-windup mechanism. Anti-windup is added to avoid building up of the integration part when the control gate is saturated or completely open. It is desired to choose small values for T_t so that the integrator resets quickly. Generally, T_t is chosen to be less than T_i.

A low pass filter is added after the proportional gain to smoothen the response time error signal as it is very noisy. The bandwidth of this filter should be suitably chosen so that its effect on the in-band characteristics of the response time errors is minor while attenuating high frequency components of that signal.

5.3 Experiments

To investigate the controller performance, a Java implementation of the controller was deployed as a web application to a Glassfish application server, placed on the server acting as traffic generator in Figure 4. The web application also included the traffic generator that generated requests for the web application. For each request, the admission control decides whether to allow the request to be sent to the database or rejected. The traffic generator for unknown traffic did not have an admission control, and was set to a specific average arrival rate that could be altered during run time. All requests sent to the database server were SELECT queries (according

to the query structure described earlier). The λ/T graph for this particular scenario setting is shown in Figure 17. The saturation of the system is not shown in the graph

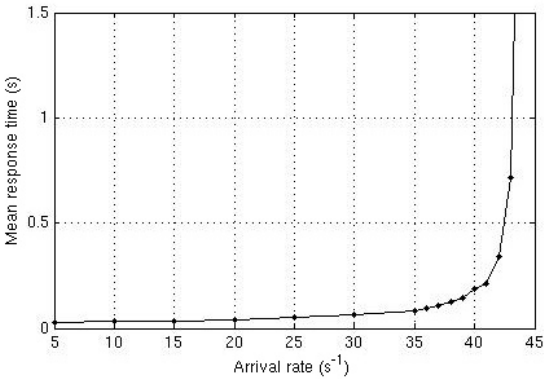

Figure 17. λ/T graph for the admission control experiments.

for clarity reasons, since the operation region is around the "knee".

To test the performance of the controller, a scenario was chosen where the load changed from slight overload to high overload. The reference response time, T_{ref}, was set to 0.2 seconds. According to the λ/T graph in Figure 17., this corresponds to a total arrival rate of approximately 40 queries per second.

In this paper, two experiments are shown, one with a step in the unknown traffic and one with a step in the measurable traffic. The controller parameters were set to $T_i = 4$, $K_i = 0.5$, $T_r = 1$, and the sampling time $h = 0.5$ seconds. T_i was determined as a multiple of the sampling time, chosen so that the controller was able to maximize the throughput while keeping the mean response times below T_{ref}. K_i was set equal to the sampling time. To give the controller time to settle this state was kept for 100 seconds after which a step in the traffic was performed. The resulting graphs are shown in Figure 18 and Figure 19. The graphs show the average dynamics from 100 runs.

In the first experiment, shown in Figure 18, the starting arrival rate was set to 23 requests per second for the measured traffic and 22 requests per second for the unknown traffic. The step increased the arrival rate of the unknown traffic by 10 (to 32) requests per second, resulting in a more severe overload situation.

The second experiment, shown in Figure 19, was similar to the first experiment. However, the arrival rate step was in the measurable traffic instead. To obtain a similar control signal response as in the first experiment, the step in the controllable traffic had to be larger. Therefore, the observable arrival rate was increased from 23 requests per second to 51 requests per second.

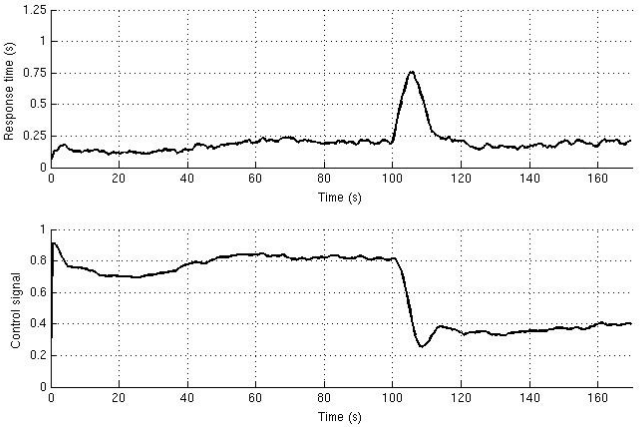

Figure 18. Performance of the LAC with step in unknown traffic.

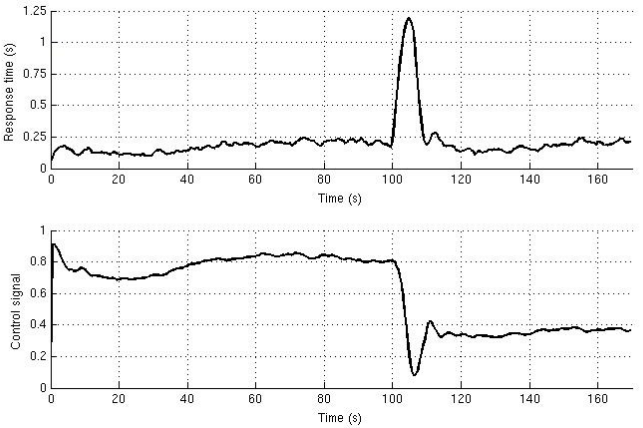

Figure 19. Performance of the LAC with step in observable traffic.

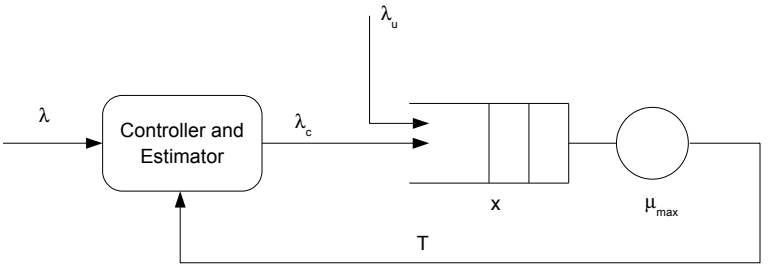

Figure 20. Schematic diagram of an abstraction of the MSS in Figure 1 with a controller and estimator.

Both experiments show a well-behaved controller, with a reasonable settling time and smooth dynamics after the step.

6. Monitoring and Estimation

The system in Figure 1 is complicated with many different queues, caches and databases. Attempting to capture all details gives models that are too complex for on-line control. Extensive experience in the field of control has clearly demonstrated that simple models that capture essential behavior can be very beneficial [Åström and Murray, 2008]. One aspect of the collaboration with Ericsson has been to explore if benefits can also be obtained for monitoring and control of the MSS. A crucial issue is what complexity of the models is required for estimation and control of the MSS.

Response time and arrival rates are variables of prime concern. The variables have strong variations, which can be reduced by averaging. A more effective way is to construct estimators that exploit the dynamic behavior of the system. Exploration of such estimators has been one of the goals of the project.

A key feature of the system shown in Figure 1 is that there are two traffic streams. The measured traffic, generated by the customer administration system has a known arrival rate λ_c, can be controlled. The unknown stream, which is created by the mobile phone users, has an arrival rate λ_u that cannot be controlled. Monitoring and control of the system can be improved if good estimates of the average service time are available.

An abstraction of the system in Figure 1 is shown in Figure 20, where an estimator and the controller have been included. In this section, we will focus on the estimator, which only has access to measurements of the measured arrival stream λ and the response time T. All actions by the NEs and the MSS have been represented by one queue that represents the aggregated behaviors.

The queue length is represented by the variable x, which captures the aggregated behavior of many different queues in the real system. The variable x can be inter-

preted as a virtual queue length. The queue length cannot be measured. The actual response time T and the actual arrival times can, however, be measured. Variations in x reflect changes in the system's load.

6.1 Flow Model

To model the system, we will make an additional abstraction by assuming that the variables x and T are continuous and that they vary continuously in time. The behavior of the system can then be captured by the simple flow model:

$$\frac{dx}{dt} = \lambda - \mu_{\max} f(x) \tag{11}$$

where x is the virtual queue length, λ_c is the known arrival rate, λ_u is the unknown arrival rate, μ_{max} is the maximum service rate and f is a monotone function with the range [0, 1]. The response time is given by:

$$T = t_0(1+x) = t_0(1+f^{-}(\rho)) \tag{12}$$

where $t_0 = 1/\mu_{max}$ is the average time to serve one job when the queue is empty and ρ is the normalized service rate or the utility $\rho = \lambda/\mu_{max}$.

The response time goes to infinity as λ approaches μ_{max} if the range of the function f is [0, 1]. The function f gives significant freedom in adjusting the behavior to real queue behavior.

The model (11), (12) has been used extensively to model queuing systems [Agnew, 1976]. The simple M/M/1 queue can be represented by (12) with $f = x/(x+1)$ [Tipper and Sundareshan, 1990].

Even if the model (11), (12) is simple it captures some important features of real queuing systems, for example the fact that response time increases with queue length. The model also captures the behavior that the rate of change of the response time increases with increasing arrival rate. The behavior of the system can be shaped by the function f.

In the project, we have investigated simulated models with servers and we have demonstrated that it is possible to find functions f which matches the steady state behavior of simulated systems. An illustration is given in Figure 21.

6.2 Estimation Algorithm

There are significant variations in the arrival and response times due to their discrete nature. To monitor and control the system it is necessary to smooth these variations. For example, the average arrival rate of the controlled stream can be estimated the simple exponential smoother

$$\begin{aligned} \hat{t}_i^{+} &= \hat{t}_i + k_3(h_a - \hat{t}_i) \\ \hat{\lambda}_c^{+} &= 1/\hat{t}_i^{+} \end{aligned} \tag{13}$$

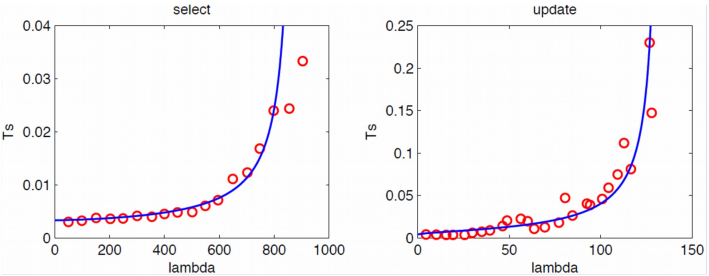

Figure 21. Service times for the operations SELECT (left) and UPDATE on an SQL server and predictions based on the model (12) with $f(x) = (1/(1+x))^n$, $n = 1.5$ and $\mu_{max} = 880$ for SELECT and $n = 0.15$ and $\mu_{max} = 132$ for UPDATE.

where t_i is the arrival time and h_a is the time since the last arrival update.

One advantage with the model (11), (12) is that it is possible to use Kalman filtering [Åström and Murray, 2008] to combine the model, which captures the gross behavior of the queuing system, with measured data.

If continuous data was available, an extended Kalman filter for the service time is given by:

$$\begin{aligned} \frac{d\hat{x}}{dt} &= \lambda_c + \lambda_u - \mu_{\max} f(\hat{x}) + k_1(T - t_0(1+\hat{x})) \\ \frac{d\lambda_u}{dt} &= k_2(T - t_0(1+\hat{x})) \end{aligned} \tag{14}$$

This filter will capture the behavior that response time increases with increasing queue length and arrival rate. The detailed behavior can be shaped by the function f.

It must be considered that the real measurements are events that represent arrival of a request or a completed response. To deal with this, we have developed an event-based Kalman filter. At arrivals, the queue length is updated according to the flow model:

$$\hat{x}^+ = \hat{x} + h_a(\hat{\lambda}_c + \hat{\lambda}_u - \mu_{\max} f(\hat{x})) \tag{15}$$

This difference equation is simply a forward Euler approximation of (11). Equation (15) is simply a prediction of x based on the model (11). Information about x is obtained when a service is completed. The queue length and the unknown arrival rate are then updated as:

$$\begin{aligned} \hat{x}^+ &= \hat{x} + h_d(\lambda_c + \lambda_u - \mu_{\max} f(\hat{x}) + k_1(T - \hat{T})) \\ \hat{\lambda}_u^+ &= \hat{\lambda}_u + h_d k_2(T - \hat{T}) \end{aligned} \tag{16}$$

where h_d is the time since the last departure update. The arrival rate can be estimated because it results from the model (11) and (12) that the arrival rate is observable from a measurement of service time [Åström and Murray, 2008].

6.3 Experiment

The Kalman filter estimator was evaluated using a discrete-event simulation program written in Java, The program simulates a single server queue with exponentially distributed service times with mean $\mu_{max} = 100$ requests per second. The queue has two arrival processes, representing the measurable and unknown traffic. The Kalman filter has been evaluated for a number of scenarios validating its performance. However, in this paper we show the results of one specific scenario.

In this scenario, the unknown arrival process was a stationary Poisson process with mean 42.5 requests per second. The measurable arrival process was basically a Poisson process with changing average rate. The arrival rate, λ, was the sum of one constant part and one part represented by a sine function as given by:

$$\lambda(t) = C + a \cdot \sin(kt) \tag{17}$$

The parameters were chosen so that the system can handle the workload over long time but with periodic overloads, hence:

$$\mu_{\max} - a < C < \mu_{\max} \tag{18}$$

Therefore, the numerical values used in the simulations are $C = 42.5$ and $a = 20$ requests per second.

The differential equations describing the behavior of the estimates between events were approximated using first order forward Euler discretization.

Figure 22 shows the response times and the arrival rate, both real values and estimates for a time period of 20 seconds during the simulation. The estimate error is shown in Figure 23. It can be seen how the Kalman filter manages to follow the real system during the quick rises in response time around time 424 and 427. Here the mean square error is $\sigma = 7.4 \cdot 10^{-4}$ for the period $415 < t < 420$ and $\sigma = 1.1 \cdot 10^{-2}$ for the period $425 < t < 430$. The mean square error for the entire experiment is $\sigma = 1.9 \cdot 10^{-2}$.

7. Conclusion

Accurate control designs using control theory are essential for resource management in computer systems. In this paper we have presented work performed in collaboration with Ericsson AB, investigating how control theory can improve the performance of a commercial mobile service support system. Together with Ericsson AB, we have identified three major control challenges, and investigated solutions. The first challenge is to find accurate performance models for the system, with the objective to capture the system dynamics. The second challenge is to develop an admission control scheme that can handle unknown traffic and load surges. The final challenge is to develop estimation methods for accurate prediction of response times and arrival rates in systems with unknown traffic.

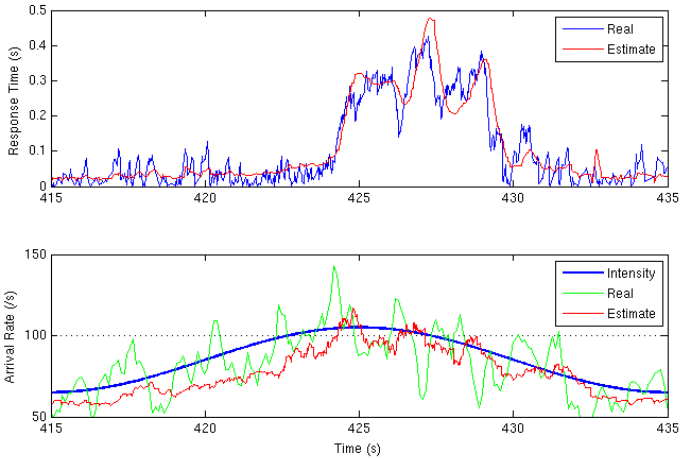

Figure 22. Kalman filter estimates of response times and estimation of arrival rate.

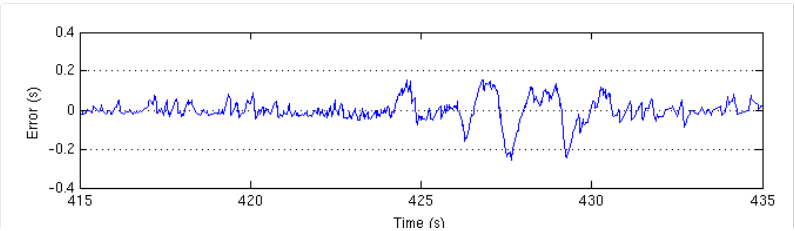

Figure 23. Proposed Kalman filter‘s response time prediction error.

In this paper, the challenges have been treated rather independent of each other. However, the future goal is to be able to use all solutions together, in order to improve the system performance and speed up the development process. The performance model could be tuned using real data and then used for validating control designs, which is much easier than implementing the designs in testbeds or the real system. Also, in the future, the estimation algorithms should be incorporated in the control system, improving the control decisions.

Acknowledgment

The authors at Lund University are members of the Lund Center for Control of Complex Engineering Systems (LCCC). Maria Kihl and Anders Robertsson are members of the Excellence Center at Linköping-Lund in Information Technology

(eLLIIT). The work is partly funded by the Swedish Research Council, grant VR 2010-5864.

References

Agnew, C. E. (1976). "Dynamic modeling and control of congestion-prone systems". *Operations research* **24**:3, pp. 400–419.

Åström, K. J. and B. Wittenmark (1997). *Computer-Controlled Systems*. Prentice Hall. ISBN: 9780133148992.

Åström, K. and R. Murray (2008). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press. ISBN: 9780691135762.

Bianchini, R. and R. Rajamony (2004). "Power and energy management for server systems". *Computer* 11, pp. 68–74.

Brawn, B. S. and F. G. Gustavson (1968). "Program behavior in a paging environment". In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part II*. ACM, pp. 1019–1032.

Cao, J., M. Andersson, C. Nyberg, and M. Kihl (2003). "Web server performance modeling using an M/G/1/K*PS queue". In: *10th International Conference on Telecommunications*.

Chen, X., H. Chen, and P. Mohapatra (2003). "Aces: an efficient admission control scheme for QoS-aware web servers". *Computer Communications* **26**:14, pp. 1581–1593.

Claussen, H., L. T. Ho, and F. Pivit (2009). "Leveraging advances in mobile broadband technology to improve environmental sustainability". *Telecommunications Journal of Australia* **59**:1, pp. 4–1.

Crocus (1975). *Systemes d'Exploitation des Ordinateurs*. Dunod, Paris.

Curiel, M. and R. Puigjaner (2001). "Using load dependent servers to reduce the complexity of large client-server simulation models". In: *Performance Engineering*. Springer, pp. 131–147.

DeWitt, D. J. (1991). "The Wisconsin benchmark: past, present, and future". In: *The Benchmark Handbook*, pp. 119–165.

Diao, Y., C. W. Wu, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, L. Chu, et al. (2005). "Comparative studies of load balancing with control and optimization techniques". In: *American Control Conference*. IEEE, pp. 1484–1490.

Dilley, J., R. Friedrich, T. Jin, and J. Rolia (1998). "Web server performance measurement and modeling techniques". *Performance evaluation* **33**:1, pp. 5–26.

Elnozahy, E. M., M. Kistler, and R. Rajamony (2003). "Energy-efficient server clusters". In: *Power-Aware Computer Systems*. Springer, pp. 179–197.

Fu, Y., H. Wang, C. Lu, and R. S. Chandra (2006). "Distributed utilization control for real-time clusters with load balancing". In: *27th IEEE International Real-Time Systems Symposium*. IEEE, pp. 137–146.

Horvath, T., T. Abdelzaher, K. Skadron, and X. Liu (2007). "Dynamic voltage scaling in multitier web servers with end-to-end delay control". *IEEE Transactions on Computers* **56**:4, pp. 444–458.

Kihl, M., A. Robertsson, and B. Wittenmark (2003). "Performance modelling and control of server systems using non-linear control theory". *Teletraffic Science and Engineering* **5**, pp. 1151–1160.

Kihl, M., A. Robertsson, M. Andersson, and B. Wittenmark (2008). "Control-theoretic analysis of admission control mechanisms for web server systems". *World Wide Web* **11**:1, pp. 93–116.

Kihl, M., G. Cedersjö, A. Robertsson, and B. Aspernäs (2011). "Performance measurements and modeling of database servers". In: *6th International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*.

Kihl, M., P. Amani, A. Robertsson, G. Radu, M. Dellkrantz, and B. Aspernäs (2012). "Performance modeling of database servers in a telecommunication service management system". In: *7th International Conference on Digital Telecommunications*, pp. 124–129.

Kjær, M. A., M. Kihl, and A. Robertsson (2009). "Resource allocation and disturbance rejection in web servers using SLAs and virtualized servers". *Network and Service Management, IEEE Transactions on* **6**:4, pp. 226–239.

Kleinrock, L. (1975). *Theory, Volume 1, Queueing Systems*. Wiley-Interscience. ISBN: 0471491101.

Leung, K. K. (2002). "Load-dependent service queues with application to congestion control in broadband networks". *Performance Evaluation* **50**:1, pp. 27–40.

Liu, X., J. Heo, L. Sha, and X. Zhu (2006). "Adaptive control of multi-tiered web applications using queueing predictor". In: *10th IEEE/IFIP Network Operations and Management Symposium*, pp. 106–114.

Mathur, V. and V. Apte (2004). "A computational complexity-aware model for performance analysis of software servers". In: *12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. IEEE, pp. 537–544.

Mei, R. D. van der, R. Hariharan, and P. Reeser (2001). "Web server performance modeling". *Telecommunication Systems* **16**:3-4, pp. 361–378.

Menascé, D. and V. Almeida (2002). *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall. ISBN: 9780130659033.

Perros, H. G., Y. Dallery, and G. Pujolle (1992). "Analysis of a queueing network model with class dependent window flow control". In: *11th Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, pp. 968–977.

Rak, M. and A. Sgueglia (2010). "Instantaneous load dependent servers (iLDS) model for web services". In: *International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, pp. 1075–1080.

Tipper, D. and M. K. Sundareshan (1990). "Numerical methods for modeling computer networks under nonstationary conditions". *IEEE Journal on Selected Areas in Communications* **8**:9, pp. 1682–1695.

Wang, Z., X. Liu, A. Zhang, C. Stewart, X. Zhu, T. Kelly, S. Singhal, et al. (2007). "Autoparam: automated control of application-level performance in virtualized server environments". In: *2nd International Workshop on Feedback Control Implementation in Computing Systems and Networks*. IEEE.

Xu, W., X. Zhu, S. Singhal, and Z. Wang (2006). "Predictive control for dynamic resource allocation in enterprise data centers". In: *10th IEEE/IFIP Network Operations and Management Symposium*. IEEE, pp. 115–126.

Paper III

Event-Based Response Time Estimation

Manfred Dellkrantz Maria Kihl Anders Robertsson
Karl Johan Åström

Abstract

Response time is a measure of quality of service in computer systems. Estimation techniques, suitable for support systems for mobile phone systems, are explored. These systems are complex queueing systems with large databases. The traffic generated by users and system administrators changes rapidly, some loads can be measured other cannot. Attempts to capture all details give models that are not suitable for on-line control. Estimators based on continuous flow models with event based measurements are designed using extended Kalman filtering. The estimators are compared with simple-data based estimators.

Originally published in 7th International Workshop on Feedback Computing, San Jose, California, USA, 2012.

1. Introduction

Resource management of computer systems, which has gained increased attention during recent years, was explored already in the late 60's [Brawn and Gustavson, 1968; Crocus, 1975]. It is an essential mechanism to handle load disturbances such as traffic surges and changes in user behavior. Poorly managed resources can severely degrade the performance of a system with potentially large economical consequences.

This paper is motivated by mobile service activation systems, i.e., the systems which the network operators utilize for all processing regarding new subscribers and services in the network. Each new subscriber or service requires processing and data storage in several network nodes. The systems are in general multi-tier systems, implemented as distributed server clusters, where web and application servers process the incoming requests and database servers are used for data storage. The resource management of these systems, based of measurements and feedback of the actual utilization, is crucial for optimization of operation costs and the guarantee of service level agreements during load surges, for example during market campaigns or various events.

Any server system with software that processes requests can basically be modeled as a network of queues which store requests in waiting of service in the processors. Therefore, queuing models can be used to describe the dynamic behavior of server systems [Cao et al., 2003; Dilley et al., 1998; Menascé and Almeida, 2002; Mei et al., 2001]. Further, tools from control theory has emerged for both analysis and design of control of these systems [Hellerstein et al., 2005].

Previous work on resource management for server systems has mainly been focused on the web and application servers. Large software systems have high energy consumption, and therefore, dynamic resource optimization of these systems may considerably lower the operating costs for the network operators [Bianchini and Rajamony, 2004; Claussen et al., 2009; Horvath et al., 2007; Elnozahy et al., 2003]. These types of servers has mainly CPU-intensive workload, which can rather easily be modeled as single server queuing systems [Cao et al., 2003].

Resource management solutions for server systems are usually based on dynamic control schemes, which monitor the systems, and provide actions when needed. Several types of resource management mechanisms have been proposed and evaluated. In larger server systems, load balancing is performed to distribute resources uniformly over computers, CPUs, memory, etc. to avoid that some units are overloaded while others are idle [Fu et al., 2006; Diao et al., 2005]. During overload periods, when more resources are requested than are available, admission control mechanisms reduce the amount of work by blocking some of the requests [Chen et al., 2003; Liu et al., 2006; Kihl et al., 2008]. Prediction based control have been shown to improve the performance compared to control systems only including feedback [Henriksson et al., 2004; Kjaer et al., 2009; Gilly et al., 2009].

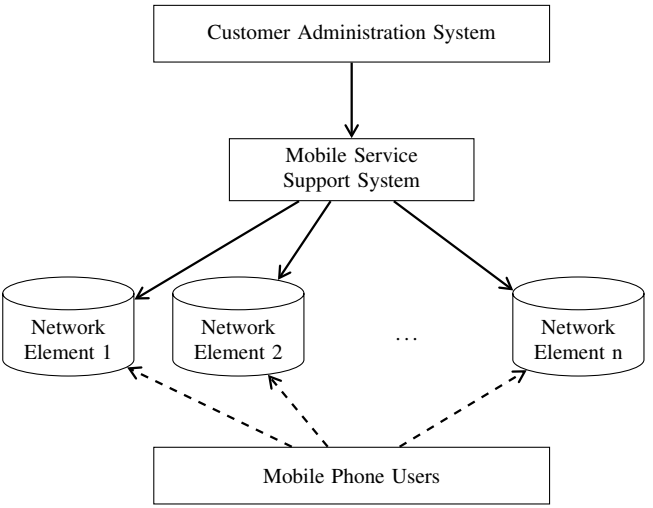

Figure 1. Schematic diagram of a support system for mobile service providers.

2. Mobile service support system

A Mobile Service Support system (MSS) handles the setup of new subscribers and services into a mobile network (illustration in Figure 1). It presents to the operator and its business support systems a unified middleware where complex functions, such as setting up a new subscriber or modifying services for an existing subscriber, can be easily invoked.

One request to the MSA from an upstream system normally results in a number of requests downstream out on the mobile network to several different network elements (NEs). A network element is usually a database storing subscriber and service data, for example, the Home Location Register (HLR). A user id which needs to be fetched from one database needs to be supplied in a query to another database to get the system consistent.

In parallel to the changes and setups that the MSA performs, the network is also used by the end users. Services being set up by the MSA are queried by base stations and other systems requiring that information. In respect to the MSA, this traffic can be considered as an unknown background traffic, in contrast to the known traffic flowing through the MSA. These two loads may interfere with each other, creating a race for resources and may put a too high load on an NE.

One NE that becomes overloaded and unresponsive may result in the entire transaction requiring rollback to avoid inconsistencies in the network. Such a rollback may require manual work which is of course costly for the operator. To protect against such situations, traffic monitoring and control is crucial.

3. Modeling

The system in Figure 1 is complicated with many different queues, caches and databases. Attempting to capture all details give models that are too complex for on-line control. Therefore we will develop simpler models that capture the gross input-output behavior. The models will be evaluated based on the quality of the estimates of the response times.

The input-output behavior of the system can be captured by the response times for each individual request. Since such a model is by nature event based we will make a further simplification by attempting to capture the gross behavior by a continuous flow model. We will recover the event-based behavior in the design of the estimators.

A simple flow model of a queue is given by [Agnew, 1976]

$$\frac{dx}{dt} = \lambda - \mu_{max} f(x) \tag{1}$$

where λ is the arrival rate, μ_{max} is the service rate and f a monotone function with the range $[0,1]$, [Agnew, 1976]. The response time is

$$T = t_0(1+x) = t_0\big(1+f^-(\rho)\big), \tag{2}$$

where $t_0 = {\mu_{max}}^{-1}$ is the average time to serve one customer when the queue is empty and ρ is the normalized service rate or the utility $\rho = \lambda/\mu_{max}$. For the simple M/M/1 queue we have $f = x/(x+1)$ [Tipper and Sundareshan, 1990].

If the function f in (1) is monotone the general behavior is that the response time increases with increasing arrival rate. The response time goes to infinity as λ approaches μ_{max} if the function f has the range $[0,1]$. Since the parameter μ_{max} is uncertain it may be desirable to have models where response rates increase significantly but that they do not go to infinity for finite λ, which can be accomplished by other choices of the function f.

When (1) is used to model an NE in Figure 1 the variable x accounts for the aggregated effect of the storage. Therefore x and T should be interpreted as apparent queue length and response time, they represent the aggregated behavior of many different queues in the real system. It is not possible to measure the apparent queue length directly but the response time can be measured. Requests that enter in a known way can also be used as an inputs.

Linearizing the model around the equilibrium x_e gives a first order system with the time constant

$$\tau = \frac{1}{\mu_{max} f'(x_e)} \geq \frac{1}{\mu_{max}}. \tag{3}$$

The inequality follows from f being monotone and $f(0) = 1$. Notice that the time constant increases significantly with increasing queue length.

4. Estimation

Different ways to estimate the response time from available measurements will now be discussed. There are significant variations in the arrival rate. The response time increases dramatically when the admission rate approaches the capacity of the system. The queue length x in the model (1) cannot be measured directly because it represents an aggregate effect of many queues as discussed in Section 3. It follows from (2) that a measurement of the response time T directly gives the queue length.

4.1 Exponential Smoothing

A simple way to estimate both response time and arrival rate is to use a moving average estimate. Since this estimator does not require a mathematical model it is used as a reference case. The estimator is given by

$$\hat{x}^+ = \hat{x} + k(x_m - \hat{x}) \tag{4}$$

where x_m is the measured quantity, $\hat{x}$, and $\hat{x}^+$ is the estimates before and after an event, and k is the filter gain. The filter can be used to estimate both response time and arrival rate. The filter coefficient can be chosen to minimize some measure of the error. The filter has the advantage that it does not require any model.

4.2 Kalman Filtering - Known Arrival Rate

In this case it is assumed that the arrival rate is measured and that the arrival time of each request and the time it takes to serve it are measured. There are significant variations in the response time. For a Poisson process the mean value and the variance are the same. A smoothed estimate is required to obtain information that is useful for control.

If the arrival rate λ is known, the apparent queue length can be predicted by the model (1) when there are no events. Hence

$$\frac{d\hat{x}}{dt} = \lambda - \mu_{max} f(\hat{x}) + k\left(T - t_0(1+\hat{x})\right), \quad \hat{T} = t_0(1+\hat{x}), \tag{5}$$

where the initial condition is taken as the estimate obtained at the most recent event. When an event occurs the estimate is updated as

$$\hat{x}^+ = \hat{x} + k(T - \hat{T}) \tag{6}$$

where T is the measured response time, $\hat{x}$ and $\hat{x}^+$ are estimates before and after an event, and k is a filter gain. The filter gain k can be computed if the statistics of $\hat{x}$ and T are known. Since it is unrealistic to assume that this information is available we will instead determine the filter gain from simulation and experiments.

4.3 Kalman Filtering - Unknown Arrival Rate

It is not always the case that all traffic can be controlled, so here we investigate if both arrival rate λ and queue length x can be estimated from measurements of response time. For simplicity we will assume that the arrival rate is constant but unknown or a random walk. Both assumptions lead to the same filter. Linearization of (1) and (2) around the equilibrium x_e, λ_e gives a dynamical system with

$$A = \begin{bmatrix} -\mu_{max} f'(x_e) & 1 \\ 0 & 0 \end{bmatrix}, \qquad C = \begin{bmatrix} t_0 & 0 \end{bmatrix}. \tag{7}$$

Estimation is possible because the system is observable. If the measurements were continuous the extended Kalman filter is

$$\begin{aligned} \frac{d\hat{x}}{dt} &= \hat{\lambda} - \mu_{max} f(\hat{x}) + k_1 \left(T - t_0(1+\hat{x})\right) \\ \frac{d\hat{\lambda}}{dt} &= k_2 \left(T - t_0(1+\hat{x})\right). \end{aligned}$$

When the measurements are event-based the model (1) is used to update the estimate when there are no events. The estimates are given by

$$\frac{d\hat{x}}{dt} = \hat{\lambda} - \mu_{max} f(\hat{x}), \qquad \frac{d\hat{\lambda}}{dt} = 0, \tag{8}$$

where the initial conditions are the estimates $x = \hat{x}^+$ and $\lambda = \hat{\lambda}^+$ obtained after a request has been serviced.

When a measurement of response time T is available the estimates are updated by

$$\begin{aligned} \hat{x}^+ &= \hat{x} + k_1 (T - \hat{T}) \\ \hat{\lambda}^+ &= \hat{\lambda} + k_2 (T - \hat{T}) \end{aligned} \tag{9}$$

where $\hat{T} = t_0(1+\hat{x})$ from the time at which the request entered the system.

4.4 Kalman Filtering - Two Arrival Streams

A characteristic feature of the system in Figure 1 is that there are two different input streams to the network elements. The stream coming from the service provider side is known but the traffic generated by the users enters the system in many different ways and cannot be measured. To capture this situation we will assume that there are two input streams. One stream is measured and the other is unknown, manifested only through variations in response time. The corresponding flow model is

$$\begin{aligned} \frac{dx}{dt} &= \lambda_c + \lambda_u - \mu_{max} f(x) \\ \hat{T} &= t_0(1+x), \end{aligned} \tag{10}$$

where λ_c is a controllable/known arrival rate and and λ_u is an uncontrollable/unknown arrival rate. Assuming that λ_u is a constant it follows that both x and λ_u are observable from measurements of T and λ_c. The event-based extended Kalman filter is obtained as a simple extension of the filter in Section 4.3.

5. Simulation

5.1 Introduction

To test the the estimators we will apply them to a known situation with an M/M/1 queue where many quantities can be evaluated analytically.

The simulated queue server system is a one-server, infinite queue system with exponentially distributed process times with mean ${\mu_{max}}^{-1}$. Jobs arrive at the queue, are finished in FIFO order and are acknowledged upon completion. The jobs were generated as a Poisson processes, with exponentially distributed inter-arrival times.

The arrival rate for the controllable stream coming from the service provider side in Figure 1, $\text{rate}_c(t)$, is a combination of a constant and a sine function variation. The arrival rate for the uncontrollable stream coming from the user side, $\text{rate}_u(t)$ was set as constant.

$$\begin{aligned} \text{rate}_c(t) &= C_c + a\sin(kt) \\ \text{rate}_u(t) &= C_u \end{aligned} \tag{11}$$

The parameters were chosen so that the system can handle the workload over long time but with periodic overloads, hence

$$\mu_{max} - a < C_c + C_u < \mu_{max}.$$

The numerical values used in the simulations are

$$\mu_{max} = 100, \qquad C_c = 42.5, \qquad C_u = 42.5, \qquad a = 20. \tag{12}$$

The same realizations were used in all simulations. For experiments with only one fully controllable stream, requests from both streams were directed to the controllable side.

The differential equations describing the behavior of the estimates between events were approximated using first order forward Euler discretization.

5.2 Exponential Smoothing

A simple way to estimate both response time and arrival rate is to use a moving average estimate. Since this estimator does not require a mathematical model it is used as a reference case. The estimator is given by (4). At the departure of a job the estimated queue length is updated as

$$\hat{x}^+ = \hat{x} + k_1\left(\frac{T}{t_0} - \frac{\hat{T}}{t_0}\right) \tag{13}$$

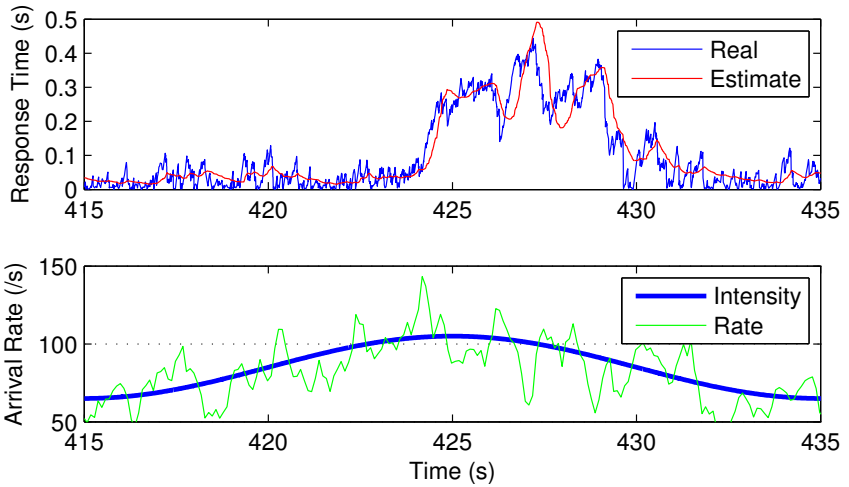

Figure 2. Estimation of response time by exponential smoothing.

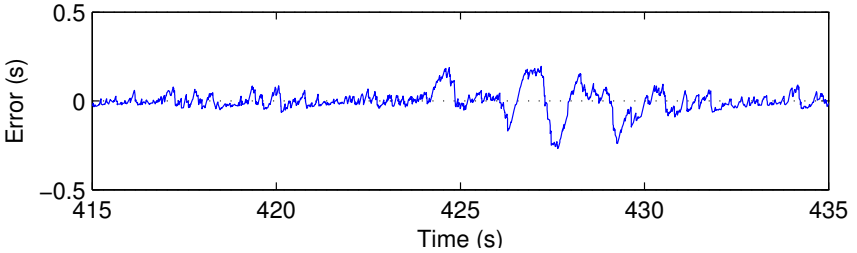

Figure 3. Error of the estimates in Figure 2. The mean square error is $\sigma = 0.018$. Notice the time variability of the error.

where T is the measured response time of the request and $\hat{T}$ is the response time estimation from when the request arrived at the system.

The choice of filter gain is a compromise, large values give a fast response with large fluctuations, small values give smoother estimates with slower response. After some experimentation the gain was chosen as $k = 0.03$, which corresponds to a time constant of about 30 events. Figure 2 shows that the simple exponential smoothing estimator gives reasonable results. It gives an efficient smoothing when the response times are small. There is however a lag in the response when the response times are changing significantly for example around times 424 and 427. The arrival rate is around 100 and the time delay is approximately 0.3 s, which matches the time constant of the estimator. The magnitude of the estimation error is very different at different periods the mean square error is $\sigma = 8.4 \cdot 10^{-4}$ in the interval $415 < t < 420$

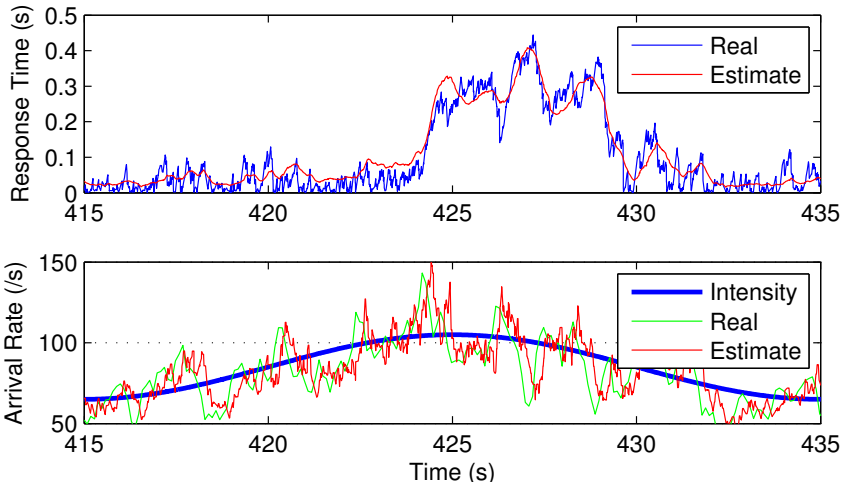

Figure 4. Estimates with known arrival rate. Arrival rates are smoothed and the response time is estimated using an event-based Kalman filter.

and $\sigma = 1.5 \cdot 10^{-2}$ in the interval $425 < t < 430$. The mean square error for the entire 600 second experiment is $\sigma = 1.8 \cdot 10^{-2}$. The different behaviors for different queue lengths indicate that it may be useful to schedule the estimator gains.

5.3 Known Arrival Rate

Since all traffic passes through the filter, exponential smoothing can be used to estimate the mean inter-arrival time which is the inverse of the arrival rate. This rate is used with the flow model (1) to estimate the response time using an extended Kalman filter. The estimates used on arrival are

$$\begin{aligned} \hat{i}^+ &= \hat{i} + k_3(h_a - \hat{i}) \\ \hat{\lambda}^+ &= (\hat{i}^+)^{-1} \\ \hat{x}^+ &= \hat{x} + h_a \left(\hat{\lambda}^+ - \mu f(\hat{x}) \right) \end{aligned} \tag{14}$$

where $\hat{i}$ is the estimate of the mean inter-arrival time, $\hat{\lambda}$ is the estimate of the arrival rate, $\hat{x}$ is the estimate of the effective queue length, and h_a is the time from the last arrival.

On departure the queue length is updated as

$$\hat{x}^+ = \hat{x} + k_1 \left(\frac{T}{t_0} - \frac{\hat{T}}{t_0} \right) \tag{15}$$

Figure 4 shows the arrival rate and the response time and their estimate. The error of the response time estimate is shown in Figure 5. A comparison with Fig-

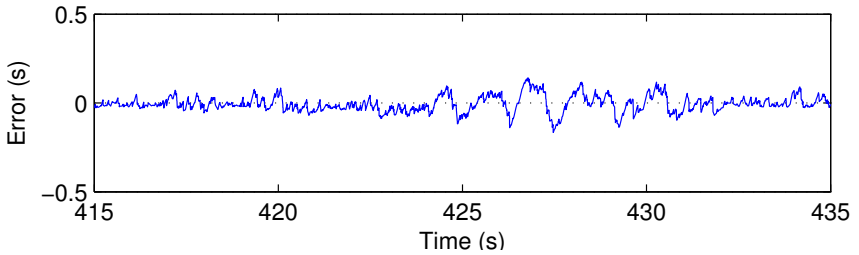

Figure 5. Error of the estimates in Figure 4. The mean square error is $\sigma = 0.0069$. Notice the time variability of the error.

ure 2 shows that a significant improvement is obtained at the times when the response times changes rapidly. Compare the behaviors around times 424 and 427. The improvement is particularly important to avoid overload during rapid increases in traffic. The magnitude of the estimation error is different at different periods the mean square error is $\sigma = 6.3 \cdot 10^{-4}$ in the interval $415 < t < 420$ and $\sigma = 4.5 \cdot 10^{-3}$ in the interval $425 < t < 430$. The total mean square error is $\sigma = 6.9 \cdot 10^{-3}$ which is significantly smaller that the error obtained by the simple exponential smoothing estimate which had $\sigma = 1.8 \cdot 10^{-2}$.

5.4 Two Arrival Streams

In this experiment we separate the two streams of traffic to simulate the two sides of the NEs in Figure 1. One stream passes the observer and one stream enters the shared resource in the background, only showing itself as an added load on the system.

Running this scenario with the simple exponential smoothing estimator presented in section 5.2 results in the response times and estimations shown in Figure 6. The estimation error is shown in Figure 7. Since the filter gets only half the amount of measurements, this situation is not identical to Figure 2. Here the mean square error is $\sigma = 8.6 \cdot 10^{-4}$ for the period $415 < t < 420$ and $\sigma = 1.1 \cdot 10^{-2}$ for the period $425 < t < 430$. The mean square error for the entire experiment is $\sigma = 9.9 \cdot 10^{-3}$.

To try the Kalman filter we use the parameters shown in table 1. The observer follows the inter-arrival times of the controllable traffic using the exponential smoothing described in Section 5.3 and equation (14). The controllable arrival rate is then

$$\hat{\lambda}_c^+ = \hat{i}^{-1}. \tag{16}$$

Table 1. Parameters used in this experiment

k_1	k_2	k_3
250	110	0.031

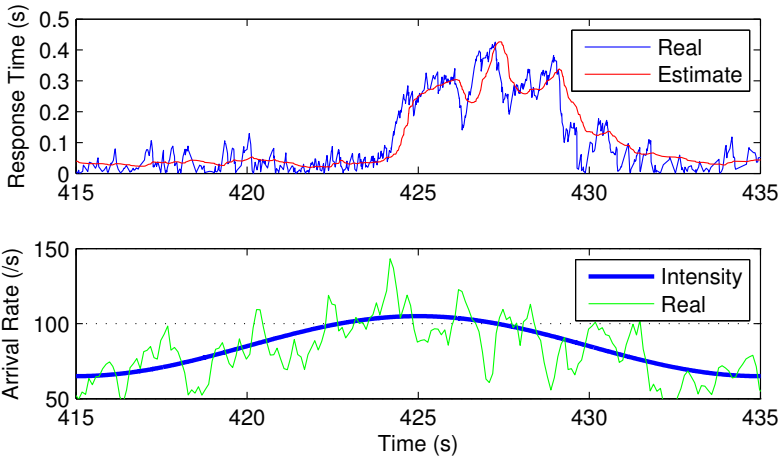

Figure 6. Exponential smoothing response times with estimate, arrival rate has no estimate.

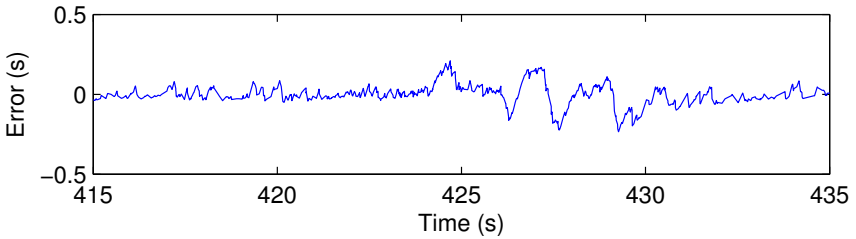

Figure 7. Exponential smoothing prediction error.

On every departure we get a response time measurement and update the estimation of the queue length and uncontrollable arrival rate as

$$\begin{aligned} \hat{x}^+ &= \hat{x} + h_d \left(\hat{\lambda}_c + \hat{\lambda}_u - \mu f(\hat{x}) + k_1 (T - \hat{T}) \right) \\ \hat{\lambda}_u^+ &= \hat{\lambda}_u + h_d k_2 (T - \hat{T}) \end{aligned} \tag{17}$$

where h_d is the time since the last departure. Figure 8 shows the response times and the arrival rate, both real values and estimates. The estimate error is shown in Figure 9. Once again we can see how the Kalman filter manages to follow the real system during the quick rises in response time around time 424 and 427. Here the mean square error is $\sigma = 7.4 \cdot 10^{-4}$ for the period $415 < t < 420$ and $\sigma = 1.1 \cdot 10^{-2}$ for the period $425 < t < 430$. The mean square error for the entire experiment is $\sigma = 1.9 \cdot 10^{-2}$.

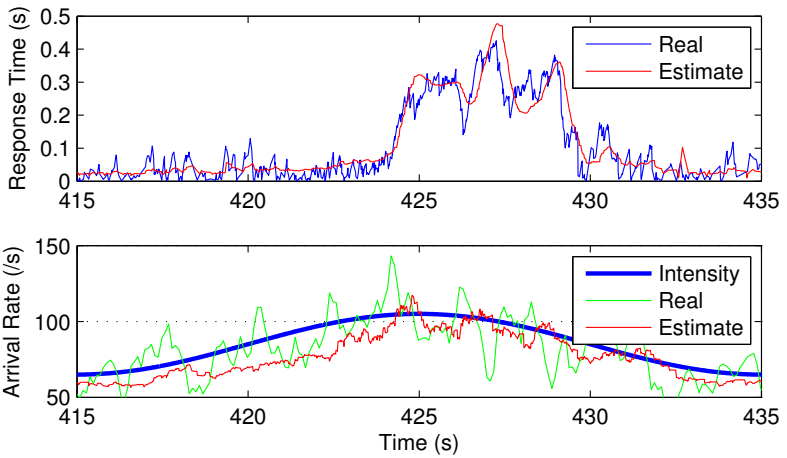

Figure 8. Kalman filter response times with estimate, arrival rate with estimate.

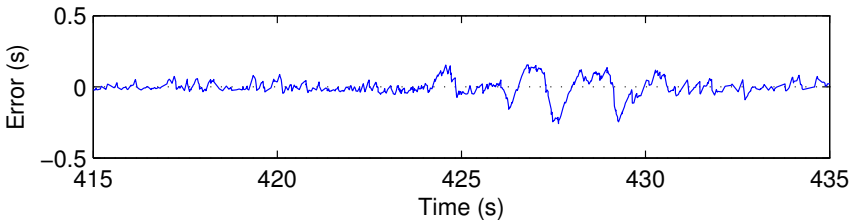

Figure 9. Kalman filter prediction error.

6. Fundamental Limitations

It is useful to know the factors that fundamentally limits how accurate the response time can be estimated. Since response times are stochastic, our best guess is the expected value. If there are n jobs in the system, our best guess will be that the next job will take $t_0 \cdot (1+n)$, where $t_0 = \mathrm{E}[X], X \sim \mathrm{Exp}(\mu_{max})$. However, it will actually take $\sum_{i=1}^{n+1} X_i, X_i \sim \mathrm{Exp}(\mu_{max})$ which is a stochastic variable. Since the sum of several exponentially distributed variables follows the *Erlang distribution* the expected value of the error as a function of queue length will be

$$E_{err}(n) = \mathrm{E}[|\mathrm{E}[Y] - Y|]$$

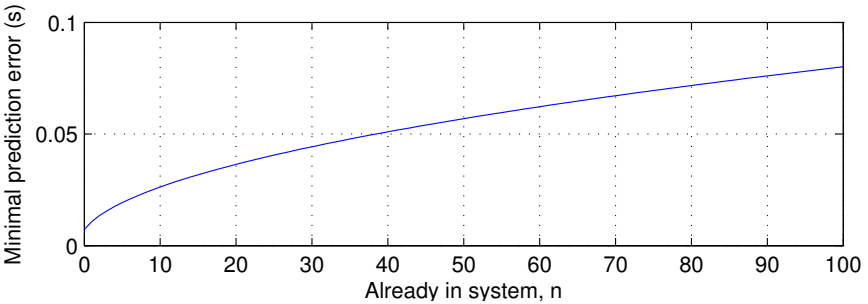

Figure 10. Minimal prediction error for a M/M/1 queue system with $\mu_{max} = 100$.

where $Y \sim \text{Erlang}(n+1, \mu_{max})$ and n is the number of requests already in the system. This gives us the following calculations for the minimal error:

$$E_{err}(n) = \frac{{\mu_{max}}^{n+1}}{n!} \int_0^\infty \left| \frac{n+1}{\mu_{max}} - x \right| x^n e^{-\mu_{max} x} \text{dx} = \frac{2(n+1)^{n+1}}{n! \mu_{max} e^{n+1}}.$$

The smallest value is obtained for $n = 0$.

Figure 10 shows the minimal prediction error as a function of queue length with $\mu_{max} = 100$.

7. Summary

Feedback control is essential for resource management in computer systems. We have investigated several ways of estimating response time which is a key measure of service quality. Simple estimators that do not require models as well as more sophisticated model based schemes have been investigated. The model-based estimators use flow models of the queuing systems and provide event-based estimates using extended Kalman filtering. The estimators have been tested by simulation for scenarios for resource management for mobile telephone operators. The simple model-free estimators give reasonable estimates but the estimates are delayed when the the queue length increases due to system overload. The delay can be reduced by using model-based estimators both in the case of a measured incoming traffic and when the incoming traffic is a mix of known and unknown background traffic.

8. Acknowledgment

This work has been partly funded by the Lund Center for Control of Complex Engineering Systems (LCCC) and the Swedish Research Council grant VR 2010-5864.

References

Agnew, C. E. (1976). "Dynamic modeling and control of congestion-prone systems". *Operations Research* **24**:3, pp. 400–419.

Bianchini, R. and R. Rajamony (2004). "Power and energy management for server systems". *IEEE Computer* **37**:11.

Brawn, B. and F. Gustavson (1968). "Program behavior in a paging environment". *Proceedings of the AFIPS Fall Joint Computer Conference*, pp. 1019–1032.

Cao, J., M. Andersson, C. Nyberg, and M. Kihl (2003). "Web server performance modeling using an M/G/1/K*PS queue". In: *10th International Conference on Telecommunications*.

Chen, X., H. Chen, and P. Mohapatra (2003). "Aces: an efficient admission control scheme for QoS-aware web servers". *Computer Communication* **26**:14.

Claussen, H., L. Ho, and F. Pivit (2009). "Leveraging advances in mobile broadband technology to improve environmental sustainability". *Telecommunications Journal of Australia* **59**:1.

Crocus (1975). *Systemes d'Exploitation des Ordinateurs*. Dunod, Paris.

Diao, Y., C. Wu, J. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, L. Chu, and J. Colaco (2005). "Comparative studies of load balancing with control and optimization techniques". In: *American Control Conference*.

Dilley, J., R. Friedrich, T. Jin, and J. Rolia (1998). "Web server performance measurement and modeling techniques". *Performance Evaluation* **33**:1.

Elnozahy, E., M. Kistler, and R. Rajamony (2003). "Energy-efficient server clusters". In: *Lecture Notes in Computer Science 2325*. Springer-Verlag Berlin Heidelberg.

Fu, Y., H. Wang, C. Lu, and R. Chandra (2006). "Distributed utilization control for real-time clusters with load balancing". In: *IEEE International Real-Time Systems Symposium*.

Gilly, K., C. Juiz, S. Alcaraz, and R. Puigjaner (2009). "Adaptive admission control algorithm in a QoS-aware web system". In: *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*.

Hellerstein, J., Y. Diao, S. Parekh, and D. Tilbury (2005). "Control engineering for computing systems". *IEEE Control System Magazine* **25**:6.

Henriksson, D., Y. Lu, and T. Abdelzaher (2004). "Improved prediction for web server delay control". In: *16th Euromicro Conference on Real-Time Systems*.

Horvath, T., T. Abdelzaher, K. Skadron, and X. Liu (2007). "Dynamic voltage scaling in multitier web servers with end-to-end delay control". *IEEE Transactions on Computers* **56**:4.

Kihl, M., A. Robertsson, M. Andersson, and B. Wittenmark (2008). "Control theoretic analysis of admission control mechanisms for web server systems". *The World Wide Web Journal* **11**:1.

Kjaer, M., M. Kihl, and A. Robertsson (2009). "Resource allocation and disturbance rejection in web servers using SLAs and virtualized servers". *IEEE Transaction on Network and Service Management* **6**:4.

Liu, X., J. Heo, L. Sha, and X. Zhu (2006). "Adaptive control of multi-tiered web applications using queuing predictor". In: *10th IEEE/IFIP Network Operation Mangement Symposium*.

Mei, R. D. van der, R. Hariharan, and P. K. Reeser (2001). "Web server performance modeling". *Telecommunication Systems* **16**:3.

Menascé, D. A. and V. A. F. Almeida (2002). *Capacity Planning for Web Services*. Prentice Hall.

Tipper, D. and M. Sundareshan (1990). "Numerical methods for modeling computer networks under nonstationary conditions". *IEEE Journal on Selected Areas in Communications* **8**:9, pp. 1682 –1695.

Paper IV

Control-Theoretical Load-Balancing for Cloud Applications with Brownout

Jonas Dürango[1] **Manfred Dellkrantz**[1] **Martina Maggio**[1]
Cristian Klein[2] **Alessandro Vittorio Papadopoulos**[1]
Francisco Hernández-Rodriguez[2] **Erik Elmroth**[2] **Karl-Erik Årzén**[1]

Abstract

Cloud applications are often subject to unexpected events like flash crowds and hardware failures. Without a predictable behaviour, users may abandon an unresponsive application. This problem has been partially solved on two separate fronts: first, by adding a self-adaptive feature called *brownout* inside cloud applications to bound response times by modulating user experience, and, second, by introducing replicas — copies of the applications having the same functionalities — for redundancy and adding a load-balancer to direct incoming traffic.

However, existing load-balancing strategies interfere with brownout self-adaptivity. Load-balancers are often based on response times, that are already controlled by the self-adaptive features of the application, hence they are not a good indicator of how well a replica is performing.

In this paper, we present novel load-balancing strategies, specifically designed to support brownout applications. They base their decision not on response time, but on user experience degradation. We implemented our strategies in a self-adaptive application simulator, together with some state-of-the-art solutions. Results obtained in multiple scenarios show that the proposed strategies bring significant improvements when compared to the state-of-the-art ones.

1. Introduction

Cloud computing has dramatically changed the management of computing infrastructures. On one hand, public infrastructure providers, such as Amazon EC2, allow service providers, such as Dropbox and Netflix, to deploy their services on large infrastructures with no upfront cost [Buyya et al., 2009], by simply leasing computing capacity in the form of VMs. On the other hand, the flexibility offered by cloud technologies, which allow VMs to be hosted by any Physical Machine (PM) (or server), favors the adoption of private clouds [Gulati et al., 2011]. Therefore, self-hosting service providers themselves are converting their computing infrastructures into small clouds.

One of the main issues with cloud computing infrastructures is **application robustness** to unexpected events. For example, flash-crowds are sudden increments of end-users, that may raise the required capacity up to five times [Bodik et al., 2010]. Similarly, hardware failures may temporarily reduce the capacity of the infrastructure, while the failure is repaired [Barroso and Hölzle, 2009]. Also, unexpected performance degradations may arise due to workload consolidation and the resulting interference among co-located applications [Mars et al., 2011]. Due to the large magnitude and short duration of such events, it may be economically too costly to keep enough spare capacity to properly deal with them. As a result, unexpected events may lead to infrastructure overload, that translates to unresponsive services, leading to dissatisfied end-users and revenue loss.

Cloud services therefore greatly benefit from self-adaptation techniques [Salehie and Tahvildari, 2009], such as **brownout** [Klein et al., 2014; Maggio et al., 2014]. A brownout service adapts itself by reducing the amount of computations it executes to serve a request, so as to maintain response time around a given setpoint. In essence, some computations are marked as mandatory — for example, displaying product information in an e-commerce website — while others are optional — for example, recommending similar products. Whenever an end-user request is received, the service can choose to execute the optional code or not according to its available capacity, and to the previously measured response times. Note that executing optional code directly translates into a better service for the end-user and more revenue for the service provider. This approach has proved to be successful for dealing with unexpected events [Klein et al., 2014]. However, there, brownout services were composed of a single *replica*, i.e., a single copy of the application, running inside a single VM.

In this paper, we extend the brownout paradigm to services featuring multiple replicas — i.e., multiple, independent copies of the same application, serving the user the same data — hosted inside individual VMs. Since each VM can be hosted by different PMs, this enhances brownout services in two directions. First, *scalability* of a brownout application — the ability for an application to deal with more users by adding more computing resources — is improved, since applications are no longer limited to using the resources of a single PM. Second, resilience is improved:

in case a PM fails, taking down a replica, other replicas whose VMs are hosted on different PMs can seamlessly take over.

The component that decides which replica should serve a particular end-user request is called a *load-balancer*. Despite the fact that load-balancing techniques have been widely studied [Barroso and Hölzle, 2009; Lu et al., 2011; Lin et al., 2012; Nakrani and Tovey, 2004], state-of-the-art load-balancers forward requests based on metrics that cannot discriminate between a replica that is avoiding overload by not executing the optional code and a replica that is not subject to overload. Therefore, the novelty of our problem consists in finding a brownout-compliant load-balancing technique that is aware of each replica's self-adaptation mechanism.

The contribution of this paper is summarized as follows.

- We present extensions to load-balancing architectures and the required enhancements to the replicas that convey information about served optional content and allow to deal with brownout services efficiently (Section 3).
- We propose novel load-balancing algorithms that, by receiving information about the adaptation happening at the replica level, try to maximize the performance of brownout services, in terms of frequency of execution of the optional code (Section 4).
- We show through simulations that our brownout-aware load-balancing algorithms outperform state-of-the-art techniques (Section 5).

2. Related Work

Load-balancers are standard components of Internet-scale services [Wang et al., 2002], allowing applications to achieve scalability and resilience [Barroso and Hölzle, 2009; Hamilton, 2007; Wolf and Yu, 2001]. Many load-balancing policies have been proposed, aiming at different optimizations, spanning from equalizing processor load [Stankovic, 1985] to managing memory pools [Patterson et al., 1995; Diao et al., 2005], to specific optimizations for iterative algorithms [Bahi et al., 2005]. Often load-balancing policies consider web server systems as a target [Manfredi et al., 2013; Cardellini et al., 2003], where one of the most important result is to bound the maximum response time that the clients are exposed to [Huang and Abdelzaher, 2005]. Load-balancing strategies can be guided by many different purposes, for example geographical [Andreolini et al., 2008; Ranjan et al., 2004], driven by the electricity price to reduce the datacenter operation cost [Doyle et al., 2013], or specifically designed for cloud applications [Barroso and Hölzle, 2009; Lu et al., 2011; Lin et al., 2012].

Load-balancing solutions can be divided into two different types: static and dynamic. Static load-balancing refers to a fixed, non-adaptive strategy to select a replica to direct traffic to [Ni and Hwang, 1985; Tantawi and Towsley, 1985]. The

most commonly used technique is based on selecting each replica in turn, called **Round Robin (RR)**. It can be either deterministic, storing the last selected replica, or probabilistic, picking a replica at **Random**. However, due to their static nature, such techniques would not have good performance when applied to brownout-compliant applications as they do not take into account the inherent fluctuations of a cloud environment and the control strategy at the replica level, which leads to changing capabilities of replicas.

On the contrary, dynamic load-balancing is based on measurements of the current system's state. One popular option is to choose the replica which had the lowest response time in the past. We refer to this algorithm as **Fastest Replica First (FRF)** if the choice is based on the last measured response time of each replica, and **FRF-EWMA** if the choice is based on an Exponentially Weighted Moving Average over the past response times of each replica. A variation of this algorithm is **Two Random Choices (2RC)** [Mitzenmacher, 2001], that randomly chooses two replicas and assigns the request to the fastest one, i.e., the one with the lowest maximum response time.

Through experimental results, we determined that FRF, FRF-EWMA and 2RC are unsuitable for brownout applications. They base their decision on response times alone, which leads to inefficient decisions for brownout services. Indeed, such services already keep their response-time at a given setpoint, at the expense of reducing the ratio of optional content served. Hence, by measuring response-time alone, it is not possible to discriminate between a replica that is avoiding overload by not executing the optional code and a replica that is not subject to overload executing all optional code, both achieving the desired response times.

Another adopted strategy is based on the pending request count and generally called **Shortest Queue First (SQF)**, where the load-balancer tracks the pending requests and select the replicas with the least number of requests waiting for completion. This strategy pays off in architectures where the replicas have similar capacities and the requests are homogeneous. To account for non-homogeneity, Pao and Chen proposed a load balancing solution using the remaining capacity of the replicas to determine how the next request should be managed [Pao and Chen, 2006]. The capacity is determined through a combination of factors like the remaining available CPU and memory, the network transmission and the current pending request count. Other approaches have been proposed that base their decision on remaining capacity. However, due to the fact that brownout applications indirectly control CPU utilization, by adjusting the execution of optional content, so as to prepare for possible request bursts, deciding on remaining capacity alone is not an indicator of how a brownout replica is performing.

A merge of the fastest replica and the pending request count approach was implemented in the BIG-IP Local Traffic Manager [*BIG-IP Local Traffic Manager*], where the replicas are ranked based on a linear combination of response times and number of routed requests. Since the exact specification of this algorithm is not open, we tried to mimic as follows: A **Predictive** load balancer would rank the

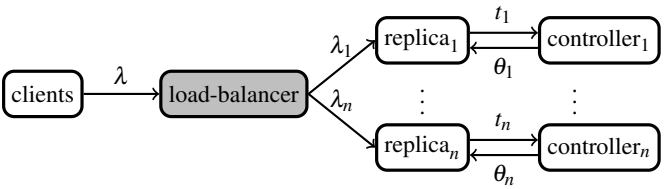

Figure 1. Architecture of a brownout-compliant cloud application featuring multiple replicas.

replicas based on the difference between the past metrics and the current ones. One of the solutions proposed in this paper extends the idea of looking at the difference between the past behavior and the current one, although our solution observes the changes in the ratio of optional code served and tries to maximize the requests served enabling the full computation.

Dynamic solutions can be control-theoretical [Zhang et al., 2002; Kameda et al., 2000] and also account for the cost of applying the control action [Diao et al., 2004] or for the load trend [Casolari et al., 2009]. This is especially necessary when the load balancer also acts as a resource allocator deciding not only where to route the current request but also how much resources it would have to execute, like in [Ardagna et al., 2012]. In these cases, the induced sudden lack of resources can result in poor performance. However, we focus only on load-balancing solutions, since brownout applications are already taking care of the potential lack of resources [Klein et al., 2014].

3. Problem Statement

Load-balancing problems can be formulated in many ways. This is especially true for the case addressed in this paper where the load-balancer should distribute the load to adaptive entities, that play a role by themselves in adjusting to the current situation. This section discusses the characteristics of the considered infrastructure and clearly formulates the problem under analysis.

Figure 1 illustrates the software architecture that is deployed to execute a brownout-compliant application composed of multiple replicas. Despite the modifications needed to make it brownout-compliant, the architecture is widely accepted as the reference one for cloud applications [Barroso and Hölzle, 2009].

Given the generic cloud application architecture, access can only be done through the load-balancer. The clients are assumed to be closed-loop: They first send a request, wait for the reply, then think by waiting for an exponentially distributed time interval, and repeat. This client model is a fairly good approximation for users that interact with web-sites requiring a pre-defined number of requests to complete a goal, such as buying a product [García and García, 2003] or booking a flight. The resulting traffic has an unknown but measurable rate λ.

Each client request is received by the load-balancer, that sends it to one of the n replicas. The chosen replica produces the response and sends it back to the load-balancer, which forwards it to the original client. We measure the **response time** of the request as the time spent within the replica, assuming negligible time is taken for the load-balancer execution and for the routing itself. Since the responses are routed back to the load-balancer, it is possible to attach information to be routed back to aid balancing decisions to it.

Each replica i receives a fraction λ_i of the incoming traffic and is a stand-alone version of the application. More specifically, each replica receives requests at a rate $\lambda_i = w_i \cdot \lambda$, such that $w_i \geq 0$, and $\sum_i w_i = 1$. In this case, the load balancer simply computes the **replica weights** w_i according to its load-balancing policy.

Special to our case is the presence of a controller within each replica [Klein et al., 2014]. This controller receives periodic measurements of the response time t_i of the requests served by the replica, and adjusts the percentage of requests θ_i served with optional components. Here t_i is the 95-th percentile of the response times for a control period. Following the approach of [Klein et al., 2014], we model the response times from a replica as

$$t_i^{k+1} = \alpha_i^k \cdot \theta_i^k$$

where α_i^k is an unknown parameter estimated online (details omitted here). The control loop is then closed using the PI controller

$$\theta_i^{k+1} = \theta_i^k + \frac{1 - p_1}{\hat{\alpha}_i^k} \cdot e_i^{k+1}$$

where e_i^{k+1} is the control error and p_1 the closed-loop pole. As the controller output is restricted, anti-windup measures are employed. In our experiments, p_1 is set to 0.99, the replica control period is to 0.5s, while the load-balancer acts every second.

As given by the brownout paradigm, a replica i responds to requests either partially, where only mandatory content is included in the reply, or fully, where both mandatory and optional content is included. This decision is taken independently for each request with a probability θ_i for success. The service rate for a partial response is μ_i while a full response is generated with a rate M_i. Obviously, partial replies are faster to compute than full ones, hence, $\mu_i \geq M_i$. Assuming the replica is not saturated, it serves requests fully at a rate $\lambda_i \theta_i$ and partially at a rate $\lambda_i(1 - \theta_i)$.

Many alternatives can be envisioned on how to extend existing load balancers to deal with brownout-compliant applications. In our choice, the load-balancer receives information about θ_i from the replicas. This solution results in less computationally intensive load-balancers with respect to the case where the load-balancer should somehow estimate the probability of executing the optional components, but requires additional communication. The overhead, however, is very limited, since only one value would be reported per replica. For the purpose of this paper, we assume that to aid load-balancing decisions, each replica piggy-backs the current

value of θ_i through the reply, so that this value can be observed by the load-balancer, limiting the overhead. The load-balancer does not have any knowledge on *how* each replica controller adjusts the percentage θ_i, it only knows the reported value. This allows to completely separate the action of the load- balancer from the one of the self-adaptive application.

Given this last architecture, we want to solve the problem of designing a **load-balancer policy**. Knowing the values of θ_i for each replica $i \in [1,n]$, a load-balancer should compute the values of the weights w_i such that

$$\sum_{k=0}^{\infty} \sum_{i} w_i(k)\theta_i(k) \tag{1}$$

is maximized, where k denotes the discrete time. Given that we have no knowledge of the evolution in time of the involved quantities, we aim to maximize the quantity $\sum_i w_i\theta_i$ in every time instant, assuming that this will maximize the quantity defined in Equation (1). In other words, the load-balancer should maximize the ratio of requests served with the optional part enabled. For that, the aim is to maximize the ratio of optional components served in any time instant. In practice, this would also maximize the application owner's revenue [Klein et al., 2014].

4. Solution

This section describes three different solutions for balancing the load directed to self-adaptive brownout-compliant applications composed of multiple replicas. The first two strategies are heuristic solutions that take into account the self-adaptivity of the replicas. The third alternative is based on optimization, with the aim of providing guarantees on the best possible behavior.

4.1 Variational principle-based heuristic (VPBH)

Our first solution is inspired by the predictive approach described in Section 2. The core of the predictive solution is to examine the variation of the involved quantities. While in its classical form, this solution relies on variations of response times or pending request count per replica, our solution is based on how the control variables θ_i are changing.

If the percentage θ_i of optional content served is increasing, the replica is assumed to be less loaded, and more traffic can be sent to it. On the contrary, when the optional content decreases, the replica will receive less traffic, to decrease its load and allow it to increase θ_i.

The replica weights w_i are initialized to $1/n$ where n is the number of replicas. The load-balancer periodically updates the values of the weights based on the values of θ_i received by the replicas. At time k, denoting with $\Delta\theta_i(k)$ the variation $\theta_i(k) - \theta_i(k-1)$, the solution computes a potential weight $\tilde{w}_i(k+1)$ according to

$$\tilde{w}_i(k+1) = w_i(k) \cdot [1 + \gamma_P \Delta\theta_i(k) + \gamma_I \theta_i(k)], \tag{2}$$

where γ_P and γ_I are constant gains, respectively related to a proportional and an integral load-balancing action. As calculated, $\tilde{w}_i$ values can be negative. This is clearly not feasible, therefore negative values are truncated to a small but still positive weight ε. Using a positive weight instead of zero allows us to probe the replica and see whether it is favorably responding to new incoming requests or not. Moreover, the computed values do not respect the constraint that their sum is equal to 1, so they are then re-scaled according to

$$w_i(k) = \frac{\max(\tilde{w}_i(k), \varepsilon)}{\sum_i \max(\tilde{w}_i(k), \varepsilon)}. \tag{3}$$

We selected $\gamma_P = 0.5$ based on experimental results. Once γ_P is fixed to a selected value, increasing the integral gain γ_I calls for a stronger action on the load-balancing side, which means that the load-balancer would take decisions very much influenced by the current values of θ_i, therefore greatly improving performance at the cost of a more aggressive control action. On the contrary, decreasing γ_I would smoothen the control signal, possibly resulting in performance loss due to a slower reaction time. The choice of the integral gain allows to exploit the trade-off between performance and robustness. For the experiments we chose $\gamma_I = 5.0$.

4.2 Equality principle-based heuristic (EPBH)

The second policy is based on the heuristic that a near-optimal situation is when all replica serves the same percentage optional content. Based on this assumption, the control variables θ_i should be as close as possible to one another. If the values of θ_i converge to a single value, this means that the traffic is routed so that each replica can serve the same percentage of optional content, i.e., a more powerful replica receives more traffic then a less powerful one. This approach therefore selects weights that encourages the control variables θ_i to converge towards the mean $\frac{1}{n}\sum_j \theta_j$.

The policy computes a potential weight $\tilde{w}_i(k+1)$

$$\tilde{w}_i(k+1) = w_i(k) + \gamma_e \left(\theta_i(k) - \frac{1}{n} \sum_j \theta_j(k) \right) \tag{4}$$

where γ_e is a strictly positive parameter which accounts for how fast the algorithm should converge. For the experiments we chose $\gamma_e = 0.025$. The weights are simply modified proportionally to the difference between the current control value and the average control value set by the replicas. Clearly, the same saturation and normalization described in Equation (3) has to be applied to the proposed solution, to ensure that the sum of the weights is equal to one and that they have positive values — i.e., that all the incoming traffic is directed to the replicas and that each replica receives at least some requests.

4.3 Convex optimization based load-balancing (COBLB)

The third approach is to update the replica weights based on the solution of an optimization problem, where the objective is to maximize the quantity $\sum_i w_i \theta_i$.

In this solution, each replica is modeled as a queuing system using a Processor Sharing (PS) discipline. The clients are assumed to arrive according to a Poisson process with intensity λ_i, and will upon arrival enter the queue where they will receive a share of the replicas processing capability. The simplest queueing models assume the required time for serving a request to be exponentially distributed with rate $\tilde{\mu}$. However, in the case of brownout, the requests are served either with or without optional content with rates M_i and μ_i, respectively. Therefore the distribution of service times S_i for the replicas can be modelled as a mixture of two exponential distributions with a probability density function $f_{S_i}(t)$ according to

$$f_{S_i}(t) = (1-\theta_i) \cdot \mu_i \cdot e^{-\mu_i \cdot t} + \theta_i \cdot M_i \cdot e^{-M_i \cdot t}, \tag{5}$$

where t represents the continuous time and θ_i is the probability of activating the optional components. Thus, a request entering the queue of replica i will receive an exponentially distributed service time with a rate with probability θ_i being M_i, and probability $1-\theta_i$ being μ_i. The resulting queueing system model is of type $M/G/1/PS$ and has been proven suitable to simulate the behavior of web servers [Cao et al., 2003].

It is known that for $M/G/1$ queueing systems adopting the PS discipline, the mean response times will depend on the service time distribution only through its mean [Kleinrock, 1967; Sakata et al., 1971], here given for each replica by

$$\mu_i^* = \frac{1}{\mathbb{E}[S_i]} = \left[\frac{1-\theta_i}{\mu_i} + \frac{\theta_i}{M_i} \right]^{-1}. \tag{6}$$

The mean response times for a $M/G/1/PS$ system themselves are given by

$$\tau_i = \frac{1}{\mu_i^* - \lambda w_i}. \tag{7}$$

The required service rates μ_i^* needed to ensure that there is no stationary error can be obtained by inverting Equation (7)

$$\mu_i^* = \frac{1 + \tau_i^* \lambda w_i}{\tau_i^*} \tag{8}$$

with τ_i^* being the set point for the response time of replica i.

Combining Equation (6) and (8), it is then possible to calculate the steady-state control variables θ_i^* that gives the desired behavior

$$\theta_i^* = \frac{M_i \cdot (\mu_i \tau_i^* - 1 - \lambda w_i \tau_i^*)}{(1 + \lambda w_i \tau_i^*) \cdot (\mu_i - M_i)} = \frac{A_i - B_i w_i}{C_i + D_i w_i}. \tag{9}$$

with A_i, B_i, C_i and D_i all positive. Note that the values of θ_i^* are not used in the replicas and are simply computed by the optimization based load-balancer as the optimal stationary conditions for the control variables θ_i. Clearly, one could also think of using these values within the replicas but in this investigation we want to completely separate the load-balancing policy and the replicas internal control loops.

Recalling that θ_i is the probability of executing the optional components when producing the response, the values θ_i^* should be constrained to belong to the interval $[0,1]$, yielding the following inequalities (under the reasonable assumptions that $\tau_i^* > 1/M_i$ and $\mu_i \geq M_i$)

$$\frac{A_i - C_i}{B_i + D_i} \leq w_i \leq \frac{A_i}{B_i}. \tag{10}$$

Using these inequalities as constraints, it is possible to formally state the optimization problem as

$$\begin{aligned} \max_{w_i} \quad & J = \textstyle\sum_i w_i \theta_i = \sum_i w_i \dfrac{A_i - B_i w_i}{C_i + D_i w_i} \\ \text{s.t.} \quad & \textstyle\sum_i w_i = 1, \\ & \dfrac{A_i - C_i}{B_i + D_i} \leq w_i \leq \dfrac{A_i}{B_i}. \end{aligned} \tag{11}$$

Since the objective function J is concave and the constraints linear in w_i, the entire problem is concave and can be solved using efficient methods [Boyd and Vandenberghe, 2004]. We use an interior point algorithm, implemented in CVXOPT[1], a Python library for convex optimization problems, to obtain the values of the weights.

Notice that solving optimization problem (11) guarantees that the best possible solution is found for the single time instant problem, but requires a lot of knowledge about the single replicas. In fact, while other solutions require knowledge only about the incoming traffic and the control variables for each replica, the optimization-based solution relies on knowledge of the service time of requests with and without optional content M_i and μ_i that might not be available and could require additional computations to be estimated correctly.

5. Evaluation

In this section we describe our experimental evaluation, discussing the performance indicators used to compare different strategies, the simulator developed and used to emulate the behavior of brownout-compliant replicas driven by the load-balancer, and our case studies.

[1] `http://cvxopt.org/`

5.1 Performance indicators

Performance measures are necessary to objectively compare different algorithms. Our first performance indicator is defined as the **percentage** $\%_{oc}$ of the total requests served with the optional content enabled, which is a reasonable metric given that we assume that users perform a certain number of clicks to use the application.

We also would like to introduce some other performance metrics to compare the implemented load-balancing techniques. For this, we use the **user-perceived stability** σ_u [Andreolini et al., 2008]. This metric refers to the variation of performance as observed by the users, and it is measured as the standard deviation of response times. Its purpose is to measure the ability of the replicas to respond timely to the client requests. The entire brownout framework aims at stabilizing the response times, therefore it should achieve better user-perceived stability, regardless of the presence of the load-balancer. However, the load-balancing algorithm clearly influences the perceived response times, therefore it is logical to check whether the newly developed algorithms achieve a better perceived stability than the classical ones. Together with the value of the user-perceived stability, we also report the **average response time** μ_u to distinguish between algorithms that achieve a low response time with possibly high fluctuations from solutions that achieve a higher but more stable response time.

5.2 Simulator

To test the load-balancing strategies, a Python-based simulator for brownout-compliant applications is used. In the simulator, it is easy to plug-in new load-balancing algorithms. The simulator is based on the concepts of *Client*, *Request*, *LoadBalancer* and *Replica*.

When a new client is defined, it can behave according to the open-loop client model, where it simply issues a certain number of unrelated requests (as it is true for clients that respect the Markovian assumption), or according to the closed-loop one [Schroeder et al., 2006; Alomari and Menascé, 2013]. Closed-loop clients issue a request and wait for the response, when they receive the response they think for some time (in the simulations this time is exponentially distributed with mean 1s) and subsequently continue sending another request to the application. While this second model is more realistic, the first one is still useful to simulate the behavior of a large number of clients. The simulator implements both models, to allow for complete tests, but we will evaluate our results with closed-loop clients given the nature of the applications, that requires users to perform a certain number of clicks.

Requests are received by the load-balancer, that directs them towards different replicas. The load-balancer can work on a per-request basis or based on weights. The first case is used to simulate policies like Round Robin, Random, Shortest Queue First and so on, that do not rely on the concept of weights. The weighted load-balancer is used to simulate the strategies proposed in this paper.

Each replica simulates the computation necessary to serve the request and

chooses if it should be executed with or without the optional components activated. If the optional content is served the service time is a random number from a gaussian distribution with mean ϕ_i and variance 0.01, while if the optional content is not served, the mean is ψ_i and the variance is 0.001. The parameters ϕ_i and ψ_i are specified when replicas are created and can be changed during the execution. The service rate of requests with the optional component is $M_i = 1/\phi_i$ while for serving only the mandatory part of the request the service rate is $\mu_i = 1/\psi_i$. The replicas are also executing an internal control loop to select their control variables θ_i [Klein et al., 2014]. The replicas use PS to process the requests in the queue, meaning that each of the n active requests will get $1/n$ of the processing capability of the replica.

The simulator receives as input a *Scenario*, which describes what can happen during the simulation. The scenario definition supports the insertion of new clients and the removal of existing ones. It also allows to turn on and off replicas at specific times during the execution and to change the service times for every replica, both for the optional components and for the mandatory ones. This simulates a change in the amount of resources given to the machine hosting the replica and it is based on the assumption that these changes are unpredictable and can happen at the architecture level, for example due to the cloud provider co-locating more applications onto the same physical hardware, therefore reducing their computation capability [Tomás and Tordsson, 2013].

With the scenarios, it is easy to simulate different working conditions and to have a complete overview of the changes that might happen during the load-balancing and replica execution. In the following, we describe two experiments conducted to compare the load-balancing strategies when subject to different execution conditions.

5.3 Reacting to client behavior

The aim of the first test is to evaluate the performance of different algorithms when new clients arrive and existing clients disconnect.

In the experiment the infrastructure is composed of four replicas. The first replica is the fastest one and has $\phi_1 = 0.05$s (average time to execute both the mandatory and the optional components) and $\psi_1 = 0.005$s (average time to compute only the mandatory part of the response). The second replica is slower, with $\phi_2 = 0.25$s and $\psi_2 = 0.025$s. The third and fourth replicas are the slowest ones, having $\phi_{3,4} = 0.5$s and $\psi_{3,4} = 0.05$s.

Clients adhere to the closed-loop model. 50 clients are accessing the system at time 0s, and 10 of them are removed after 200s. At time 400s, 25 more clients query the application and 25 more arrives again at 600s. 40 clients disconnect at time 800s and the simulation is ended at time 1000s.

The right column in Figure 2 shows the control variable θ_i for each replica, while the left column shows the effective weights w_i, i.e., the weights that have been assigned by the load-balancing strategies computed a posteriori. Since solutions like

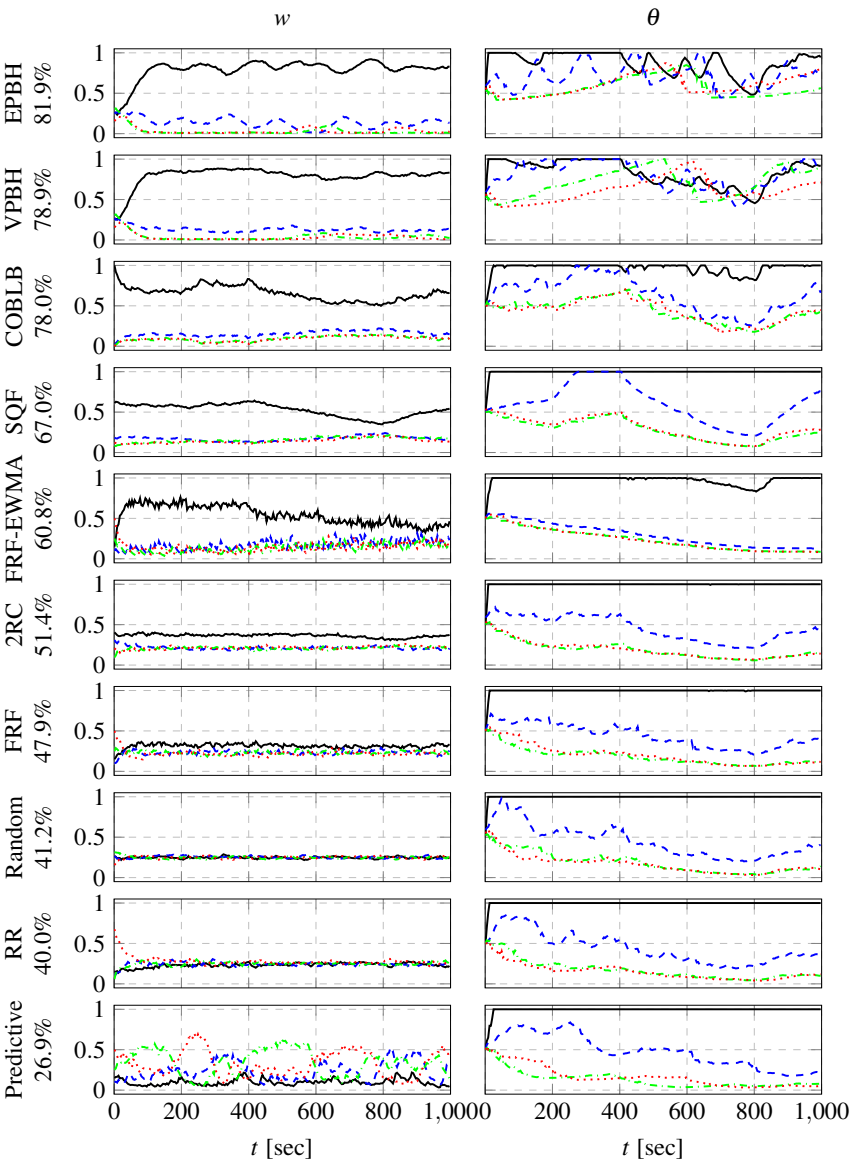

Figure 2. Results of a simulation with four replicas and clients entering and leaving the system at different time instants. The left column shows the effective weights while the right column shows the control variables for each replica. The first replica is shown in black solid lines, the second in blue dashed lines, the third in green dash-dotted lines, and the fourth in red dotted lines.

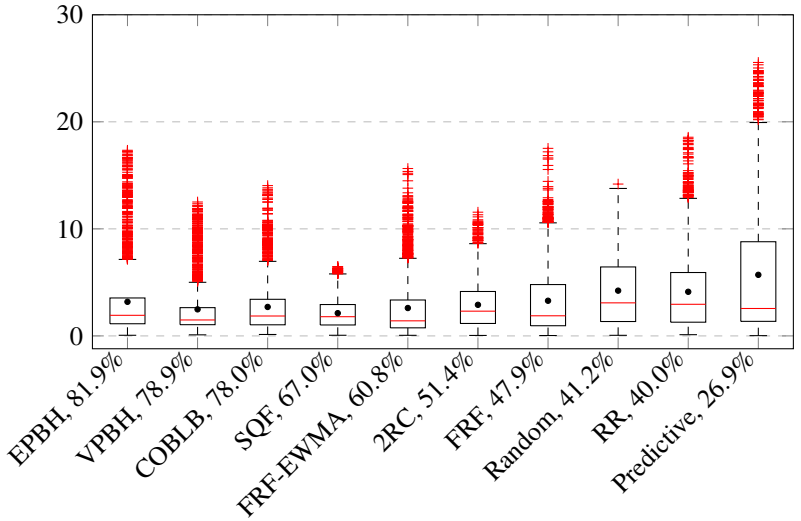

Figure 3. Box plots of the maximum response time in all the replicas for every control interval. Each box shows from the first quartile to the third. The red line shows the median; outliers are represented with red crosses while the black dots indicate the average value (also considering the outliers).

RR do not assign directly the weights, we decided to compute the effective values that can be found after the load-balancing assignments.

The algorithms are ordered by decreasing percentage $\%_{oc}$ of optional content served, where EPBH achieves the best percentage overall, followed by VPBH and by COBLB.

For this scenario, the strategies that are brownout-aware achieve better results in terms of percentage of optional content served. The SQF algorithm is the only existing one capable of achieving similar (yet lower) performance in terms of optional content delivered. The scenario also illustrates the benefit of using a brownout-aware strategy, as there is a constant underutilization of replica 1 for SQF.

To analyze the effect of the load-balancing strategies on the replicas response times, Figure 3 shows box plots of the maximum response time experienced by the replicas. The load-balancing strategies are ordered from left to right based on the percentage of optional code $\%_{oc}$ achieved. The bottom line of each box represents the first quartile, the top line the third and the red line is the median. The red crosses show the outliers. In addition to the classical box plot information, the black dots show for each algorithm the average value of the maximum response time measured during the experiment, also considering the outliers.

The box plots clearly show that all the solutions presented in this paper achieve distributions that have outliers, as well as almost all the literature ones. The only

Table 1. Performance with variable infrastructure resources

Algorithm	$\%_{oc}$	μ_u	σ_u
COBLB	**90.9**%	0.78	0.97
EPBH	89.5%	1.06	1.95
VPBH	87.7%	1.02	1.90
SQF	83.3%	**0.55**	**0.40**
RR	75.5%	1.11	2.42
Random	72.9%	0.86	2.23
2RC	72.2%	0.74	1.64
FRF	70.4%	1.27	2.03
FRF-EWMA	51.4%	1.44	3.41
Predictive	47.4%	1.66	3.48

exception seems to be SQF, that achieves very few outliers, predictable maximum response time, with a median that is just slightly higher than the one achieved by VPBH. EPBH offers the highest percentage of optional content served, by sacrificing the response time bound. From this additional information one can conclude that the solutions presented in this paper should be tuned carefully if response time requirements are hard. For example, for certain tasks, users prefer a very responsive applications instead of many features, hence the revenue of the application owner may be increased through lower response times. Notice that the proposed heuristics (EPBH and VPBH) have tunable parameters that can be used to exploit the trade-off between response time bounds and optional content.

This case study features only a limited number of replicas. However, we have conducted additional tests, also in more complex scenarios, featuring up to 20 replicas, reporting results similar to the ones presented herein. In the next section we test the effect of infrastructural changes to load-balancing solutions and response times.

5.4 Reacting to infrastructure resources

In the second case study the architecture is composed of five replicas. At time 0s, the first replica has $\phi_1 = 0.07$s, $\psi_1 = 0.001$s. The second and third replicas are medium fast, with $\phi_{2,3} = 0.14$s and $\psi_{2,3} = 0.002$s. The fourth and fifth replicas are the slowest with $\phi_{4,5} = 0.7$s and $\psi_{4,5} = 0.01$s.

At time 250s the amount of resources assigned to the first replica is decreased, therefore $\phi_1 = 0.35$s and $\psi_1 = 0.005$s. At time 500s, the fifth replica receives more resources, achieving $\phi_5 = 0.07$s and $\psi_5 = 0.001$s. The same happens at time 750 to the fourth replica.

Table 1 reports the percentage $\%_{oc}$, the average response time and the user-perceived stability for the different algorithms. It should be noted again that our strategies obtain better optional content served at the expense of slightly higher response times. However, COBLB is capable of obtaining both low response times and high percentage of optional content served. This is due to the amount of in-

formation that it uses, since we assume that the computation times for mandatory and optional part are known. The optimization-based strategy is capable of reacting fast to changes and achieves predictability in the application behavior. Again, if one does not have all the necessary information available, it is possible to implement strategies that would better exploit the trade-off between bounded response time and optional content.

6. Conclusion

We have revisited the problem of load-balancing different replicas in the presence of self-adaptivity inside the application. This is motivated by the need of cloud applications to withstand unexpected events like flash crowds, resource variations or hardware changes. To fully address these issues, load-balancing solutions need to be combined with self-adaptive applications, such as brownout. However, simply combining them without special support leads to poor performance.

Three load-balancing strategies are described, specifically designed to support brownout-compliant cloud applications. The experimental results clearly show that incorporating the application adaptation in the design of load balancing strategies pay off in terms of predictable behavior and maximized performance. They also demonstrated that the SQF algorithm is the best non-brownout-aware solution and therefore it should be used whenever it is not possible to adopt one of our proposed solution. The granularity of the actuation of the SQF load-balancing strategy is on a per-request based and the used information are much more updated with respect to the current infrastructure status, which is an advantage compared to weight-based solutions and helps SQF to serve requests faster. In future work we plan to investigate brownout-aware per-request solutions.

Finally, the application model used in this paper assumes a finite number of clicks per user, therefore the developed load-balancer strategies maximize the percentage of optional content served. However, when a different application model is taken into account, optimizing the absolute number of requests served with optional content is another possible goal, that should be investigated in future work.

References

Alomari, F. and D. Menascé (2013). “Efficient response time approximations for multiclass fork and join queues in open and closed queuing networks”. *IEEE Transactions on Parallel and Distributed Systems* 99, pp. 1–6.

Andreolini, M., S. Casolari, and M. Colajanni (2008). “Autonomic request management algorithms for geographically distributed internet-based systems”. In: *SASO*.

Ardagna, D., S. Casolari, M. Colajanni, and B. Panicucci (2012). "Dual time-scale distributed capacity allocation and load redirect algorithms for clouds". *J. Parallel Distrib. Comput.* **72**:6.

Bahi, J. M., S. Contassot-Vivier, and R. Couturier (2005). "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms". *IEEE Trans. Parallel Distrib. Syst.* **16**:4.

Barroso, L. A. and U. Hölzle (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool.

BIG-IP Local Traffic Manager. `http://www.f5.com/products/big-ip/big-ip-local-traffic-manager/`. Accessed: 2013-12-31.

Bodik, P., A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson (2010). "Characterizing, modeling, and generating workload spikes for stateful services". In: *SOCC*.

Boyd, S. and L. Vandenberghe (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA. ISBN: 0521833787.

Buyya, R., C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic (2009). "Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility". *Future Generation Computer Systems* **25**:6.

Cao, J., M. Andersson, C. Nyberg, and M. Kihl (2003). "Web server performance modeling using an M/G/1/K*PS queue". In: *10th International Conference on Telecommunications ICT 2003*. Vol. 2, pp. 1501–1506.

Cardellini, V., M. Colajanni, and P. S. Yu (2003). "Request redirection algorithms for distributed web systems". *IEEE Trans. Parallel Distrib. Syst.* **14**:4.

Casolari, S., M. Colajanni, and S. Tosi (2009). "Self-adaptive techniques for the load trend evaluation of internal system resources". In: *ICAS*.

Diao, Y., J. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano (2004). "Incorporating cost of control into the design of a load balancing controller". In: *RTAS*.

Diao, Y., C. W. Wu, J. Hellerstein, A. Storm, M. Surenda, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, L. Chu, and J. Colaco (2005). "Comparative studies of load balancing with control and optimization techniques". In: *ACC*.

Doyle, J., R. Shorten, and D. O'Mahony (2013). "Stratus: load balancing the cloud for carbon emissions control". *IEEE Transactions on Cloud Computing* **1**:1.

García, D. F. and J. García (2003). "TPC-W e-commerce benchmark evaluation". *Computer* **36**:2, pp. 42–48.

Gulati, A., G. Shanmuganathan, A. Holler, and I. Ahmad (2011). "Cloud-scale resource management: challenges and techniques". In: *HotCloud*.

Hamilton, J. (2007). "On designing and deploying internet-scale services". In: *LISA*.

Huang, C. and T. Abdelzaher (2005). "Bounded-latency content distribution feasibility and evaluation". *IEEE Transactions on Computers* **54**:11.

Kameda, H., E.-Z. Fathy, I. Ryu, and J. Li (2000). "A performance comparison of dynamic vs. static load balancing policies in a mainframe-personal computer network model". In: *CDC*.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014). "Brownout: building more robust cloud applications". In: *ICSE*.

Kleinrock, L. (1967). "Time–shared systems: a theoretical treatment". *Journal of the ACM* **14**:242-261.

Lin, M., Z. Liu, A. Wierman, and L. L. H. Andrew (2012). "Online algorithms for geographical load balancing". In: *IGCC*.

Lu, Y., Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg (2011). "Join-idle-queue: a novel load balancing algorithm for dynamically scalable web services". *Perform. Eval.* **68**:11.

Maggio, M., C. Klein, and K.-E. Årzén (2014). "Control strategies for predictable brownout in cloud computing". In: *IFAC WC*.

Manfredi, S., F. Oliviero, and S. Romano (2013). "A distributed control law for load balancing in content delivery networks". *IEEE/ACM Transactions on Networking* **21**:1.

Mars, J., L. Tang, R. Hundt, K. Skadron, and M. L. Soffa (2011). "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations". In: *MICRO*, pp. 248–259.

Mitzenmacher, M. (2001). "The power of two choices in randomized load balancing". *IEEE Trans. Parallel Distrib. Syst.* **12**:10.

Nakrani, S. and C. Tovey (2004). "On honey bees and dynamic server allocation in internet hosting centers". *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems* **12**:3-4.

Ni, L. and K. Hwang (1985). "Optimal load balancing in a multiple processor system with many job classes". *IEEE Transactions on Software Engineering* **11**:5.

Pao, T.-L. and J.-B. Chen (2006). "The scalability of heterogeneous dispatcher-based web server load balancing architecture". In: *PDCAT*.

Patterson, R. H., G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka (1995). "Informed prefetching and caching". In: *SOSP*.

Ranjan, S., R. Karrer, and E. Knightly (2004). "Wide area redirection of dynamic content by internet data centers". In: *INFOCOM*.

Sakata, M., S. Noguchi, and J. Oizumi (1971). "An analysis of the M/G/1 queue under round-robin scheduling". *Operations Research* **19**:2, pp. 371–385.

Salehie, M. and L. Tahvildari (2009). "Self-adaptive software: landscape and research challenges". *ACM Trans. Auton. Adapt. Syst.* **4**:2.

Schroeder, B., A. Wierman, and M. Harchol-Balter (2006). “Open versus closed: a cautionary tale”. In: *NSDI*.

Stankovic, J. A. (1985). “An application of bayesian decision theory to decentralized control of job scheduling”. *IEEE Trans. Comput.* **34**:2.

Tantawi, A. N. and D. Towsley (1985). “Optimal static load balancing in distributed computer systems”. *J. ACM* **32**:2.

Tomás, L. and J. Tordsson (2013). “Improving cloud infrastructure utilization through overbooking”. In: *CAC*, pp. 1–10.

Wang, L., V. Pai, and L. Peterson (2002). “The effectiveness of request redirection on CDN robustness”. In: *OSDI*.

Wolf, J. L. and P. S. Yu (2001). “On balancing the load in a clustered web farm”. *ACM Trans. Internet Technol.* **1**:2.

Zhang, L., Z. Zhao, Y. Shu, L. Wang, and O. W. W. Yang (2002). “Load balancing of multipath source routing in ad hoc networks”. In: *ICC*.

Paper V

Improving Cloud Service Resilience using Brownout-Aware Load-Balancing

Cristian Klein[1] **Alessandro Vittorio Papadopoulos**[2]
Manfred Dellkrantz[2] **Jonas Dürango**[2]
Martina Maggio[2] **Karl-Erik Årzén**[2]
Francisco Hernández-Rodriguez[1] **Erik Elmroth**[1]

Abstract

We focus on improving resilience of cloud services (e.g., e-commerce website), when correlated or cascading failures lead to computing capacity shortage. We study how to extend the classical cloud service architecture composed of a load-balancer and replicas with a recently proposed self-adaptive paradigm called brownout. Such services are able to reduce their capacity requirements by degrading user experience (e.g., disabling recommendations).

Combining resilience with the brownout paradigm is to date an open practical problem. The issue is to ensure that replica self-adaptivity would not confuse the load-balancing algorithm, overloading replicas that are already struggling with capacity shortage. For example, load-balancing strategies based on response times are not able to decide which replicas should be selected, since the response times are already controlled by the brownout paradigm.

In this paper we propose two novel brownout-aware load-balancing algorithms. To test their practical applicability, we extended the popular lighttpd web server and load-balancer, thus obtaining a production-ready implementation. Experimental evaluation shows that the approach enables cloud services to remain responsive despite cascading failures. Moreover, when compared to Shortest Queue First (SQF), believed to be near-optimal in the non-adaptive case, our algorithms improve user experience by 5%, with high statistical significance, while preserving response time predictability.

1. Introduction

Due to their ever-increasing scale and complexity, hardware failures in cloud computing infrastructures are the norm rather than the exception [Barroso and Hölzle, 2009; Guan and Fu, 2013]. This is why Internet-scale interactive applications – also called *services* – such as e-commerce websites, include replication early in their design [Hamilton, 2007]. This makes the service not only more scalable, i.e., more users can be served by adding more replicas, but also more resilient to failures: In case a replica fails, other replicas can take over. In a replicated setup, a single or replicated load-balancer is responsible for monitoring replicas' health and directing requests as appropriate. Indeed, this practice is well established and can successfully deal with failures as long as computing capacity is sufficient [Hamilton, 2007].

However, failures in cloud infrastructures are often correlated in time and space [Gallet et al., 2010; Yigitbasi et al., 2010]. Therefore, it may be economically inefficient for the service provider to provision enough spare capacity for dealing with all failures in a satisfactory manner. This means that, in case correlated failures occur, the service may *saturate*, i.e., it can no longer serve users in a timely manner. This in turn leads to dissatisfied users, that may abandon the service, thus incurring long-term revenue loss to the service provider. Note that the saturated service causes infrastructure overload, which by itself may trigger additional failures [Chuah et al., 2013], thus aggravating the initial situation. Hence, a mechanism is required to deal with rare, cascading failures, that feature temporary capacity shortage.

A promising self-adaptation technique that would allow dealing with this issue is *brownout* [Klein et al., 2014]. In essence, a service is extended to serve requests in two modes: with mandatory content only, such as product description in an e-commerce website, and with both mandatory and optional content, such as recommendations of similar products. Serving more requests with optional content, increases the revenue of the provider [Fleder et al., 2010], but also the capacity requirements of the service. A carefully designed controller decides the ratio of requests to serve with optional content, so as to keep the response time below the user's tolerable waiting time [Nah, 2004]. From the data-center's point-of-view, the service modulates its capacity requirements to match available capacity.

Brownout has been successfully applied to services featuring a single replica. Extending it to multiple replicas needs to be done carefully: The self-adaptation of each replica may confuse commonly used load-balancing algorithms (Section 2).

In this paper we enhance the resilience of replicated services through brownout. In other words, the service performs better at hiding failures from the user, as measured in the number of timeouts a user would observe. As a first step, a commonly-used load-balancing algorithm, Shortest Queue First (SQF), proved adequate for most scenarios. However, we found a few corner cases where the performance of the load-balancer could be improved using two novel, queue-length-based, brownout-aware algorithms that are fully event-driven.

Our contribution is three-fold:

1. We present two novel load-balancing algorithms, specifically designed for brownout services (Section 3.1).
2. We provide a production-ready brownout-aware load-balancer (Section 3.2).
3. We compare fault-tolerance without and with brownout, and existing load-balancing algorithms to our novel ones (Section 4).

Results show that the resulting service can tolerate more replica failures and that the novel load-balancing algorithms improve the number of requests served with optional content, and thus the revenue of the provider by up to 5%, with high statistical significance. Note that SQF is thought to be near-optimal, in the sense that it minimizes average response time for non-adaptive services [Gupta et al., 2007].

To make our results reproducible and foster further research on improved resilience through brownout, we make all source code available online[1].

2. Background and Motivation

In this section we provide the relevant background and define the challenge to address with respect to previous contributions.

2.1 Single Replica Brownout Services

To provide predictable performance in cloud services, the **brownout** paradigm [Klein et al., 2014] relies on a few, minimally intrusive code changes (e.g., 8 lines of code) and an online adaptation strategy that controls the response time of a single-replica based service. The service programmer builds a brownout-compliant cloud service breaking the service code into two distinct subsets: Some functions are marked as *mandatory*, while others as *optional*. For example, in an e-commerce website, retrieving the characteristics of a product from the database can be seen as mandatory – a user would not consider the response useful without this information – while obtaining comments and recommendations of similar products can be seen as optional – this information enhances the quality of experience of the user, but the response is useful without them.

For a brownout-compliant service, whenever a request is received, the mandatory part of the response is always computed, whereas the optional part of the response is produced only with a certain probability given by a control variable, called the **dimmer** value. Not executing the optional code reduces the computing capacity requirements of the service, but also degrades user experience. Clearly, the user would have a better experience seeing optional content, such as related products and comments from other users. However, in case of overload and transient failure

[1] `https://github.com/cloud-control/brownout-lb-lighttpd`

conditions, it is better to obtain partial information than to have increased response times or no response, due to insufficient capacity.

Keeping the service responsive is done by adjusting the probability of executing the optional components [Klein et al., 2014]. Specifically, a controller monitors response times and adjusts the dimmer value to keep the 95th percentile response time observed by the users around a certain setpoint. Focusing on 95th percentile instead of average, allows more users to receive a timely response, hence improve their satisfaction [DeCandia et al., 2007]. A setpoint of 1 second can be used, to leave a safety margin to the user's tolerable waiting time, estimated to be around 4 seconds [Nah, 2004]. While the initial purpose of the brownout control was to enhance the service's tolerance to a sudden increase in popularity, it also significantly improves responsiveness during infrastructure overload phases, when the service is not allocated enough capacity to manage the amount of incoming requests without degrading the user experience. However, the brownout approach was used only in services composed of a single replica, thus the service could not tolerate hardware failures.

Let us briefly describe the design of the controller. Denoting the dimmer value with Θ and using a simple and useful model, we assume that the 95th percentile response time of the service, measured at regular time intervals, follows the equation

$$t(k+1) = \alpha(k) \cdot \Theta(k) + \delta t(k), \tag{1}$$

i.e., the 95th percentile response time $t(k+1)$ of all the requests that are served between time index k and time index $k+1$ depends on a time varying unknown parameter $\alpha(k)$ and can have some disturbance $\delta t(k)$ that is a priori unmeasurable. $\alpha(k)$ takes into account how the dimmer Θaffects the response time, while $\delta t(k)$ is an additive correction term that models variations that do not depend on the dimmer choice — for example, variation in retrieval time of data due to cache hit or miss. Notice that the used model ignores the time needed to compute the mandatory part of the response, but it captures the service behavior enough for the control action to be useful. The controller design aims for canceling the disturbance $\delta t(k)$ and selecting the value of $\Theta(k)$ so that the 95th percentile response time would be equal to the setpoint value.

With a control-theoretical analysis [Klein et al., 2014], it is possible to select the dimmer value to provide some guarantees on the service behavior. The selection is based on the adaptive proportional and integral controller

$$\Theta(k+1) = \Theta(k) + \frac{1-p_1}{\tilde{\alpha}(k)} \cdot e(k), \tag{2}$$

where the value $\tilde{\alpha}(k)$ is an estimate of the unknown parameter $\alpha(k)$ computed with a Recursive Least Square (RLS) filter. The error $e(k)$ is the difference measured at time index k between the setpoint for the response time and its measured value, p_1 is a parameter of the controller, that allows to trade reactivity for robustness. A formal

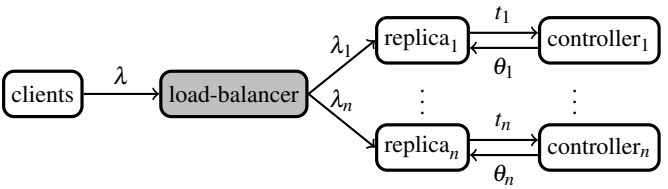

Figure 1. Architecture of a brownout cloud service featuring multiple replicas.

analysis of the guarantees provided by the controller and the effect of the value of p_1 can be found in [Klein et al., 2014].

Besides computing a new dimmer value, the model parameter α is re-estimated as $\tilde{\alpha}(k)$, which is computed using the last estimation $\tilde{\alpha}(k-1)$, the measured response time $t(k)$ and the current dimmer $\theta(k)$, as illustrated in the following RLS filter equations

$$\begin{aligned}
\varepsilon(k) &= t(k) - \Theta(k)\tilde{\alpha}(k-1) \\
g(k) &= P(k-1)\Theta(k)\left[f + \Theta(k)^2 P(k-1)\right]^{-1} \\
P(k) &= f^{-1}\left[P(k-1) - g(k)\Theta(k)P(k-1)\right] \\
\tilde{\alpha}(k) &= \alpha(k-1) + \varepsilon(k)g(k),
\end{aligned} \tag{3}$$

where ε is the so called "prediction error", g is a gain factor, f is a "forgetting factor" and P is the covariance matrix of the prediction error.

Through empirical testing on two popular cloud applications, RUBiS [*Rice University Bidding System* 2014] and RUBBoS, we found the following values to give a good trade-off between reactivity and stability: $p_1 = 0.9$ and $f = 0.95$. In the end, making a single-replica cloud service brownout-compliant improves its robustness to sudden increases in popularity and infrastructure overload.

2.2 Multiple Replica Brownout-Compliant Services

For fault tolerance, cloud services should feature multiple replicas. Figure 1 illustrates the software architecture that is deployed to execute a brownout-compliant service composed of multiple replicas. Besides the addition of replica controllers to make it brownout-compliant, the architecture is widely accepted as the reference one for replicated cloud services [Barroso and Hölzle, 2009].

In the given cloud service architecture, access can only happen through the load-balancer. The client requests are assumed to arrive at an unknown but measurable rate λ. Each client request is received by the load-balancer, that forwards it to one of the n replicas. Each replica independently decides if the request should be served with or without the optional part. The chosen replica produces the response and sends it back to the load-balancer, which forwards it to the original client. Since all responses of the replicas go through the load-balancer, it is possible to piggy-back the current value of the dimmer Θ_i of each replica i through the response, so that this value can be *observed* by the load-balancer.

For better decoupling and redundancy, the load-balancer does not have any knowledge on *how* each replica controller adjusts Θ_i. Hence, the load-balancer only stores soft state, reducing impact in case of failover to a backup load-balancer. Also, operators can deploy our solution incrementally, first adding brownout to replicas, then upgrading the load-balancer.

In the end, each replica i receives a fraction λ_i of the incoming traffic and serves requests with a 95th percentile response time around the same setpoint of 1 second. Each replica i chooses a dimmer Θ_i that depends on the amount of traffic it receives and the computing capacity available to it. Noteworthy is the fact that by directing too many requests to a certain replica the load-balancer may indirectly decrease the amount of optional requests served by that replica.

Preliminary simulation results [Dürango et al., 2014] compared different load-balancing algorithms for this architecture, such as round-robin, fastest replica first, random and two random choices. The main result of this comparison is that load-balancing algorithms that are based on measurements of the response times of the single replicas are not suited to be used with brownout-compliant services, since the replica controllers already keep the response times close to the setpoint. The *only* existing algorithm that proved to work adequately with brownout-compliant services is Shortest Queue First (SQF) [Gupta et al., 2007; Dürango et al., 2014]. It works by tracking the number of queued requests q_i on each replica and directing the next request to the replica with the lowest q_i.

However, SQF proved to be inadequate for maximizing the optional content served, such as recommendations, hence producing lower revenues for the service provider [Fleder et al., 2010]. Brownout-aware load-balancers do better in maximizing the optional component served. However, to date, only **weight-based** algorithms were considered, where each replica gets a fraction of the incoming traffic proportional to a dynamic weight. A controller periodically adjusts the weights based on the dimmer values of each replica [Dürango et al., 2014]. Results suggested that deciding periodically gives good results in steady-state, however, the resulting service is not reactive enough to sudden capacity changes, as would be the case when a replica fails.

2.3 Problem Statement

The main objective is to improve resilience of cloud services. On one hand, the service should serve requests with a 95th percentile response time as close as possible to the setpoint. On the other hand, the service should maximize the optional content served.

In this paper we propose novel brownout-aware load-balancers that are event-based, for better reactivity. We limit the comparison to SQF, since it was shown to be the only reasonable choice to maximize optional content in brownout-compliant services.

3. Design and Implementation

This section describes the core of our contribution, two load-balancing algorithms and a production-ready implementation.

3.1 Brownout-Compliant Load-Balancing Algorithms

Here we discuss two brownout-compliant control-based load-balancing algorithms. Those are based on some ideas presented in [Dürango et al., 2014], but with two major modifications. First, all the techniques proposed in [Dürango et al., 2014] are trying to maximize the optional content served by acting on the fraction of incoming traffic sent to a specific replica, while here the algorithms are acting in an SQF-like way but with **queue-offsets** that are dynamically changed in time. The queue-offsets u_i take into account the measured performance of each replica i in terms of dimmers, and are subtracted from the actual value of the queue length q_i so as to send the request to the replica with the lowest $q_i - u_i$.

The second and most important modification is that in [Dürango et al., 2014] all the algorithms run periodically, independently of the incoming traffic, while in this paper we are considering algorithms that are **fully event-driven**, updating the queue-offsets and taking a decision for each request. Therefore all gains in the two following algorithms need to be scaled by the time elapsed since the last queue-offsets update.

These two modifications highly improve the achieved performance, both in terms of optional content served and response time, rendering the service more reactive to sudden capacity changes, as is the case with failures. Let us now present two algorithms for computing the queue-offsets u_i.

PI-Based Heuristic (PIBH) Our first algorithm is based on a variant of the PI (Proportional and Integral) controller on incremental form, which is typical in digital control theory [Landau et al., 2006]. In principle, the PI control action in incremental form is based both on the variation of the dimmers value (which is related to the proportional part), and their actual values (which is related to the integral part).

As presented above, the values of the queue offsets u_i are updated every time a new request is received by the service, according to the last values of the dimmers Θ_i, piggy-backed by each replica i through a previous response, and on the queue lengths q_i, using the formula

$$u_i(k+1) = (1-\gamma)\left[u_i(k) + \gamma_P \Delta\Theta_i(k) + \gamma_I \Theta_i(k)\right] + \gamma q_i(k), \tag{4}$$

where $\gamma \in (0,1)$ is a filtering constant, γ_P and γ_I are constant gains related to the proportional and integral action of the classical PI controller.

We selected $\gamma = 0.01$ and $\gamma_P = 0.5$ based on empirical testing. Once γ and γ_P are fixed to a selected value, increasing the integral gain γ_I calls for a stronger action on the load-balancing side, which means that the load-balancer would take decisions

very much influenced by the current values of Θ_i, therefore greatly improving performance at the cost of a more aggressive control action. On the contrary, decreasing γ_I would smooths the control action, possibly resulting in performance loss due to a slower reaction time. The choice of the integral gain allows to exploit the trade-off between performance and robustness. For the experiments we chose $\gamma_I = 5.0$.

Equality Principle-Based Heuristic (EPBH) The second algorithm is based on the heuristic that the system will perform well in a situation when all replicas have the same dimmer value. By comparing Θ_i for each replica i with the mean dimmer of all replicas, a carefully designed update rule can deduce which replica should receive more load, in order to drive all dimmer to equality. The queue offsets can thus be updated as

$$u_i(k+1) = u_i(k) + \gamma_e \left(\Theta_i(k) - \frac{1}{n} \sum_{j=1}^{n} \Theta_j(k) \right), \tag{5}$$

where γ_e is a constant gain. The gain decides how fast the controller should act. Based on empirical tuning we chose $\gamma_e = 0.1$.

Since the implementation only updates the dimmer measurements in the load balancer when responses are sent, EPBH risks ending up in a situation where a replica gets completely starved. To remedy this, the algorithm first chooses a random empty replica ($q_i = 0$) if there are any, otherwise chooses the replica with the lowest $q_i - u_i$, as described above.

3.2 Implementation

In order to show the practical applicability of the two algorithms and evaluate their performance, we decided to implement them in an existing load-balancing software. We chose `lighttpd`[2], a popular open-source web server and load-balancing software, that features good scalability, thanks to an event-driven design. `lighttpd` already included all necessary prerequisites, such as HTTP request forwarding, HTTP response header parsing, replica failure detection and the state-of-the-art queue-length-based SQF algorithm. HTTP response header parsing allowed us to easily implement dimmer piggy-backing through the custom `X-Dimmer` HTTP response header, with a small overhead of only 20 bytes. In the end, we obtained a production-ready brownout-aware load-balancer implementation featuring the two algorithms, with less than 180 source lines of C code[3].

4. Empirical Evaluation

In this section we show through real experiments the benefits in terms of resilience that can be obtained through our contribution. First, we describe our experimental

[2] `http://www.lighttpd.net/`

[3] `https://github.com/cloud-control/brownout-lb-lighttpd`

setup. Next, we show the benefits that brownout can add to a replicated cloud service which uses the state-of-the-art load-balancing algorithm, SQF. Finally, we show the improvements that can be made using our brownout-specific load-balancing algorithms.

4.1 Experimental Setup

Experiments were conducted on a single physical machine equipped with two AMD Opteron™ 6272 processors[4] and 56 GB of memory. To simulate a typical cloud environment and allow us to easily fail and restart replicas, we use the Xen hypervisor [Barham et al., 2003]. Each replica is deployed with all its tiers – web server and database server – inside its own VM, as is commonly done in practice [Sripanidkulchai et al., 2010], e.g., using a LAMP stack [*Tutorial: Installing a LAMP Web Server* 2013]. Each VM was configured with a static amount of memory, 6 GB, enough to hold all processes and the database in-memory, and a number of virtual cores depending on the experiment.

Inside each replica we deployed an identical copy of RUBiS [*Rice University Bidding System* 2014], an eBay-like e-commerce prototype, that is widely-used for cloud benchmarking [Gong et al., 2010; Shen et al., 2011; Zheng et al., 2009; Stewart and Shen, 2005; Vasić et al., 2012; Stewart et al., 2007; Chen et al., 2007]. RUBiS was already brownout-compliant, thanks to a previous contribution [Klein et al., 2014] and adding piggy-backing of the dimmer value was trivial[5]. The replica controllers are configured the same, with a target 95th percentile response time of 1 second. To avoid having to deal with synchronization or consistency issues, we only used a read-only workload. However, adding consistency to replicated services is well-understood [Diegues and Romano, 2013; Cooper et al., 2010; Ardekani et al., 2013] and, in case of RUBiS, would only require an engineering effort. The load-balancer, i.e., `lighttpd` extended with our brownout-aware algorithms, was deployed inside the privileged VM in Xen, i.e., Dom0, pinned to a dedicated core.

To generate the workload, we had to choose between three system models: open, closed or partly-open [Schroeder et al., 2006]. In an open system model, typically modeled as Poisson process, requests are issued with an exponentially-random inter-arrival time, characterized by a rate parameter, without waiting for requests to actually complete. In contrast, in a closed system model, a number of users access the service, each executing the following loop: issue a request, wait for the request to complete, "think" for a random time interval, repeat. The resulting average request inter-arrival time is the sum of the average think-time and the average response time of the service, hence dependent on the performance of the evaluated service. A partly-open system model is a mixture between the two: Users arrive according to a Poisson process and leave after some time, but behave closed

[4] 2100 MHz, 16 cores per processor, no hyper-threading.

[5] `https://github.com/cloud-control/brownout-lb-rubis`

while in the system. As with the closed model, the inter-arrival time depends on the performance of the evaluated system.

We chose to use an open system model workload generator. Since its behavior does not depend on the performance of the service, this allows us to eliminate a factor potentially contributing to noise when comparing our contribution to competing approaches. We extended this model to include timeouts, as required to emulated users' tolerable waiting time of 4 seconds [Nah, 2004].

Given our chosen model and the need to measure brownout-specific behavior, the workload generator provided with RUBiS was insufficient for three reasons. First, RUBiS's workload generator uses a closed system model, without timeouts. Second, it only reports statistics for the whole experiment and does not export the time series data, preventing us from observing the service's behavior during transient phases. Finally, the tool cannot measure the number of requests served with optional content, which represents the quality of the user-experience and the revenue of the service provider. Therefore, we extended our own workload generator, `httpmon`[6], as required.

We made sure that the results are reliable and unbiased as follows:

- replicas were warmed up before each experiment, i.e., all virtual disk content was cached in the VM's kernel;
- replicas were isolated performance-wise by pinning each virtual core to its own physical core;
- experiments were terminated after the workload generator issued the same number of requests;
- `httpmon` and the `lighttpd` were each executed on a dedicated core;
- no non-essential processes nor `cron` scripts were running at the time of the experiments.

To qualify the resilience of the service, we chose two metrics that measure how well the service is performing in hiding failures, or, otherwise put, how strongly the user is affected by failures. The **timeout rate** represents the number of requests per second that were not served by the service within 4 seconds, due to overload. In production, a request that timed out will make a user unhappy. She may leave the service to join other competitors, thus incurring long-term losses to the service provider. The **optional content ratio** represents the percentage of requests served with optional content. Serving a request with optional content, such as recommendations of similar products, may increase the service provider's revenue by 50% [Fleder et al., 2010]. Therefore, a request served without optional content also represents a revenue loss to the provider, albeit, a smaller one than the long-term

[6] `https://github.com/cloud-control/httpmon`

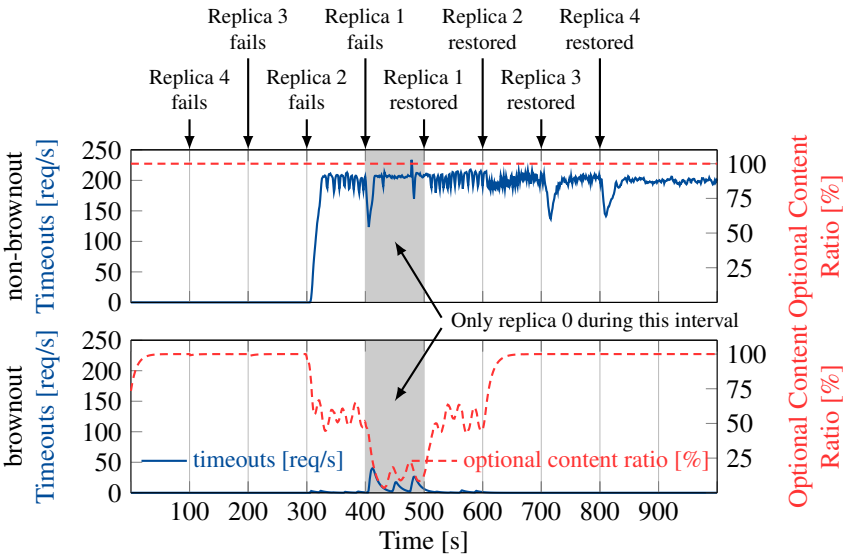

Figure 2. Experimental results comparing resilience without and with brownout. Configuration: 5 replicas, each having 4 cores.

loss incurred by a timeout. Ideally, the service should strive to maximize the optional content ratio, without causing timeouts. Finally, to give insight into the system's behavior, we also report the **response time**, i.e., the time it took to serve a request from the user's perspective, including the time required to traverse the load-balancer.

4.2 Resilience without and with Brownout

In this section, we show through experiments how brownout can increase resilience, even if used with a brownout-unaware load-balancing algorithm, such as SQF. To this end, we expose both a non-brownout and a brownout service to cascading failures and their recovery. The experiment starts with 5 replicas, each being allocated 4 cores, i.e., the service is allocated a total computing capacity of 20 cores. Every 100 seconds a replica crashes until only a single one is active. Then, every 100 seconds a replica is restored. Crashing and restoring replicas are done by respectively killing and restarting both the web server and the database server of the replica.

We plot the timeout ratio and the optional content ratio. Note that, for the service without brownout, the ratio of optional content is fixed at 100%, whereas the service featuring brownout this quantity is adapted based on the available capacity, i.e., the number of available replicas. To focus on the behavior of the service due to failure, we kept the request-rate constant at 200 requests per second. Note that, the replicas

were configured with enough soft resources (file descriptors, sockets, etc.) to deal with 2500 simultaneous requests. We ran several experiments in different conditions and always obtained similar results. Therefore, to better highlight the behavior of the service as a function of time, we present the results of a single experiment instance as time series.

Figure 2 show the results. One can observe that the non-brownout service performs well even with 2 failed replicas, from time 0 to 300. Indeed, there are no timeouts and all requests are served with optional content. `lighttpd` already includes code to retry a failing requests on a different replica, hence hiding the failure from the user. During this time interval, the brownout service performs almost identically, except negligible reductions in optional content ratio at start-up and when a replica fails, until the replica controller adapts to the new conditions.

However, starting with time 300, when the third replica fails, the non-brownout service behaves poorly. Computing capacity is insufficient to serve the incoming requests fast enough and response time starts increasing. A few seconds later the service is saturated and almost all incoming requests time out. The small oscillations and spikes on the timeout per second plot are due to the randomness of the request inter-arrival time in the open client model.

Even worse, when enough replicas are restored to make capacity sufficient, the non-brownout service still does not recover. This finding may seem counter-intuitive, but repeating the experiments also in different conditions (number of allocated cores, different workloads, etc.) gave similar results. In our experiments, as common practice in production environments, user timeouts are not propagating to the service, i.e., they do not cancel pending web requests or database transactions. Thus, the database server is essentially filled with transactions that will time out, or that may have already timed out on the user-side. Hence, all computing capacity is wasted on "rotten" requests, instead of striving to serve new requests. The database server continues to waste computing capacity on "rotten" requests, even after enough replicas are restored. The non-brownout service does recover eventually, but this takes significant time, at least 10 minutes in our experiments. Of course, in production environments the service operator or a self-healing mechanism would likely disable the service, kill all pending transactions on the database servers and re-enable the service. Nevertheless, this behavior is still undesirable.

In contrast, the brownout service performs well even with few active replicas. At time 300, when the third replica fails leading the service into capacity insufficiency, the replica controllers detect the increase in response time and quickly reacts by reducing the optional content ratio to around 55%. As a results, the service does not saturate and users can continue enjoying a responsive service. At time 400 when the fourth replica fails, capacity available to the service is barely sufficient to serve any requests, even with zero optional content ratio. However, even in this case, the brownout service significantly reduces the number of timeouts by keeping the optional content ratio low, around 10%. Finally, when replicas are restored, the service recovers fairly quickly. Thanks to the action of the replica controllers, the database

Table 1. Summary of non-brownout vs. brownout results.

Scenario	Metric	Non-brownout	Brownout
4 cores	Requests served	31.2%	99.3%
200 requests/s	With optional content	31.2%	81.0%
2 cores	Requests served	31.6%	99.3%
100 requests/s	With optional content	31.6%	82.0%
heterogeneous	Requests served	68.8%	99.5%
166 requests/s	With optional content	68.8%	90.2%

servers do not fill up with "rotten" requests.

On the downside, the brownout service features some oscillations of optional content while dealing with capacity shortage. This is due to the fact that the replica controllers attempt to maximize the number of optional content served, risking short increases in response time. These increases in response time are detected by the controllers, which adapt by reducing the number of optional content served. This process repeats, thus causing the oscillations. Except when capacity is close to being insufficient even with optional content completely disabled, these oscillations are harmless. Nevertheless, we are currently investigating several research directions to mitigate them, so as to allow brownout services to function well even in extreme capacity shortage situations.

In addition to the 4-core scenario above, we devised two other experimental scenarios to confirm our findings, as summarized in Table 1. In the 2-core scenario, we configured each replica with 2 cores, while in the heterogeneous scenario the number of cores for each replica is 8, 8, 1, 1, 1, respectively. In both scenarios, we scaled down the request-rate to maintain the same request-rate per core as in the 4-core scenario. Noteworthy is that in the heterogeneous scenario, the non-brownout service recovered faster than in the 4-core and 2-core scenarios. This can be observed by comparing the difference between the percentage of requests served by the brownout service and the non-brownout service among the three scenarios. Nevertheless, the key findings still hold.

In summary, adding brownout to a replicated service improves its resilience, even when using a brownout-unaware load-balancing algorithm. The increase in resilience that can be obtained is specific to each service and depends on the ratio between the maximum throughput with optional content disabled and the one with optional content enabled. Hence, by measuring these two values a cloud service provider can either estimate the increase in resilience during capacity shortages given the current version of the service, or may decide to develop a new version of the service, with more content marked as optional, so as to reach the desired level of resilience.

4.3 SQF vs. Brownout-Aware Load-Balancers

In this section, we compare the two brownout-aware load-balancing algorithms proposed herein, i.e., PIBH and EPBH, to the best brownout-unaware one, SQF [Dürango et al., 2014]. We shall use the word *better* in the sense that we have statistical evidence that the average performance is significantly higher with a p-value smaller than 0.01, by performing a Welch two sample t-test [Welch, 1947] on the optional component served and on the response time. In other words, the probability that the difference is due to chance is less than 1%. Analogously, we use the word *similarly* to denote that the difference is not statistically significant.

For thorough comparison, we tested the three algorithms using a series of scenarios, each having a certain pattern of request rate over time and amount of cores allocated to each replica. Each scenario was executed several times, to collect enough results to draw statistically significant conclusions. We were **unable to find any scenario in which SQF would perform *better***, which supports the hypothesis that our algorithms are at least as good as SQF. In fact, in most scenarios, such as those featuring high request rate variability or many replicas failing at once, SQF performed *similarly* to our brownout-aware load-balancers (not shown for briefness). However, we observed that in scenarios featuring capacity heterogeneity, our algorithms performed *better* than SQF with respect to the optional content ratio.

As a matter of fact, in cloud computing environments, replicas may end up being allocated heterogeneous capacity, e.g., one replica is allocated 2 cores, while another replica is allocated 8 cores. This may happen due to several factors. For example, the cloud infrastructure provider may practice overbooking and the machine on which a replica is hosted becomes overloaded [Tomás and Tordsson, 2013]. As another example, previous elasticity (auto-scaling) decisions may have resulted in heterogeneously sized replicas [Sedaghat et al., 2013]. Hence, it is of uttermost importance that a load-balancing algorithm is able to deal efficiently with such cases. As illustrated below on two scenarios, **both PIBH and EPBH perform *better* than SQF**.

"2×1+3×8 cores" Scenario The first scenario consists of a constant request rate of 400 requests per second. The service consists of 5 replicas, two of which are allocated 1 core, while the other three are allocated 8 cores. This scenario leaves the service with insufficient capacity to serve all requests with optional content. Furthermore, the constant workload and capacity allows us to eliminate sources of noise and obtain statistically significant results with 30 experiments for each algorithm, a total of 90 experiments.

Figure 3 presents the results of the first scenario as scatter plots: The x-axis represents response time (average and 95th percentile respectively in the top and the bottom graph), while the y-axis represents optional content ratio, each experiment being associated with a point. The results of the paired t-test comparing the optional content ratio of the three algorithms are presented in Table 2. As can be observed, when compared to SQF, the novel brownout-aware algorithms PIBH and EPBH

Table 2. Improvement in amount of optional content served, after 120000 requests (summary of Figure 3, "2×1+3×8 cores" scenario).

Algorithms (# Optional Content)		Impr.	Statistical Conclusion
PIBH (105646)	SQF (100273)	5.34%	PIBH significantly **better** ($p < 10^{-15}$)
EPBH (104816)	SQF (100273)	4.52%	EPBH significantly **better** ($p < 10^{-15}$)

improve optional content ratio by 5.34% and 4.52%, respectively, with a high significance (low p-value). This is due to the fact that the brownout-aware algorithms are able to exploit the replicas with a higher optional content ratio, at the expense of somewhat higher response times. Slightly increasing the average response time (Figure 3 top) yet improving the optional content served to the end user is an acceptable tradeoff, also considering that we have control on the target 95th percentile of the response time (Figure 3 bottom).

Recall that the replica controllers are configured with a target response time of 1 second. Furthermore, improved optional content ratio does not interfere with the self-adaptation of the replicas. As can be seen in Figure 3, all three algorithms obtain a similar distribution of response times. In Table 3 the paired t-test is applied also to the 95th percentile of the response time. The results confirm that PIBH behaves in a similar way with respect to the SQF, but producing better performance in terms of optional content served. When comparing EPBH to SQF, the average 95th percentile is 42ms higher in the former with quite a low p-value. However, it is to be noticed that the setpoint for the 95th percentile is set to 1 second, which is way higher than all of the presented results. Thus, the higher 95th percentile response time is not a concern.

"3×1+2×8 cores" Scenario For the second scenario, we maintain the same request rate, but configure three replicas with 1 core and two replicas with 8 cores. This means that the service has even less capacity available than in the first scenario, thus being forced to further reduce the optional content ratio. Scatter plots of response time and optional content ratio are presented in Figure 4, analogously to the previous scenario, while pair-wise comparison of algorithms is presented in

Table 3. Improvement in amount of 95th percentile of the response time (summary of Figure 3, "2×1+3×8 cores" scenario).

Algorithms (95th perc. [ms])		Impr.	Statistical Conclusion
PIBH (637ms)	SQF (648ms)	-1.7%	PIBH and SQF **similar** ($p = 0.992$)
EPBH (690ms)	SQF (648ms)	6.4%	SQF significantly **better** ($p < 10^{-9}$)

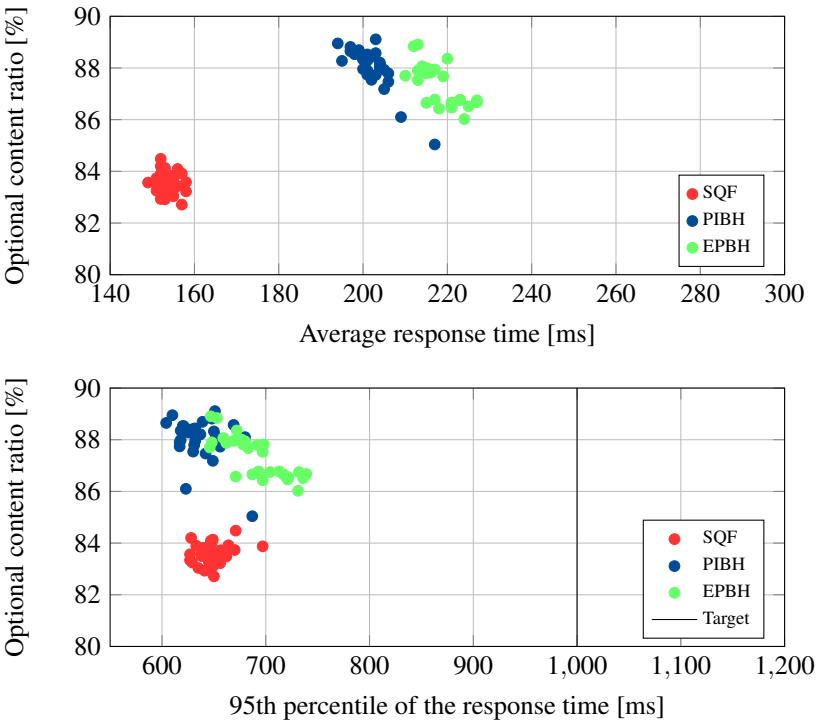

Figure 3. Comparison of SQF and brownout-aware load-balancing algorithms when two replicas have 1 core and three replicas have 8 cores.

Table 4. PIBH and EPBH outperform SQF with respect to optional content ratio by 5.17% and 3.13%, respectively.

Again, this is achieved without interfering with the self-adaptation of the replicas: 95th percentile response times are distributed similarly for all three algorithms close to the target. This is also proven by the paired t-test presented in Table 5, where both PIBH and EPBH appear to be comparable with SQF in terms of 95th percentile of the response time. In this case, since the capacity of the system is reduced, this quantity is increased, but on average still lower than the setpoint (set to 1 second). The same holds for the average response time, which is slightly increased with respect to the previous scenario.

4.4 Discussion

To sum up, our novel brownout-aware load-balancing algorithms perform at least as well as or **outperform** SQF by up to 5% in terms of optional content served, with a high statistical significance. This improvement translates into better quality

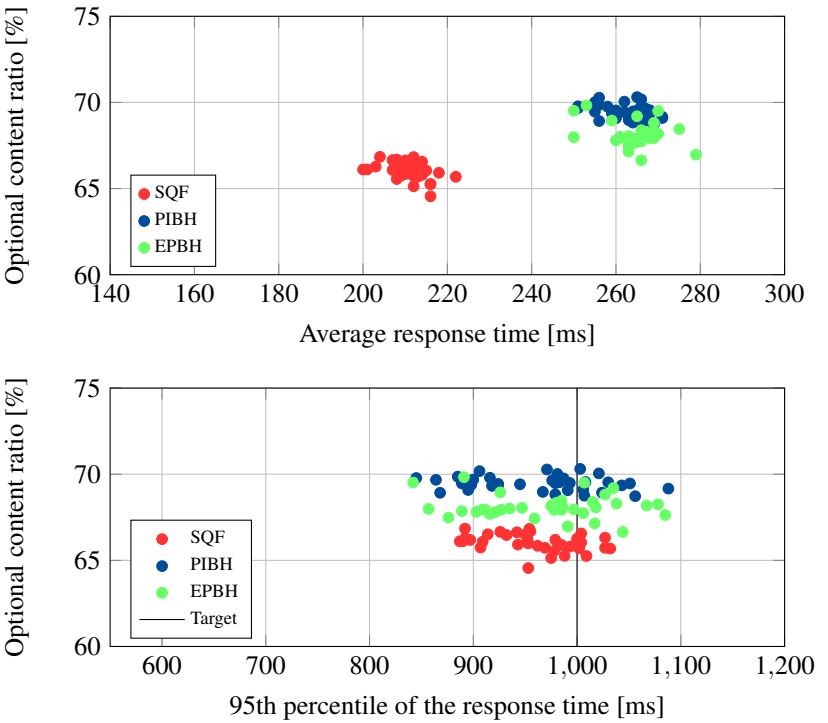

Figure 4. Comparison of SQF and brownout-aware load-balancing algorithms when three replicas have 1 core and two replicas have 8 cores.

Table 4. Improvement in amount of optional content served, after 120000 requests (summary of Figure 4, "3×1+2×8 cores" scenario).

Algorithms (# Optional Content)		Impr.	Statistical Conclusion
PIBH (83360)	SQF (79244)	5.17%	PIBH significantly **better** ($p < 10^{-15}$)
EPBH (81735)	SQF (79244)	3.13%	EPBH significantly **better** ($p < 10^{-15}$)

of experience for users and increased revenue for the service provider. Hence, our contribution helps cloud services to better hide failures leading to capacity shortages, in other words, services are more resilient.

Noteworthy is that the competitor, SQF has been found to be near-optimal with respect to response time for non-adaptive services [Gupta et al., 2007]. Thus, besides improving resilience of cloud services, our contribution may be of interest

Table 5. Improvement in amount of 95th percentile of the response time (summary of Figure 4, "3×1+2×8 cores" scenario).

Algorithms (95th perc. [ms])		Impr.	Statistical Conclusion
PIBH (963ms)	SQF (959ms)	0.4%	PIBH and SQF **similar** ($p = 0.3778$)
EPBH (969ms)	SQF (959ms)	1.0%	EPBH and SQF **similar** ($p = 0.2265$)

to other communities, to discover the limits of SQF, and sketch a possible way to design new dynamic load-balancing algorithms.

5. Related Work

The challenge of building reliable distributed systems consists in providing various safety and liveness guarantees while the system is subject to certain classes of failures. Our contribution closely relates to *multi-graceful degradation* [Lin and Kulkarni, 2013], in which the requirements that the service guarantees vary depending on the magnitude of the failure. However, due to the conflicting nature of requirements – maintaining maximum response time and maximizing optional content served, in the presence of noisy request servicing times – brownout does not provide formal guarantees. Instead, thanks to control-theoretical tools, the service is driven to a state to increase likelihood of meeting its requirements.

Brownout can be seen as a *model revision*, i.e., an existing service is extended to provide new guarantees. Specifically, we deal with crashes but also with limplocks [Do et al., 2013], the latter implying that a machine is working, but slower than expected.

In the context of self-stabilization, a new metric has been proposed to measure the recovery performance of an algorithm, the expected number of recovery steps [Fallahi et al., 2013]. An equivalent metric, the number of control decisions to recovery, could be used by a service operator for tuning the service to the expected capacity drop and the request servicing time of the replicas.

Our contribution is designed to deal with failures reactively. Failure prediction [Guan and Fu, 2013], if accurate enough, could be used as a feed-forward signal to improve reactivity and reduce the number of timeouts after a sudden drop in computing capacity.

Since the service's data has to be replicated an important issue is ensuring consistency. Various algorithms have been proposed, each offering a different trade-off between performance and guarantees [Diegues and Romano, 2013; Cooper et al., 2010; Ardekani et al., 2013]. Our contribution is orthogonal to consistency issues, hence our methodology can readily be applied no matter what consistency the service requires. However, a future extension of brownout could consist in avoiding

service saturation by reducing consistency.

In replicated cloud services, load-balancers have a crucial role for ensuring resilience but also maintain performance [Barroso and Hölzle, 2009; Hamilton, 2007]. Load-balancing algorithm can either be global (inter-data-center) or local (intra-data-center or cluster-level). Global load-balancing decides what data-center to direct a user to, depending on geographic proximity [Lin et al., 2012] or price of energy [Doyle et al., 2013]. Once a data-center has been selected a local algorithm directs the request to a machine in the data-center. Our contribution is of the local type.

Various local load-balancing algorithms have been proposed. For non-adapting replicas, Shortest Queue First (SQF) has shown to be very close to optimal, despite it using little information about the state of the replicas [Gupta et al., 2007]. Our previous simulation results [Dürango et al., 2014] show that for self-adaptive, brownout replicas, SQF performs quite well, but can be outperformed by weight-based, brownout-aware solutions. In this article, we combine the two approaches and produce queue-length-based, brownout-aware load-balancing algorithms and show that they are practically applicable for improving resilience in the case of failures leading to service capacity shortage.

6. Conclusion and Future Work

We present a novel approach for improving resilience, the ability to hide failures, in cloud services using a combination of brownout and load-balancing algorithms. The adoption of the brownout paradigm allows the service to autonomously reduce computing capacity requirements by degrading user experience in order to guarantee that response times are bounded. Thus, it provides a natural candidate for resilience improvement when failures lead to capacity shortages. However, state-of-the-art load-balancers are generally not designed for self-adaptive cloud services. The self-adaptivity embedded in the brownout service interferes with the actions of load-balancers that route requests based on measurements of the response times of the replicas.

In order to investigate how brownout can be used for improving resilience, we extended the popular `lighttpd` web server with two new brownout-aware load-balancers. A first set of experiments showed that brownout provides substantial advantages in terms of resilience to cascading failures, even when employing SQF, a state-of-the-art, yet brownout-unaware, load-balancer. A second set of experiments compared SQF to the novel brownout-aware load-balancers, specifically designed to act on a per-request basis. The obtained results indicate that, with high statistical significance, our proposed solutions consistently outperform the current standards: They reduce the user experience degradation, thus perform better at hiding failures. While designed with brownout in mind, PIBH and EPBH may be useful to load-balance other self-adaptive cloud services, whose performance is not reflected in

the response time or queue length.

During this investigation, we highlighted the difference between load-balancers that act whenever a new request is received and algorithms that periodically update the routing weights, finding out that the formers are far more effective than the latter ones. However, the brownout paradigm periodically updates the dimmer values to match specific requirements. A future improvement is to react faster also to events happening at the replica level, therefore redesigning the local replica controller to be event based. In the future, we would also like to design a holistic approach to replica control and load-balancing, extending our replica controllers with auto-scaling features [Ali-Eldin et al., 2012], that would allow to autonomously manage the number of replicas, together with the traffic routing, to obtain a cloud service that is both resilient and cost-effective. Finally, some control parameters were chosen empirically based on the many tests we have conducted. Ongoing work will quality the robustness of the system given the chosen parameters in a more systematic way and for a larger scenario space.

Acknowledgment

This work was partially supported by the Swedish Research Council (VR) for the projects "Cloud Control" and "Power and temperature control for large-scale computing infrastructures", and through the LCCC Linnaeus and ELLIIT Excellence Centers.

References

Ali-Eldin, A., J. Tordsson, and E. Elmroth (2012). "An adaptive hybrid elasticity controller for cloud infrastructures". In: *NOMS*. IEEE. DOI: 10.1109/NOMS.2012.6211900.

Ardekani, M. S., P. Sutra, and M. Shapiro (2013). "Non-monotonic snapshot isolation: scalable and strong consistency for geo-replicated transactional systems". In: *SRDS*. IEEE. DOI: 10.1109/SRDS.2013.25.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003). "Xen and the art of virtualization". In: *SOSP*. ACM. DOI: 10.1145/945445.945462.

Barroso, L. A. and U. Hölzle (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool.

Chen, Y., S. Iyer, X. Liu, D. Milojicic, and A. Sahai (2007). "SLA decomposition: translating service level objectives to system level thresholds". In: *ICAC*. IEEE. DOI: 10.1109/ICAC.2007.36.

Chuah, E., A. Jhumka, S. Narasimhamurthy, J. Hammond, J. C. Browne, and B. Barth (2013). "Linking resource usage anomalies with system failures from cluster log data". In: *SRDS*. DOI: 10.1109/SRDS.2013.20.

Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears (2010). "Benchmarking cloud serving systems with YCSB". In: *SoCC*. ACM. DOI: 10.1145/1807128.1807152.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels (2007). "Dynamo: Amazon's highly available key-value store". *SIGOPS Oper. Syst. Rev.* **41**:6. DOI: 10.1145/1323293.1294281.

Diegues, N. L. and P. Romano (2013). "Bumper: sheltering transactions from conflicts". In: *SRDS*. IEEE. DOI: 10.1109/SRDS.2013.27.

Do, T., M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi (2013). "Limplock: understanding the impact of limpware on scale-out cloud systems". In: *SoCC*. DOI: 10.1145/2523616.2523627.

Doyle, J., R. Shorten, and D. O'Mahony (2013). "Stratus: load balancing the cloud for carbon emissions control". *TCC* **1**:1. DOI: 10.1109/TCC.2013.4.

Dürango, J., M. Dellkrantz, M. Maggio, C. Klein, A. V. Papadopoulos, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén (2014). "Control-theoretical load-balancing for cloud applications with brownout". In: *Conference on Decision and Control (CDC)*. IEEE.

Fallahi, N., B. Bonakdarpour, and S. Tixeuil (2013). "Rigorous performance evaluation of self-stabilization using probabilistic model checking". In: *SRDS*. DOI: 10.1109/SRDS.2013.24.

Fleder, D., K. Hosanagar, and A. Buja (2010). "Recommender systems and their effects on consumers". In: *Electronic Commerce*. DOI: 10.1145/1807342.1807378.

Gallet, M., N. Yigitbasi, B. Javadi, D. Kondo, A. Iosup, and D. H. J. Epema (2010). "A model for space-correlated failures in large-scale distributed systems". In: *Euro-Par*. DOI: 10.1007/978-3-642-15277-1_10.

Gong, Z., X. Gu, and J. Wilkes (2010). "PRESS: predictive elastic resource scaling for cloud systems". In: *CNSM*. IEEE. DOI: 10.1109/CNSM.2010.5691343.

Guan, Q. and S. Fu (2013). "Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures". In: *SRDS*. DOI: 10.1109/SRDS.2013.29.

Gupta, V., M. Harchol Balter, K. Sigman, and W. Whitt (2007). "Analysis of join-the-shortest-queue routing for web server farms". *Perform. Eval.* **64**:9-12. DOI: 10.1016/j.peva.2007.06.012.

Hamilton, J. (2007). "On designing and deploying internet-scale services". In: *LISA*. USENIX, 18:1–18:12.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014). "Brownout: building more robust cloud applications". In: *ICSE*. DOI: 10.1145/2568225.2568227.

Landau, I. D., Y. D. Landau, and G. Zito (2006). *Digital control systems: design, identification and implementation*. Springer.

Lin, M., Z. Liu, A. Wierman, and L. L. H. Andrew (2012). "Online algorithms for geographical load balancing". In: *IGCC*. IEEE. DOI: 10.1109/IGCC.2012.6322266.

Lin, Y. and S. S. Kulkarni (2013). "Automated multi-graceful degradation: a case study". In: *SRDS*. DOI: 10.1109/SRDS.2013.17.

Nah, F. F.-H. (2004). "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour and Information Technology* **23**:3.

Rice University Bidding System (2014). URL: http://rubis.ow2.org.

Schroeder, B., A. Wierman, and M. Harchol-Balter (2006). "Open versus closed: a cautionary tale". In: *NSDI*. USENIX.

Sedaghat, M., F. Hernandez-Rodriguez, and E. Elmroth (2013). "A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling". In: *CAC*. ACM. DOI: 10.1145/2494621.2494628.

Shen, Z., S. Subbiah, X. Gu, and J. Wilkes (2011). "CloudScale: elastic resource scaling for multi-tenant cloud systems". In: *SoCC*. ACM. DOI: 10.1145/2038916.2038921.

Sripanidkulchai, K., S. Sahu, Y. Ruan, A. Shaikh, and C. Dorai (2010). "Are clouds ready for large distributed applications?" *SIGOPS Oper. Syst. Rev.* **44**:2. DOI: 10.1145/1773912.1773918.

Stewart, C. and K. Shen (2005). "Performance modeling and system management for multi-component online services". In: *NSDI*. USENIX, pp. 71–84.

Stewart, C., T. Kelly, and A. Zhang (2007). "Exploiting nonstationarity for performance prediction". In: *EuroSys*. ACM. DOI: 10.1145/1272998.1273002.

Tomás, L. and J. Tordsson (2013). "Improving cloud infrastructure utilization through overbooking". In: *CAC*. ACM. DOI: 10.1145/2494621.2494627.

Tutorial: Installing a LAMP Web Server (2013). URL: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html.

Vasić, N., D. Novaković, S. Miučin, D. Kostić, and R. Bianchini (2012). "DejaVu: accelerating resource allocation in virtualized environments". In: *ASPLOS*. ACM. DOI: 10.1145/2189750.2151021.

Welch, B. (1947). "The generalization of 'student's' problem when several different population variances are involved". *Biometrika* **34**:1-2. DOI: 10.1093/biomet/34.1-2.28.

Yigitbasi, N., M. Gallet, D. Kondo, A. Iosup, and D. H. J. Epema (2010). "Analysis and modeling of time-correlated failures in large-scale distributed systems". In: *GRID*. DOI: 10.1109/GRID.2010.5697961.

Zheng, W., R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner (2009). "JustRunIt: experiment-based management of virtualized data centers". In: *ATC*. USENIX, pp. 18–28.

Paper VI

Model-Based Deadtime Compensation of Virtual Machine Startup Times

Manfred Dellkrantz Jonas Dürango Anders Robertsson
Maria Kihl

Abstract

Scaling the amount of resources allocated to an application according to the actual load is a challenging problem in cloud computing. The emergence of autoscaling techniques allows for autonomous decisions to be taken when to acquire or release resources. The actuation of these decisions is however affected by time delays. Therefore, it becomes critical for the autoscaler to account for this phenomenon, in order to avoid over- or under-provisioning.

This paper presents a delay-compensator inspired by the Smith predictor. The compensator allows one to close a simple feedback loop around a cloud application with a large, time-varying delay, preserving the stability of the controlled system. It also makes it possible for the closed-loop system to converge to a steady-state, even in presence of resource quantization. The presented approach is compared to a threshold-based controller with a cooldown period, that is typically adopted in industrial applications.

Originally published in 10th International Workshop on Feedback Computing, Seattle, Washington, USA, 2015.

1. Introduction

1.1 Background

Cloud computing has in the recent years become the standard for quickly deploying and scaling Internet applications and services, as it gives customers access to computational resources without the need for capital investments. In the Infrastructure as a Service (IaaS) service model, cloud providers rent resources to customers in the form of physical or virtual machines (VMs), which can then be configured by the customers to run their specific application. For a cloud customer aiming at providing a service available to the public, this poses the challenge of renting enough resources for the service to remain available and provide high quality of service (QoS), and the cost of allocating too much resources. Pair this with a workload that is time-varying due to trends, weekly and diurnal access patterns and the challenge becomes more complex.

For this reason, to cope with varying load, cloud services often make use of *autoscaling*, where decisions to adjust resource allocation are made autonomously based on measurements of relevant metrics. There is currently a plethora of different autoscaling solutions available, reaching from simple threshold-based to highly sophisticated based on for example control theory or machine learning. The solutions are commonly categorized as either reactive or proactive to their nature. In the former case, decisions are based on current metric measurements relevant to the load of the cloud service, while in the latter case on a prediction of where the metrics are heading.

Both approaches have in common that they usually do not distinguish between cases where the metrics are only indirectly related to the actual QoS of the cloud service, such as the arrival rate, or metrics that are directly coupled to the QoS, such as response times. From a control theoretical point of view, we could therefore further categorize the first case as feedforward approaches and the second case as feedback approaches. Feedforward control schemes can in many cases give good performance, but generally requires excellent apriori knowledge of the system to be controlled, and lack the ability to detect any changes or disturbances that affect the system. Feedback solutions on the other hand are generally more forgiving when it comes to system knowledge requirements. They can also compensate for unforeseen changes since they base their decisions on metrics directly related to the QoS.

For cloud services, decisions to add more resources usually requires starting up a new VM. This in turn means that the cloud provider needs to place the machine, transfer the OS data it needs and boot it up. Overall, the time from decision to a VM to get fully booted typically ranges from a few tens of seconds up to several minutes [Mao and Humphrey, 2012]. The long time delays this leads to are an inherently destabilizing factor in feedback control. The key reason is the following: long time delays from a scale up decision to a full actuation prompts the feedback controller to continue commanding increased resource provisioning due to the fact that it cannot

yet see the effect of its earlier decisions.

In practice, these time delays need to be considered when designing feedback based autoscaling solutions in order to avoid destabilizing the closed loop system. Possible existing solution include having a low gain in the feedback loop, essentially making the autoscaler very careful with continuing adding more resources before the effect of past decisions start showing up. Another solution is to implement a so-called *cooldown* period, as implemented in [Amazon, 2014; Google, 2014; Rackspace, 2014]. In autoscalers employing cooldown, any decision to scale resources activates the cooldown period, during which subsequent scaling attempts are ignored.

In the current paper, we take a different approach and adopt a solution that has similarities to the Smith predictor, a technique commonly used in control theory for controlling systems with long time delays. In essence, the Smith predictor works by running a model-based simulation of the controlled system without the delays, and use the outputs from this simulation for feedback control. Only if there is a deviation between the true system output and a delayed version of the simulated output are actual measurements from the real system used for control.

1.2 Related work

As cloud computing has grown more popular, the autoscaling challenge has attracted attention and resulted in numerous proposed solutions, for example [Urgaonkar et al., 2008; Gong et al., 2010; Shen et al., 2011]. A thorough review of existing autoscaling solutions can be found in [Lorido-Botrán et al., 2012]. The level at which reconfiguration delays are explicitly considered in existing autoscaling solutions varies depending on the underlying assumption of the magnitude of the delays and choice between feedforward and feedback control structures. Ali-Eldin *et al* [Ali-Eldin et al., 2012] use an approach where scaling down is done reactively and scaling up proactively, but otherwise assumes that any reconfiguration decision is actuated immediately. Similarly, Lim *et al* [Lim et al., 2009] design a proportional thresholding controller with hysteresis where a feedback loop from response times to the number of allocated VMs is closed. Also here the assumption is that VMs can be started instantaneously.

Berekmeri *et al* [Berekmeri et al., 2014] use an empirically identified linear time-invariant model with a time delay to design a controller for deploying resources in a MapReduce cluster to handle incoming work. The time delay corresponds to the reconfiguration delay and is assumed to be constant. As shown by Mao *et al* [Mao and Humphrey, 2012], VM startup times can vary heavily, both depending on application and infrastructure.

In Gandhi *et al* [Gandhi et al., 2012] the authors identify reconfiguration delays as the main reason for poor performance in many reactive and proactive approaches. In their proposed solution, a feedback scheme from the number of concurrently running jobs in a key-value based cloud application is used for scaling up the number

of allocated physical servers. Since starting servers usually takes longer time than shutting them off, they then pack the incoming work on as few servers as possible and equip each server with a timer. If no requests arrive at an empty server during the timer duration, the server is shut down.

1.3 Contribution

In this paper, we present an autoscaling solution using inspiration from the Smith predictor. The result is a feedback controller for cloud services that can quickly reconfigure allocated resources when faced with load variations that leads to a lowered QoS. It also avoids the low controller gains and cooldown solutions otherwise commonly used in feedback autoscalers.

In section 2 we present how a cloud application can be seen as a dynamic mapping from resources to a set of performance metrics, and the proposed delay-compensator. In section 3 we focus on a specific case where we apply our proposed solution to control response times. Simulation results from this scenario are shown in section 4. Section 5 concludes the paper.

2. Delays in cloud applications

2.1 Dynamic mapping

Cloud applications can generally be regarded as software executing on a set of virtualized resources. Their purpose is often to compute a response to requests made to them. This arrival of requests, usually time-varying in its nature, generates a load on the cloud application, which affects the performance and QoS of a cloud application and can be quantified by a number of relevant metrics, such as response times. In order to keep the performance metrics close to some specific value, as specified by a service level objective (SLO), when facing time-varying load, cloud applications are required to be reconfigurable in terms of resources allocated. We have already outlined how a main challenge for this is the long delays when reconfiguring the deployed amount of resources. Further complicating is the fact that virtual resources usually only can be provisioned in a quantized fashion or are available in preset configurations. For example, the number of VMs provisioned must be integer, memory might only be configured in whole gigabytes, etc.

With this in mind, we view a cloud application as a dynamic mapping from deployed resources and incoming load to a set of performance metrics. This gives us the setup shown in Figure 1. Input is the desired amount of resources m and outputs are the actual deployed resources m_r, the metric denoted T, and also we assume that we can measure the incoming load λ. The amount of resources also needs quantization before being actuated.

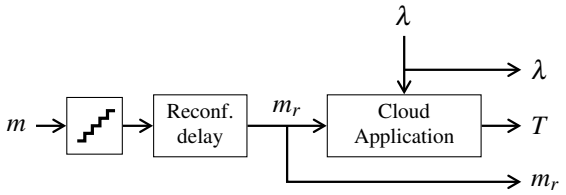

Figure 1. Schematic diagram of the cloud application as a dynamic mapping from desired amount of resources m via deployed resources m_r to the performance metric T. λ is the incoming load of the application and is assumed to measurable. The signal m is also subject to quantization before being sent to the infrastructure.

2.2 Delay compensation

The Smith predictor [Smith, 1957] is commonly used for controlling processes with long time delays, and was originally intended for stable, linear, time-invariant SISO systems with a well-known constant time delay. A key assumption for the Smith predictor is the availability of a delay-free model of the system to be controlled. Using this model, the system's response to a given input can be predicted by running a simulation. An identical, but delayed, simulation is also done using the model. Finally, an aggregated measurement signal $\hat{T}$ that adds the output of the real system T and the delay-free model output T_2 and subtracts the delayed model output T_1 can be formed and used for designing a feedback controller. The result is a situation where the feedback only consists of the delay-free model output if the delayed model and system output perfectly matches each other, allowing for higher control gains. Only when there is a mismatch between model and system is the actual system output used for feedback control.

The Smith predictor usually assumes the actuation delays to be constant, which however, as already mentioned, is generally not true for cloud services. For cloud applications, the delays when reconfiguring the deployed resources are stochastic and may even vary during the day [Mao and Humphrey, 2012]. For this reason we modify the original formulation of the Smith predictor so that the delayed model instead uses m_r, the amount of actually deployed resources, as it is not problematic to measure. This gives the setup shown in Figure 2.

As previously mentioned, resources can usually only be deployed in a quantized fashion. But assuming the delay-free model can handle non-quantized amount of resources (m), our setup also comes with the benefit that even changes in m too small to change the output of the quantization actually has an impact on the compensated response time $\hat{T}$ through the delay-free model.

For the remainder of this paper, we focus on applying our solution to a case where we scale the number of homogeneous VMs allocated to a cloud application to ensure that response times are kept bounded. Note that the key assumption in our approach is that we can model the application. Therefore the compensation

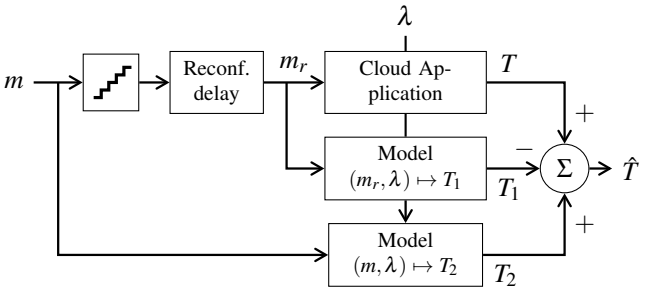

Figure 2. Smith-inspired delay-compensator for cloud applications. The delayed model uses the measured m_r from the cloud application instead using an implementation of a estimate of the delay.

should be applicable also to other types of resources and applications than the one considered here, such as heterogeneous VMs or MapReduce jobs.

3. Response time control

In this section we present a case where the delay compensation described in 2.2 is used. The application under consideration is stateless and the VMs are assumed to be homogeneous. A continuous time dynamic model is derived using queueing theory and the feedback loop for controlling the mean response time is closed using a PI controller. For comparison we also implement a threshold-based autoscaler with cooldown based on [Amazon, 2014].

3.1 Queueing model

Queueing theory is a commonly used approach for modeling servers. For example, in [Cao et al., 2003] measurements from web servers were found to be consistent with an `M/G/1` queueing system. In this paper we model each VM as an `M/M/1` queueing system with service rate μ. Traffic is assumed to arrive to the application according to a Poisson process with intensity λ. A load balancer is then used to spread the traffic randomly over m_r currently running VMs, leading to an arrival rate of $\frac{\lambda}{m_r}$ per VM. A schematic diagram of the model is shown in Figure 3. Response times are recorded and sent to the feedback controller, responsible for reconfiguration decisions. Decisions to scale up come with a stochastic startup delay for each VM. Decisions to scale down are effective immediately, as it can be carried out by simply reconfiguring the load balancer and terminating the VM. The quantization effect in this case consists of a ceiling function to make sure that we get the lowest integer value greater than the desired number of VMs.

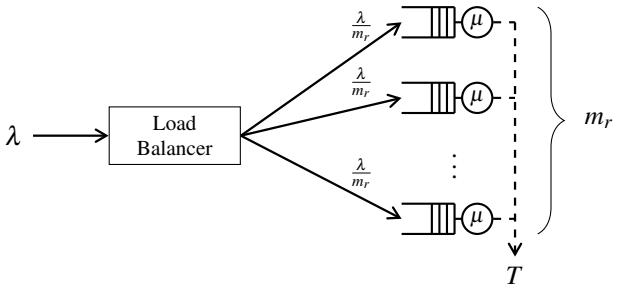

Figure 3. Schematic diagram of the load balancing of m_r running VMs.

3.2 Continuous dynamic approximation

Queueing models are generally mostly concerned with the stationary behavior of a system. However in our case, we are also interested in the cloud application dynamics. By viewing the queueing models considered here as systems of flow, we can use the results from [Agnew, 1976; Rider, 1976; Wang et al., 1996] to formulate the following approximative model of the dynamics of a `M/M/1` queueing system:

$$\begin{aligned} \dot{x} &= f(x,m,\lambda) = \alpha\left(\frac{\lambda}{m} - \mu\frac{x}{x+1}\right) \\ T &= g(x,m,\lambda) = \mu^{-1}(x+1) \end{aligned} \tag{1}$$

where x corresponds to the queue length, λ/m the arrival rate per running VM, μ the service rate of each VM, T the mean response time and α is a constant used in [Rider, 1976] to better fit the transients of the model to experimental data. It is easy to verify that the equilibrium points of the system (1) for any $0 \leq \lambda < \mu$ coincide with the results from a stationary analysis of a `M/M/1` system. In [Tipper and Sundareshan, 1990], Tipper *et al* show how system (1) in the case $\alpha = 1$ provides a reasonable approximation to the exact behavior of the non-stationary `M/M/1` queue as found by numerically solving the corresponding Chapman-Kolmogorov equations under certain conditions. Based on the stationary queue length and the stationary response time of the `M/M/1` we can find the output response time T of the flow model.

From now on we will be using the system (1) and its state variable x as the average state of all VMs. Since all virtual machines are equal it is straight-forward to show that

$$\dot{\bar{x}} = \frac{1}{m}\sum_{i=1}^{m}\dot{x} \approx f(\bar{x},m,\lambda)$$

if we assume all x_i (the states of the individual virtual machine) are the same. This is not true for transients in newly started machines, but as an approximation it is good enough.

Note that system (1) is not dependent on m being integer.

3.3 Control analysis

For control synthesis purposes, we linearize the system equations (1) around the stationary point corresponding to a traffic level λ_0 and response time reference T_{ref}, where we can make use of the fact that stationary queue length x_0 and the stationary number of machines m_0 can be uniquely determined through the other variables as

$$x_0 = T_{ref}\,\mu - 1$$
$$m_0 = \frac{T_{ref}\,\lambda_0}{T_{ref}\,\mu - 1}$$

The linearization yields the following system:

$$\Delta\dot{x} = -\frac{\alpha}{\mu\,{T_{ref}}^2}\Delta x - \alpha\frac{(T_{ref}\,\mu - 1)^2}{{T_{ref}}^2\,\lambda_0}\Delta m + \alpha\frac{T_{ref}\,\mu - 1}{T_{ref}\,\lambda_0}\Delta\lambda$$
$$\Delta T = \mu^{-1}\Delta x$$

Note that the dynamics of the linearized system does not change with varying load, while the input gains do. The transfer function from number of machines m to response time T becomes

$$G_p(s) = \frac{\partial g}{\partial x}\left(s - \frac{\partial f}{\partial x}\right)^{-1}\frac{\partial f}{\partial m}\bigg|_{\substack{x=x_0\\ m=m_0\\ \lambda=\lambda_0}} = -\frac{A}{s+a} \tag{2}$$

with $A = \alpha(T_{ref}\,\mu - 1)^2/({T_{ref}}^2\,\lambda_0\,\mu)$ and $a = \alpha/(\mu\,{T_{ref}}^2)$ both greater than zero.

Since the system is of order one, we conclude that a PI controller of the form

$$G_c(s) = K_p + \frac{K_i}{s} \tag{3}$$

should suffice, leading us to the following closed loop dynamics from T_{ref} to T:

$$G_1(s) = \frac{G_c\,G_p}{1 + G_c\,G_p} = \frac{A(K_p s + K_i)}{s^2 + s(a - A\,K_p) - A\,K_i}. \tag{4}$$

The closed loop dynamics from λ to T is given by the transfer function

$$G_2(s) = \frac{G_p}{1 + G_c\,G_p} = -\frac{As}{s^2 + s(a - A\,K_p) - A\,K_i}. \tag{5}$$

We require of the controller that G_1 and G_2 are asymptotically stable. Furthermore we require that the zero in G_1 is not non-minimum phase. Since this zero also shows up in the transfer function from $\Delta\lambda$ to Δm this would otherwise lead to the controller responding to a step increase in traffic by transiently turning off VMs. Lastly, we require that the transfer functions be fully damped, i.e. that all closed loop poles are

real. This is because we want to avoid overshoots in the control signal when faced with a step shaped disturbance or reference change, as it would lead us to starting up VMs that are almost immediately turned off again. Combining these requirements puts the following constraints on the controller parameters:

$$K_i < 0, \quad K_p \leq 0, \quad -4AK_i \leq (a - AK_p)^2$$

In order to simplify controller design, we can reparameterize the closed loop poles in the following way:

$$s = -\frac{a - AK_p}{2} \pm \sqrt{\frac{(a - AK_p)^2}{4} + AK_i} = -\varphi \pm \xi, \quad \varphi \geq \xi \geq 0$$

allowing us to find the following expression for the controller parameters:

$$K_p = \frac{a - 2\varphi}{A}, \quad \varphi \geq \frac{a}{2}$$
$$K_i = \frac{\xi^2 - \varphi^2}{A}$$

where the condition on φ makes sure that the zero in $G_1(s)$ is minimum phase.

3.4 Threshold-based controller

For comparison we also implement a threshold-based controller with cooldown, based on the autoscaling solution used in Amazon Web Services [Amazon, 2014]. The controller measures the average response times over a time period h, and compares it to two given thresholds, one upper T_{upper} and one lower T_{lower}. Whenever h_t measurements in a row are either above the upper or below the lower threshold, an autoscaling event is triggered, either trying to start or shut down one VM.

Successfully executing an autoscaling event (shutting down or starting up a VM) also starts a cooldown period, with length $h_{cooldown}$. Whenever a cooldown period is running no new autoscaling events are triggered.

4. Experimental Results

4.1 Delay-compensated control

To evaluate the delay-compensator described in Section 2.2 we run a set of discrete event-based simulation experiments. The cloud application is an implementation of the model described in Section 3.1. The PI controller derived in section 3.3 is implemented in discrete time as such:

$$\begin{aligned} e_k &= T_{ref} - \hat{T}_k \\ i_k &= i_{k-1} + K_i h e_k \\ m_k &= K_p e_k + i_k \end{aligned} \tag{6}$$

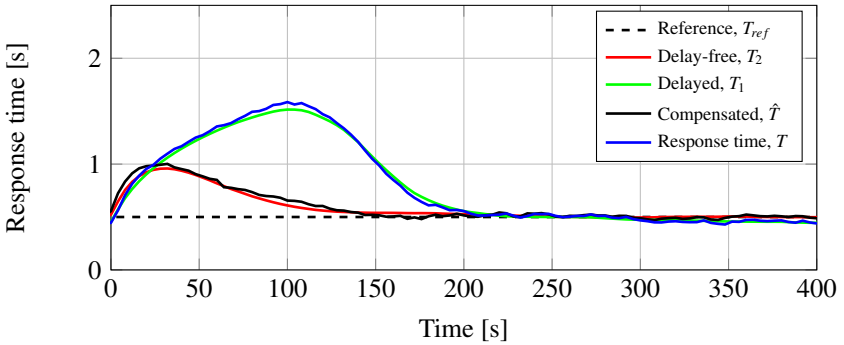

Figure 4. Response time results from simulation of step up. The compensated response times reach the reference much before the actual response times.

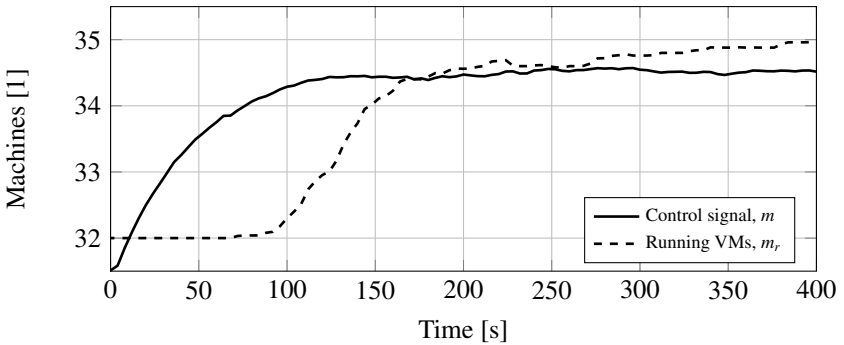

Figure 5. Control signals from simulation of step up. The controller manages to respond to the change in load with little overshoot, which is important.

where m_k is the control signal, i_k is the integrator state and $\hat{T}_k$ is the mean of all delay-compensated response times between sampling points $k-1$ and k. For this implementation we omit anti-windup since the only saturation in the system is $m > 0$, and all experiments are designed to stay far away from that point. The VMs have a service rate $\mu = 22$ and uniformly distributed startup delays in the interval $[80, 120]$ seconds, while shutting down a VM is immediate. The linearization point is chosen as $\lambda_0 = 630$ and $T_{ref} = 0.5$ s, and the controller parameters are chosen so that $\varphi = 0.0545$, $\xi = 0.0432$. The controller runs every $h = 2$ s. Experimental trial showed that using $\alpha = 0.5$ in our cases provided a reasonable transient fit.

The delay compensator updates the state of the delayed and the delay-free model on every request leaving the cloud application. The continuous models are discretized using the Runge-Kutta method.

In the first experiment, the incoming traffic to the application is changed as a

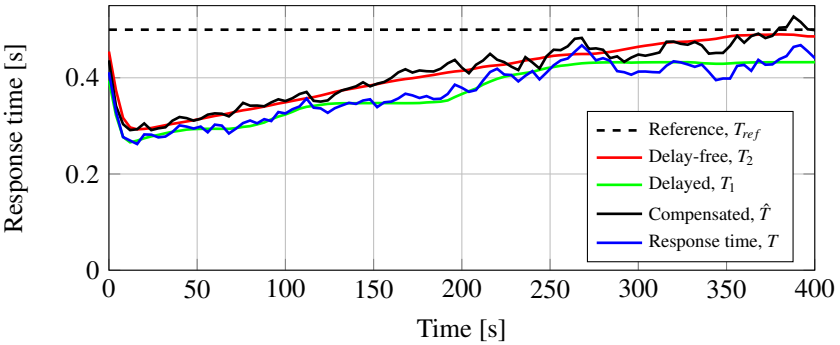

Figure 6. Response time results from simulation of step down. The difference between delayed and delay-free is that the delay-free model has no quantization.

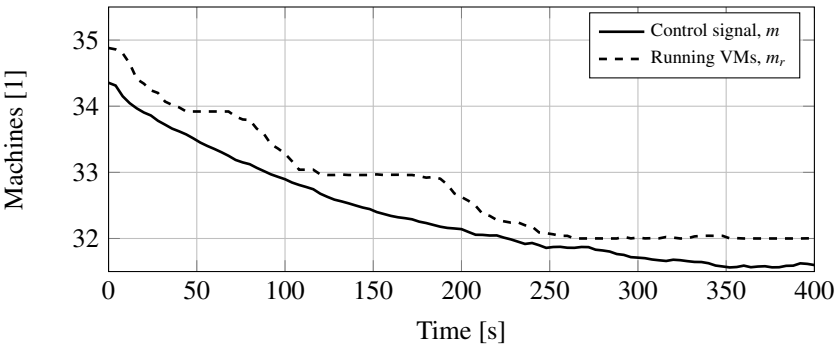

Figure 7. Control signals from simulation of step down. The controller gradually turns off machines to find the equilibrium.

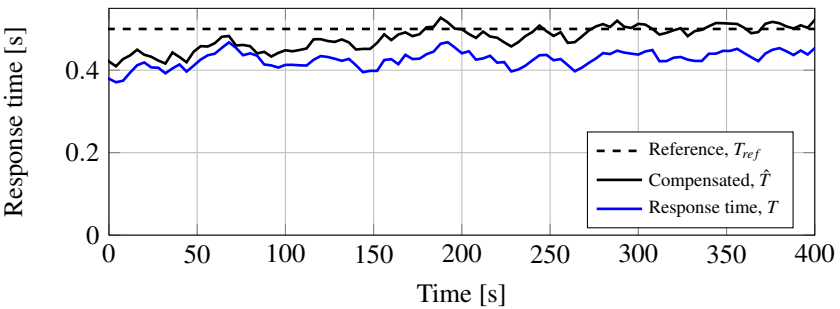

Figure 8. Steady state with $\lambda = 630$. The controller finds the lowest number of machines to come below T_{ref} and then compensates for the difference.

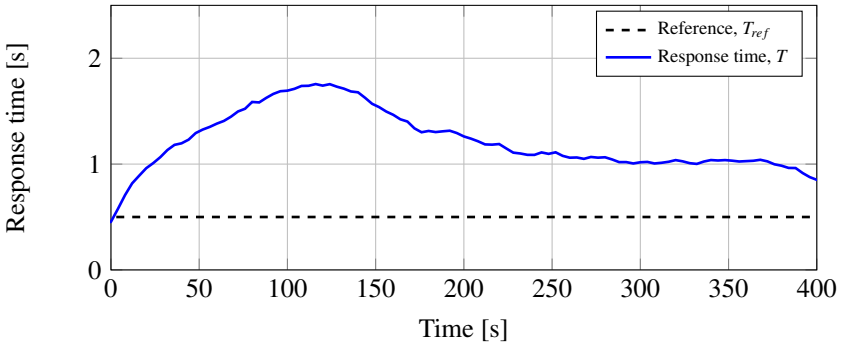

Figure 9. Response times for the step up scenario when using the threshold controller with cooldown

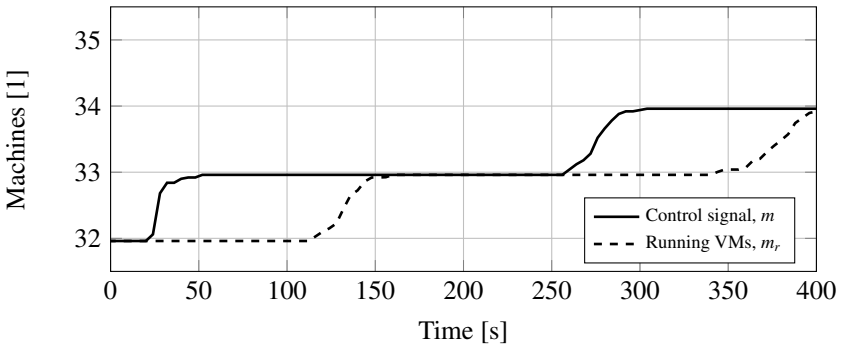

Figure 10. Number of machines for the step up scenario when using the threshold controller with cooldown

step from 630 to 690 requests per second. We perform a set of 25 step response experiments, and aggregate the results to calculate the average response times and number of VMs over a window of 4 seconds. The results are shown in Figures 4 and 5.

As we can see in Figure 4 the real response times reach its highest point about the same time as the first newly started VM becomes active. Figure 5 shows the average control signal (m) and running VMs (m_r). The controller manages to respond to the change in load, without significant overshoot, which is the typical problem caused by actuation delays.

Plots of simulations of the step down from 690 to 630 per second is shown in Figure 6 and 7. The difference between delayed and delay-free model while scaling down is that the delay-free model has no quantization. In less than 300 seconds we reach the theoretical stationary value $m_r = 32$.

Shown in Figure 8 is a plot of the average behavior when the system is approaching steady state with $\lambda = 630$. As can be seen, response times are not varying around T_{ref}, but slightly below. This is because $m_0 = T_{ref}\,\lambda_0/(T_{ref}\,\mu - 1) = 31.5$ is not an integer. Since we can only run integer number of machines and the ideal number is a fraction, an uncompensated PI controller would oscillate between the two values 31 and 32 for m_r. The compensated controller on the other hand finds the smallest integer m_r larger than m_0 and compensates away the part of the error that can not be removed without exceeding T_{ref}. T approaches $T_0 = \mu^{-1}(\frac{\lambda_0}{\mu\lceil m_0\rceil - \lambda_0} + 1) \approx 0.43$ s instead of $T_{ref} = 0.5$ s.

With this controller, for all 25 experiments, we use on average 33.7 machine hours per hour. The mean response time during scale-up is 0.804 seconds and during scale-down 0.373 seconds.

4.2 Threshold-based controller

For comparison we also run the same experiment as previously described with the threshold controller described in 3.4. The controller is run with the parameters $T_{lower} = 0.35$ s, $T_{upper} = 0.6$ s, $h_t = \frac{20\text{ s}}{h}$, $h_{cooldown} = 120$ s.

The mean response times and number of running VMs are shown in Figures 9 and 9 respectively. As we can see the controller does not even manage to get the response times back to the reference value before 400 seconds have passed. Due to the fact that the controller cannot act while in a cooldown period, we respond too slowly to the increase in traffic.

With this controller, for the full experiment, we use 33.3 machine hours per hour. Mean response time during scale-up is 1.224 seconds and during scale-down 0.327 seconds.

4.3 Discussion

As can be seen in Figures 4, 5, 9 and 10 the delay-compensated controller manages to quickly respond to changes in the incoming load. The control signal m reaches its final value of $34 < m < 35$ before the first actual machine has even started. Since the threshold controller needs to wait for its cooldown to pass it is slow to respond. This is also why the delay-compensated controller uses more resources on average.

In Figure 8 we see how we are left with a stationary offset between the response times T and T_{ref}. Since no integer number of virtual machines will result in stationary response times at T_{ref}, the controller finds the lowest amount of machines needed to stay below T_{ref} and then compensates away the error which can't be controlled away.

5. Conclusions

In this paper we have extended the, in the control community, commonly used Smith predictor for compensating for VM startup delay. The classic Smith predictor needs

knowledge about the length of the time delay, but since it is reasonable to assume that we can at all times know the number of currently running VMs we don't need to know or implement the delay. The only thing we need is a model of the behavior of the cloud application after the delay.

Through simulations we show that the compensator can compensate for the startup delay of VMs and that the resource management can be solved using a simple PI controller. Thanks to the delay-compensation the controller can reach the final number of machines before the first machine has even started. The compensator picks the lowest number of VMs which gives response times below the reference.

6. Acknowledgments

This work was supported by the Swedish Research Council (VR) for the project "Cloud Control", and through the LCCC Linnaeus and ELLIIT Excellence Centers.

References

Agnew, C. E. (1976). "Dynamic modeling and control of congestion-prone systems". *Operations research* **24**:3, pp. 400–419.

Ali-Eldin, A., J. Tordsson, and E. Elmroth (2012). "An adaptive hybrid elasticity controller for cloud infrastructures". In: *IEEE/IFIP Network Operations and Management Symposium*. IEEE, pp. 204–212.

Amazon (2014). *Auto scaling concepts — Amazon Web Services documentation*. Accessed: 2014-08-27. URL: `https://web.archive.org/web/20140729191545/http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/AS_Concepts.html`.

Berekmeri, M., D. Serrano, S. Bouchenak, N. Marchand, B. Robu, et al. (2014). "A control approach for performance of big data systems". In: *IFAC World Congress*.

Cao, J., M. Andersson, C. Nyberg, and M. Kihl (2003). "Web server performance modeling using an M/G/1/K* PS queue". In: *10th International Conference on Telecommunications*. Vol. 2. IEEE, pp. 1501–1506.

Gandhi, A., M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch (2012). "Autoscale: dynamic, robust capacity management for multi-tier data centers". *ACM Transactions on Computer Systems (TOCS)* **30**:4, p. 14.

Gong, Z., X. Gu, and J. Wilkes (2010). "Press: predictive elastic resource scaling for cloud systems". In: *International Conference on Network and Service Management*. IEEE, pp. 9–16.

Google (2014). *Google compute engine autoscaler — Google Cloud Platform Documentation*. Accessed: 2014-12-01. URL: https://web.archive.org/web/20141201094332/https://cloud.google.com/compute/docs/autoscaler/.

Lim, H. C., S. Babu, J. S. Chase, and S. S. Parekh (2009). "Automated control in cloud computing: challenges and opportunities". In: *1st Workshop on Automated control for datacenters and clouds*. ACM, pp. 13–18.

Lorido-Botrán, T., J. Miguel-Alonso, and J. A. Lozano (2012). "Auto-scaling techniques for elastic applications in cloud environments". *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09* **12**, p. 2012.

Mao, M. and M. Humphrey (2012). "A performance study on the VM startup time in the cloud". In: *IEEE 5th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 423–430.

Rackspace (2014). *How auto scale cooldowns work — Rackspace Knowledge Center*. Accessed: 2014-11-17. URL: https://web.archive.org/web/20141117122211/http://www.rackspace.com/knowledge_center/article/how-auto-scale-cooldowns-work.

Rider, K. L. (1976). "A simple approximation to the average queue size in the time-dependent M/M/1 queue". *Journal of the ACM (JACM)* **23**:2, pp. 361–367.

Shen, Z., S. Subbiah, X. Gu, and J. Wilkes (2011). "Cloudscale: elastic resource scaling for multi-tenant cloud systems". In: *2nd ACM Symposium on Cloud Computing*. ACM, p. 5.

Smith, O. J. M. (1957). "Closer control of loops with dead time". In: *Chem. Eng. Progr.* Vol. 53, pp. 217–219.

Tipper, D. and M. K. Sundareshan (1990). "Numerical methods for modeling computer networks under nonstationary conditions". *IEEE Journal on Selected Areas in Communications* **8**:9, pp. 1682–1695.

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

Wang, W.-P., D. Tipper, and S. Banerjee (1996). "A simple approximation for modeling nonstationary queues". In: *15th Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation*. Vol. 1. IEEE, pp. 255–262.