Control Strategies for Improving Cloud Service Robustness

Jonas Dürango

LUND
UNIVERSITY

Department of Automatic Control

Lic. Tech. Thesis
ISRN LUTFD2/TFRT--3270--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

This thesis addresses challenges in increasing the robustness of cloud-deployed applications and services to unexpected events and dynamic workloads. Without precautions, hardware failures and unpredictable large traffic variations can quickly degrade the performance of an application due to mismatch between provisioned resources and capacity needs. Similarly, disasters, such as power outages and fire, are unexpected events on larger scale that threatens the integrity of the underlying infrastructure on which an application is deployed.

First, the self-adaptive software concept of brownout is extended to replicated cloud applications. By monitoring the performance of each application replica, brownout is able to counteract temporary overload situations by reducing the computational complexity of jobs entering the system. To avoid existing load balancers interfering with the brownout functionality, brownout-aware load balancers are introduced. Simulation experiments show that the proposed load balancers outperform existing load balancers in providing a high quality of service to as many end users as possible. Experiments in a testbed environment further show how a replicated brownout-enabled application is able to maintain high performance during overloads as compared to its non-brownout equivalent.

Next, a feedback controller for cloud autoscaling is introduced. Using a novel way of modeling the dynamics of typical cloud application, a mechanism similar to the classical Smith predictor to compensate for delays in reconfiguring resource provisioning is presented. Simulation experiments show that the feedback controller is able to achieve faster control of the response times of a cloud application as compared to a threshold-based controller.

Finally, a solution for handling the trade-off between performance and disaster tolerance for geo-replicated cloud applications is introduced. An automated mechanism for differentiating application traffic and replication traffic, and dynamically managing their bandwidth allocations using an MPC controller is presented and evaluated in simulation. Comparisons with commonly used static approaches reveal that the proposed solution in overload situations provides increased flexibility in managing the trade-off between performance and data consistency.

Acknowledgments

I would like to start by thanking my supervisors Bo Bernhardsson and Martina Maggio. Their relentless support and guidance, and ability to quickly take an embryo of an idea and turn it into a research problem, have meant tremendously much to me.

I would also like to thank all my collaborators and coauthors I have worked with over the years, in particular the people in the Cloud Control project, and Erik Elmroth and his research group in Umeå. Finally I know how it feels to be part of the cool research project with the best and most fun workshops! Not counting parts of that last occasion of course, alone in the middle of nowhere in Västerbotten during midwinter... Of my research colleagues, Manfred Dellkrantz deserves special praise. I have spent countless hours in his company discussing everything from research topics to the finer details of Diablo's loot system. Apparently, we make such a good team we get to share workshop accommodation even when everyone else gets a cabin (!) of their own. During the later stages of my work I have very much enjoyed collaborating with William Tärneberg. His perception and impeccable attention to details have played a big part in this thesis being in much better shape than it would have been otherwise.

Over the years I have had the pleasure of sharing offices and getting to know some great people: Jerker Nordh, Björn Olofsson, Ola Johnsson, (and more recently) Martin Karlsson, Victor Millnert and Olof Troeng. Anders Mannesson never truly shared our office, but as he seemed to tolerate the rest of us walking through his room all the time, I consider him an honorary office mate. Having so many bright people around is almost surely (a.s.) a guarantee for someone being able to help you solve a hard problem, or just being up for some fun discussions and banter.

It has in many ways been a privilege to work at such an inspirational, intellectual and fun workplace as I have. Thanks to all my colleagues, former and current, at the department for contributing in making it so. The life of a PhD student would have been a lot less smooth if it was not for the technical and administrative staff keeping things up and running: Eva Westin, Ingrid Nilsson, Mika Nishimura, Monika Rasmusson, Pontus Andersson, Anders Nilsson and Anders

Blomdell. Special thanks to Leif Andersson for opening my eyes to how much more there is to LaTeX to learn. Thanks to Anders Robertsson for proof-reading parts of this thesis.

Finally, I would like to thank my family and friends for their support and encouragement. Lastly, my dearest thanks to Emelie for her unwavering support and love which has kept me going through these years.

Financial support

The work contained in this thesis has been partly funded by the Swedish Research Council (VR) through the LCCC Linnaeus Center and projects “Cloud Control” (VR CLOUD 2012-5908) and “Power and Temperature Control for Large-Scale Computing Infrastructures” (VR 621-2013-5490). The author is a member of the ELLIIT Excellence Center, funded by VINNOVA.

Contents

1

Introduction

1.1 Background and motivation

Today, online services are expected to be able to accommodate large traffic volumes all while providing users with a high Quality of Service (QoS). Cloud computing has in recent years grown to become the de-facto standard for rapidly deploying and scaling such services and applications. Public cloud service providers such as Amazon, Google, Microsoft and Rackspace have given small upstarts hosting opportunities where the tenants pay only for running costs while not having to make the equivalent capital investment in computing infrastructure themselves. In doing so, users have been enabled to almost seamlessly scale up their cloud applications as their popularity grows. Similarly, using the same foundational principles, private and hybrid clouds are enabling larger enterprises to transition many of their core functions to a cloud service model. Recent estimates put expected global spending on cloud solutions by enterprises at $235 billion by 2017, tripling the number from 2011[1]. Cloud computing has also lowered the entry barrier for large-scale scientific computing applications, such as machine learning applications and big data analytics, by enabling users to rapidly and inexpensively provision resources from a virtually infinite pool.

Public cloud providers were previously mainly offering tenants availability guarantees. Now, there is an increasing demand for supporting stricter guarantees on performance and fault tolerance as cloud computing is becoming a viable platform for deploying business critical services. Existing cloud infrastructure and management models are too static to support this in face of ever increasing scale and complexity of cloud-based services, highlighting the need for new integrated and holistic approaches.

Cloud applications are often exposed to dynamic workloads with large variability. This poses a challenge: service predictability with such workloads is difficult to achieve, with applications suddenly going from well-functioning to un-

[1] http://press.ihs.com/press-release/design-supply-chain/cloud-related-spending-businesses-triple-2011-2017, accessed 2016-05-13.

responsive unless properly managed. Moreover, cloud applications are regularly exposed to unexpected events, such as hardware failures, power outages and extreme and sudden traffic surges. The example of the death of Michael Jackson is famous, where the additional traffic almost brought Twitter offline and made Google News wrongly classify related search terms as spam[2]. The work contained in this thesis addresses these challenges in three different directions outlined below.

First, the self-adaptive software concept of brownout [Klein et al., 2014; Maggio et al., 2014] is extended to support cloud applications spanning multiple servers. By designing brownout-compliant load balancers that distribute the traffic across the available servers, brownout applications are enabled to guarantee application performance in settings where workload and infrastructure are variable while also keeping application service levels high. Experimental evaluations in both simulation and in a testbed environment show that the contribution enables cloud applications to withstand traffic surges and infrastructure failures with robust performance while improving service levels as compared to state-of-the-art load balancers.

Next, feedback-based autoscaling for cloud applications is considered, where a model-based controller is designed using feedback from end-user performance, here measured by application response times. To reduce the deleterious effects of time delays caused by Virtual Machine (VM) startup times, a controller structure similar to the classical Smith predictor is derived. Simulation experiments show that the design is able to outperform a widely employed rule-based autoscaler.

Finally the topic of geo-replicated cloud applications for increased disaster tolerance is considered. Using Model Predictive Control (MPC), a solution is presented for managing network bandwidth allocations in high-load situations to deal with trade-offs between a replicated application's performance and robustness to replica failure. Using simulations, it is shown that the proposed solution offers more flexibility in handling dynamic workloads as compared to existing static techniques for managing bandwidth allocation.

1.2 Contributions

The main contributions of this thesis can be summarized as follows:

- Load balancers designed for replicated brownout-enabled applications. These are evaluated against state-of-the-art load balancers in simulation and in testbed.

[2] http://www.telegraph.co.uk/technology/5649500/How-did-Michael-Jacksons-death-affect-the-internets-performance.html, accessed 2016-05-13.

- A Smith predictor-like model-based feedback controller for cloud autoscaling. Simulation evaluation shows that it can outperform a standard rule-based autoscaler in terms of fast control of application response times.
- A dynamic approach based on MPC for network bandwidth allocations in a geo-replicated cloud application. Simulations show that the solution provides more flexibility in handling the trade-off between application performance and disaster tolerance as compared to commonly used static approaches.

1.3 Publications

The following is a list of publications included in this thesis along with a statement of the author's contribution.

Paper I

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, USA.

J. Dürango was the main author of the paper and contributor on the optimization-based solution. He assisted in implementing the other solutions and the simulation framework, and in designing and running the experiments.

Paper II

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 (SRDS)*. Nara, Japan.

J. Dürango, along with M. Dellkrantz, designed and implemented the queue-based load balancers. J. Dürango assisted in designing the experiments and interpreting the results.

Paper III

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.

J. Dürango assisted M. Dellkrantz in designing the delay compensation mechanism and the experiments, designed the controller, and co-wrote the paper.

Paper IV

Dürango, J., W. Tärneberg, L. Tomás, J. Tordsson, M. Kihl, and M. Maggio (2016, submitted). "A control theoretical approach to non-intrusive geo-replication for cloud services". In: *55th IEEE Conference on Decision and Control*. Las Vegas, NV, USA. Submitted.

J. Dürango was the main author of the paper and implemented the simulation framework in collaboration with W. Tärneberg. He also designed the controller and ran the experiments after design inputs from the other authors.

1.4 Additional publications

In addition to the publications above, the following is a list of related publications by the author that are not included in this thesis.

Mehta, A., J. Dürango, J. Tordsson, and E. Elmroth (2015). "Online spike detection in cloud workloads". In: *2015 IEEE International Conference on Cloud Engineering*.

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* **52**, pp. 24–34.

2

Background

This chapter provides the relevant background to the topics of this thesis. First, cloud computing and its underlying concepts are described. Next, a summary of the topics of the thesis is given.

2.1 Cloud computing

Although cloud computing lacks a formal technical definition, many adhere to the definition offered by the National Institute for Standards and Technology (NIST) [Mell and Grance, 2011]:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

The NIST definition goes on by identifying five key characteristics of cloud computing [Mell and Grance, 2011]:

I. On-demand self-service. Users can unilaterally provision computing resources from a service provider without human interaction.

II. Broad network access. Capabilities are widely accessible through standard mechanisms with no focus on a particular client platform.

III. Resource pooling. Service providers pool all available resources and assign them to consumers based on their needs. Multiple consumers may share the same physical resources, but are kept isolated and unaware of this fact.

IV. Rapid elasticity. Resources can be provisioned and released rapidly to follow the needs of a consumer.

V. Measured service. Resource usage is measured by the cloud provider and the measurements are made available to the consumer.

While cloud computing has grown immensely popular in recent years, it is not the result of any particular new disruptive technology, but can rather attributed to the combination and development of several preexisting technologies. Some consider it a fulfillment of computer scientist John McCarthy's vision in the 1960's of computation some day being organized as a utility, available instantly and in virtually unlimited quantities [Garfinkel and Abelson, 1999]. Many underlying concepts of cloud computing are shared with grid computing, which emerged in the 1990's. Grid computing was mainly a response to academia's need of access to vast and inexpensive computing resources on-demand for solving large-scale computational problems [Foster and Kesselman, 2003]. Grid computing differed from traditional supercomputing design by instead employing a distributed infrastructure model with many relatively small computing nodes, often geographically distributed and connected over the public Internet. Using mainly commodity hardware rather than server-grade equipment, nodes were built to be cheap and to fit well with parallelizable workloads. Although cloud computing has retained the same fundamental infrastructure model as grid computing, the two differ in other aspects. [Foster et al., 2008] attributed many of these discrepancies to differing business models: grid computing has traditionally assumed resources being provisioned for long-term or short-term projects with multiple stakeholders, whereas cloud computing has assumed a more consumer-producer like model, where resources are offered to consumers on a fine-grained on-demand basis. To further underline the differences, [Vogels, 2008] lists three additional key features that are novel to cloud computing:

- Computing resources appear to be available on-demand in infinite quantities.
- Users make no up-front commitment, thereby allowing them to start with little resources and increase provisioning as their needs increase.
- The ability to pay for use of computing resources on a short-term basis as needed and release them when no longer needed.

Recent advancements in virtualization techniques have been key in enabling cloud computing to achieve these features [Adams and Agesen, 2006; Barham et al., 2003; Kivity et al., 2007]. Cloud providers use virtualization to offer compute resources in a wide range of virtual CPU, memory, storage and network permutations. Traditionally, hardware virtualization or paravirtualization has been utilized to offer resources in the form of VMs in different configurations, letting consumers configure the environment to their needs. In doing so, multiple tenants can share the same physical infrastructure isolated under the impression that they operate on their own separate hardware. In an effort to reduce overhead in situations where the consumer has no explicit need to run a full VM with its own operating system, recent years have seen the development of operating system-level virtualization as a more lightweight alternative. Solutions like

Linux containers [Linuxcontainers.org, 2016] and Docker [Docker, 2016] allow tenants to run applications in software containers, offering users similar security and performance isolation as VMs but at a much lower startup time. While the use of VMs is still the prevalent solution, software containers are becoming more popular for resource provisioning in cloud computing, and some results indicate that they indeed are able to reduce the performance overhead that virtualization usually carry [Felter et al., 2015].

Cloud computing services are typically made available to consumers using three different service models: Infrastructure as a Service (IaaS), Software as a Service (SaaS) and Platform as a Service (PaaS) [Mell and Grance, 2011]. IaaS offers resources as VMs in different configurations, leaving it to the consumer to configure the setup themselves regarding operating system and applications. The other two service models hide the underlying infrastructure and rather provide the consumer with pre-configured platforms for deploying their own software, or provide an entire software solution that the consumer pays to get access to. In this thesis, it is mainly IaaS solutions, such as Amazon EC2 [Amazon, 2016], that are considered.

2.2 Brownout

Cloud computing infrastructures can be susceptible to failures, such as hardware failures of the physical servers on which an application is running [Barroso et al., 2013]. Unexpected events like these risk degrading the performance of a cloud application as there suddenly can be a mismatch between the application's resource demand and the available capacity. Another example of such an unexpected event is so-called *flash crowds*, which are sudden and unexpected traffic influxes that may increase the resource demand of an application many times over [Bodik et al., 2010; Ari et al., 2003]. Due to the rapid course of events during a flash crowd or hardware failure, there is typically not enough time to fully rely on scaling up an application in order to offset the lost resources or load increase. To cope with these types of unexpected events, strategies not reliant on provisioning additional resources are necessary. Such strategies typically have in common that they increase robustness by temporarily reducing the load on the infrastructure. An often used approach is admission control, where some jobs or requests are refused service so as to reduce the system load and free up resources for other jobs [Kihl et al., 2008].

For a cloud application responding to requests issued by end users, such as a webservice, an alternative approach for maintaining a high performance during a flash crowd or hardware failure is to reduce the computational complexity for some of the requests, based on measurements of the application's performance. In case the application is only temporarily overloaded, it can be advantageous to be able to continue serving all requests and accept some service level degrada-

tion, as compared to admission control where some requests would be outright denied service. A typical example could be an e-commerce site, where a recommendation system is used to recommend to users items they might be interested in, based on their purchase and browsing history. Such a system can provide the site with extra revenue, but is not integral for the site to operate and might be computationally costly. During a temporary overload, rather than denying some users service completely as in admission control, or displaying a default fallback page option, one can alternatively choose to reduce the frequency with which the recommendation system is displayed in order to still be able to process requests within acceptable time. This is the motivation behind *brownout*, which was introduced for cloud applications in [Klein et al., 2014; Maggio et al., 2014].

In a brownout-enabled application, the computations necessary to handle a user request are decomposed into a mandatory part and an optional part. In the example above, the recommendation system would constitute the optional part, while the computations for serving the rest of a request constitute the mandatory part. Brownout then employs a probabilistic solution to decide whether a request should be served the optional content or not. If the application is well-provisioned, the large majority of the requests will get the full content, whereas if the application is under heavy load, the fraction of full requests served is reduced so as to maintain performance. In brownout, application performance is determined from a statistic, either the average or 95th percentile, of the response times of requests served in a sampling interval $[k-1,\ k]$, and is denoted by $t(k)$. A brownout controller then compares $t(k)$ to a setpoint response time t_{ref} and adjusts a so-called *dimmer* in order for the application to achieve the desired performance. The dimmer corresponds to the probability of a request getting served the full content.

Following [Klein et al., 2014], a simple dynamical model of $t(k)$ is used:

$$t(k+1) = \alpha(k)\theta(k) + \delta t(k). \tag{2.1}$$

Here $\alpha(k)$ is a possibly time-varying unknown parameter estimated during runtime as $\hat{\alpha}(k)$ using recursive least squares, $\delta t(k)$ an unknown disturbance and $\theta(k)$ the dimmer. The corresponding transfer function from dimmer to response times is

$$P(z) = \frac{\alpha}{z}. \tag{2.2}$$

By measuring $t(k)$ and comparing it to the setpoint t_{ref}, the brownout controller $C(z)$ for adjusting the dimmer $\theta(k)$ can be designed using pole placement for the pole of the closed loop system $G_{cl}(z)$ from t_{ref} to $t(k)$:

$$G_{cl}(z) = \frac{P(z)C(z)}{1 + P(z)C(z)} = \frac{1-p}{z-p}.$$

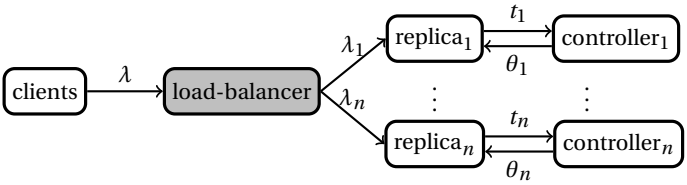

Figure 2.1 System architecture of replicated brownout applications as in Papers I and II. User requests to the application are dispatched to indepent replicas of the application by a load balancer. Each replica contains a separate brownout controller that maintains the response time of its respective replica.

Solving for $C(z)$ and using the estimate $\hat{\alpha}(k)$ gives the brownout controller as an adaptive I controller:

$$C(z) = \frac{1-p}{\hat{\alpha}} \cdot \frac{z}{z-1}. \tag{2.3}$$

In the original work, brownout was designed for applications running on a single server. As cloud applications typically are of larger scale, Papers I and II propose an extension of brownout to a larger setting where the application is replicated across several VMs and a load balancer is used to route traffic to the different replicas. When scaling a brownout application to span multiple VMs, it must be decided where to locate the brownout functionality. In order to promote application robustness, the approach taken in Paper I and II is to make replicas independent by having a local brownout controller in each replica, keeping replicas separated from each other. This approach also makes application scaling relatively straightforward, since scaling up in this setting entails starting another VM, configuring and starting the application and registering the replica with the load balancer to start receiving traffic, without the replicas having to interact with each other. Load balancing is then used to distribute the load over the available replicas. The system setup is illustrated in Figure 2.1. In the simplest case, load balancing can be done either using round robin or by routing a request to a replica chosen at random, giving replica i out of n available replicas a share $\lambda_i = \lambda/n$ of the total traffic λ. However, when the application is deployed on a set of heterogeneous VMs with different capacities, this creates an uneven work distribution, possibly resulting in some replicas becoming overloaded while others are left poorly utilized. This can be mitigated by attributing a weight w_i to each replica according to its capacity, giving the traffic shares

$$\lambda_i = \lambda \cdot w_i, \quad \sum_{i=1}^{n} w_i = 1. \tag{2.4}$$

Other approaches include Shortest Queue First (SQF), where the load balancer routes the each request to the replica with currently fewest requests in

process. However, this requires for the load balancer to keep track of the request count at each replica, which for example in a distributed load balancing setting can result in high communication overhead [Lu et al., 2011].

As traffic is distributed across available replicas, a natural goal to aim for is to maximize the fraction of requests that get served the full content, thereby maximizing the generated revenue. Since existing load balancing strategies are not designed with brownout in mind, they cannot be expected to perform to satisfaction under all circumstances. Moreover, as many load balancers use feedback from the performance of the system there is a risk for the load balancer and brownout functionality to interfere with each other as they both struggle to control the application response times [Mitzenmacher, 2001; Pao and Chen, 2006]. The topic of Paper I is the design of brownout-aware load balancers that handle this source of interference while also aiming to serve as many user requests as possible the full content. For this purpose, a simulation study is done to evaluate the proposed strategies and compare them to a set of widely used state-of-the-art load balancers during scenarios where traffic and infrastructure are allowed to vary. Paper II then extends this work by implementing and evaluating some of the load balancers on a real testbed consisting of a brownout-enabled web application running on several VMs.

2.3 Elasticity control and autoscaling

When cloud applications are subjected to dynamic workloads, such as time-varying request rates, the resulting load on the applications can vary greatly. A well-performing application can during traffic surges quickly become overloaded, resulting in the application becoming unresponsive and facing unacceptable QoS degradation as a consequence, unless properly managed. Similarly during traffic declines, already provisioned resources risk being underutilized if they cannot be put to good use elsewhere. To address this, and to match the allocated resources to the current load, service providers and cloud users utilize elasticity control to scale an application [Herbst et al., 2013]. By continuously re-evaluating its resource needs, a cloud application can ideally be scaled with the load to achieve the goal of high QoS while keeping the costs of the acquired resources down.

Scaling in elastic cloud applications is done in two ways: horizontally and vertically. In horizontal scaling, the resource allocation for an application is modified by adding or removing VMs to which the application is deployed. When new VMs are provisioned, they boot, configure the necessary environments and applications, and register with a load balancer to start receiving jobs. In vertical scaling on the other hand, the VMs themselves are reconfigured during runtime, adding or removing virtual resources such as CPUs, storage and memory. Of the two approaches, horizontal scaling is by far the most widely-used method.

While some scaling decisions can be made manually by operators, it is generally preferable to automate such decisions. This automated decision making process, commonly known as *autoscaling*, allows cloud applications to autonomously scale with the load with little or no manual intervention. Autoscaling has attracted significant research interest in recent years, with proposals to use methods from many different domains such as control theory, time-series analysis and queueing theory [Lorido-Botran et al., 2014]. Autoscaling solutions are usually broadly classified as either reactive or proactive. Reactive autoscalers base their decision on an assessment of the current state of the cloud application [Lim et al., 2010; Lim et al., 2009; Gandhi et al., 2012]. Relevant metrics, such as request rate, load, response time and number of jobs currently being processed are monitored and used to determine if scaling is needed. Given the random and often rapid fluctuations in traffic and load combined with the relatively coarse and slow resource allocation in horizontal scaling, it is not unexpected if such solutions suffer from periods of resource mismatch due to under- and over-provisioning. For this reason, much effort has been put into developing proactive autoscalers where the future state of the cloud application is predicted from measurements and historical data [Gong et al., 2010; Herbst et al., 2014; Ali-Eldin et al., 2012]. These autoscalers are, on the other hand, possibly sensitive to prediction errors, but have nonetheless been shown to often be able to offer better performance than their reactive counterparts in some settings.

Feedforward and feedback autoscaling

From a decision to scale up and add another VM there will be some time before it comes online and is able to process jobs. The exact time can vary, but estimates of boot times of up to 10 minutes are not uncommon [Mao and Humphrey, 2012]. This delay may negatively impact the QoS, but does not pose a threat to the service stability and integrity for autoscalers using metrics unrelated to the state of the application, such as the request rate, as they essentially are feedforward solutions. Using application state metrics, such as response times, on the other hand constitute a feedback solution, to which the inherent delays of starting a new VM can be a destabilizing factor. In state-of-the-art autoscalers found in commercial cloud services, it is customary to address this issue using a cooldown period. It is a time period that starts when a new VM is added, during which the autoscaler is prohibited from starting additional VMs. As soon as the cooldown ends, the autoscaler is again permitted to add or remove VMs. While preventing the autoscaler from continuing to deploy VMs until the last started VM has been able to affect the service performance, it also makes it impossible to start additional VMs during this interval even if it is truly necessary. Alternative approaches to handling the time delays include setting the autoscaler sampling time to the time taken to start new VMs [Ali-Eldin et al., 2012].

Controlling a system using a controller based purely on feedforward generally requires very accurate system models for acceptable control performance. Cloud computing systems are notoriously hard to model, so autoscalers based on feedforward are sensitive to modeling errors or changes in workload composition. For this reason such autoscalers can have a hard time fulfilling strict performance requirements. Feedback can potentially mitigate these problems, but its ability to do so is contingent on what metrics are used. By design, some solutions use feedback from metrics that are only indirectly related to the QoS of the service [Gandhi et al., 2012]. Doing so can be sensible implementation-wise as some metrics may be more easily accessible than others, but requires a good understanding of the relationship between the used metrics and QoS-relevant metrics in order to maintain control of performance.

Paper III presents an autoscaler that uses feedback from actual application performance, as measured by response times, to achieve improved control of the QoS. Furthermore, to reduce the effect of startup delays for new VMs, a delay compensating mechanism similar to the classical Smith predictor [Smith, 1957] is derived. Key in achieving this delay compensation is the ability to accurately model a cloud application. Paper III therefore also presents a novel dynamical model for the response times of a typical cloud application. A derivation of the model is presented in details below. Simulations show that the presented controller is able to provide better and faster control of the performance of a simulated cloud application subjected to traffic variations as compared to a threshold-based autoscaler.

Dynamical modeling of cloud applications

Modeling of computer systems, applications and servers in cloud computing and elsewhere, has historically been done using queueing theory [Kleinrock, 1975; Harchol-Balter, 2013; Cao et al., 2003]. The behavior of servers and applications is described using inter-connected networks of buffers and processors. In the simplest setting, a server can be treated as single node consisting of a processor and a buffer where arriving jobs are enqueued to wait for their turn to be processed. The processor then schedules which of the enqueued jobs to process next according to some scheduling discipline, such as first in, first out (FIFO) or processor sharing [Kleinrock, 1967]. Using the notation of [Kleinrock, 1975], the exogenous arrival of jobs to a server can be described using independent and identically distributed (i.i.d.) samples from an inter-arrival time distribution with cumulative distribution function (CDF) $A(t)$, making the arrival process be of renewal type. Similarly are the service time for the jobs, i.e. the time it would take to process a job if it was alone in the server, described as i.i.d. samples from a service time distribution with CDF $B(t)$.

The single most well-studied and simplest example queueing system is the M/M/1 system [Kleinrock, 1975], where jobs arrive according to a Poisson pro-

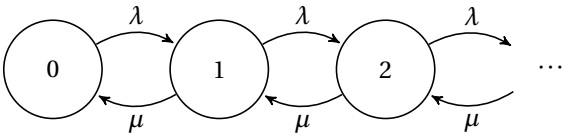

Figure 2.2 State diagram of an M/M/1 queueing system with arrival rate λ and service rate μ.

cess with mean rate λ and the service times are exponentially distributed with mean $1/\mu$, or equivalently the servers having a service rate μ. The M/M/1 system forms a continuous time Markov process on $\mathbb{Z}_{\geq 0}$, where the state denotes the current number of jobs in the system, and can be illustrated by the state diagram in Figure 2.2. By letting $p_k(t)$ denote the probability of the system having exactly k jobs at time t, the corresponding Kolmogorov forward equations for the system are [Kleinrock, 1975]:

$$\begin{aligned} \frac{dp_0(t)}{dt} &= \mu p_1(t) - \lambda p_0(t) \\ \frac{dp_k(t)}{dt} &= \lambda p_{k-1}(t) + \mu p_{k+1}(t) - \big(\mu + \lambda\big) p_k(t), \quad k \geq 1. \end{aligned} \tag{2.5}$$

Under stable conditions, i.e., when $\lambda < \mu$, and by introducing the system utilization $\rho = \lambda/\mu$, it is straightforward to verify from Equation (2.5) for the stationary queue length distribution X that $X \sim \text{Geom}(1-\rho)$ with a probability mass distribution given by

$$p_k = \big(1-\rho\big)\rho^k \tag{2.6}$$

with mean

$$\mathbb{E}[X] = \frac{\rho}{1-\rho}. \tag{2.7}$$

From an end-users perspective, the mean response time T for a job, i.e. the total time spent by a job waiting in the queue and being processed, can be expressed as

$$T = \frac{\mathbb{E}[X]+1}{\mu} = \frac{1}{\mu-\lambda}. \tag{2.8}$$

While results of a stationary analysis are very helpful in e.g. dimensioning server systems, a dynamical analysis is necessary if transients need to be considered, such as when the arrival rate is time-varying. Unfortunately, even for the simplest queueing system, a dynamical analysis quickly becomes cumbersome. For this reason, simplified approximations are desirable. A possible approach is to let $x(t) = \mathbb{E}[X]$ and use Equations (2.5) to show that

$$\frac{dx(t)}{dt} = \sum_{k=0}^{\infty} k\frac{dp_k(t)}{dt} = \lambda(t) - \mu(1 - p_0(t)) = \lambda(t) - \mu\rho(t). \tag{2.9}$$

An obvious complication is the inclusion of the unknowns $p_0(t)$ or $\rho(t)$, neither of which is easily described during non-stationary conditions. [Rider, 1976; Agnew, 1976; Tipper and Sundareshan, 1990] all propose a state dependent approximation on the form

$$\rho(t) = \frac{x(t)}{x(t)+1} \tag{2.10}$$

which, together with Equation (2.9), yields an approximative dynamical model for the queue length of an M/M/1 system as

$$\frac{dx(t)}{dt} = \lambda(t) - \mu\frac{x(t)}{x(t)+1}. \tag{2.11}$$

Note that the model given by Equation (2.11) has a stationary point that corresponds to the mean queue length in Equation (2.7). For the response times, a time-varying version of Equation (2.8) is used to give

$$T(t) = \frac{x(t)+1}{\mu} \tag{2.12}$$

In Paper III, the queue length and response time models given by Equations (2.11)–(2.12) are used to describe a replicated cloud application. Assuming a time-varying request rate $\lambda(t)$ and a variable number of currently running VMs $n(t)$, each with capacity μ, and with randomized load balancing, the dynamics of a replica can be described by

$$\begin{aligned} \frac{dx(t)}{dt} &= \frac{\lambda(t)}{n(t)} - \mu\frac{x(t)}{x(t)+1} \\ T(t) &= \frac{x(t)+1}{\mu}. \end{aligned} \tag{2.13}$$

The model is then used in designing a feedback controller for the response times T to follow a reference T_{ref}. To address the issue of time delays when starting a new VM, the controller uses a mechanism with close resemblance to the classical Smith predictor [Smith, 1957].

2.4 Disaster tolerance in geo-replicated cloud services

As enterprise utilization of public cloud provider infrastructure has grown, the propensity of deploying business critical operations on cloud platforms has increased as well. While this allows users to benefit from the flexibility cloud computing offers, it also exposes them to the risk of infrastructural failures that may

interrupt services, such as power outages, operator misconfiguration and natural disasters. That way, business users are exposed to the risk of large revenue losses or even being put out of business [Keeton et al., 2004; Ji et al., 2003]. Therefore, it is becoming increasingly important for businesses to make sure that applications and services can fully or partly resume operation shortly after a disaster. This is commonly referred to as Business Continuity (BC), and is, in the case of cloud services, achieved using mechanisms for Disaster Recovery (DR) [Wood et al., 2010].

DR mechanisms for cloud applications typically rely on provisioning redundant servers and storage, and keeping one or multiple replicas of an application standing by to take over operation in case of a disaster. Note that this is different from the procedure previously discussed, where replication serves as a means to increase the total capacity of a cloud application. To provide a high degree of disaster tolerance, replicas are kept geographically separated in different Data Centers (DCs). This way, in the unfortunate event of a disaster such as a fire, the main site at which an application is deployed might be brought down, but operation can continue at a remote site. Providing this kind of redundancy entails mirroring the state of an application along with its associated data at the remote sites, so-called geo-replication.

As data is being written by the application, in order to maintain consistency, the corresponding write operations need also to be carried out at the backup sites. Typically, replicating the data is done using either synchronous or asynchronous replication. For write operations at the main replica to complete when using synchronous replication, the corresponding operations need also to be successfully completed and verified at all sites. While providing a high degree of resilience to data loss, synchronous replication can result in degraded application performance as the latency and bandwidth between replicas become a bottleneck for the write throughput [Wood et al., 2011]. This is particularly true for geo-replicated applications, where the physical separation of replicas is likely to negatively impact both latency and bandwidth. Using asynchronous replication, on the other hand, can partly overcome the latency limitation and improve performance by letting write operations complete when the changes have been written only locally at the primary application site. This way, the main application replica is allowed to "pull ahead", potentially leaving the other replicas in an inconsistent state with data not yet replicated buffered until it can be sent. As the application is no longer depending on write operations being completed at all replicas, write throughput can increase, but comes at the cost of potentially losing not-yet replicated data in case of a disaster. A commonly used tool for data replication in Linux-based systems is DRBD [Reisner and Ellenberg, 2005], which provides both synchronous and asynchronous replication modes for block storage devices.

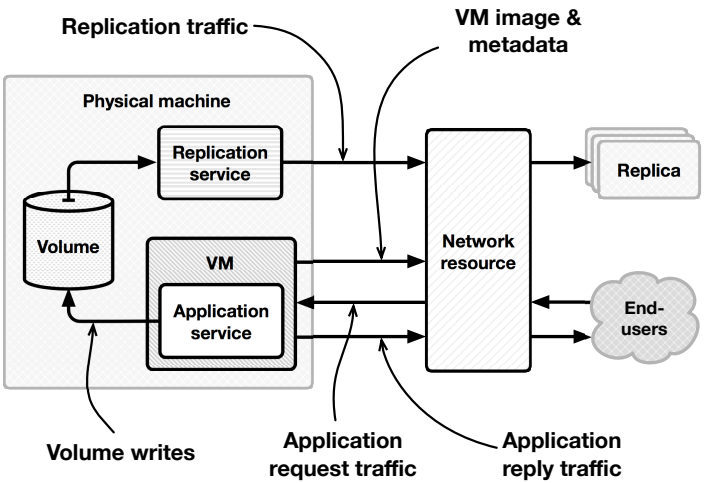

Figure 2.3 Setup of the system considered in Paper IV. A geo-replicated application replies to user requests, while a replication service replicates the data written by the application along with regularly copying an image and corresponding metadata of the VM hosting the application.

When employing DR solutions for cloud services in a geo-replicated setting, such as over a Metropolitan Area Network (MAN) or Wide Area Network (WAN), networking limitations of the infrastructure on which the application is running can also come into play. This is because the networking resources generally are shared between the traffic sent by the application to its end users and the traffic generated by the DR mechanism. The alternative would be to use dedicated networking for the replication traffic, but doing so would generally be too costly for many small and medium-sized businesses. Here, the focus is rather on more cost efficient approaches. Periods of high load can lead to the aggregated bandwidth needs exceeding the network capacity. When this happens, a trade-off between application performance and keeping the replicas consistent is introduced. To minimize the level of interference between application traffic and replication traffic, different traffic control mechanisms can be employed [Hubert et al., 2002]. In some settings, it is customary to restrict replication traffic to use more than certain share of the bandwidth available to the system in order for it to not negatively impact the application performance. In other settings, traffic is managed by assigning priorities to different traffic classes, with higher prioritized traffic, typically belonging to the application, preempting the transmission of lower prioritized traffic.

To improve the flexibility in managing the bandwidth allocation of the different traffic types, Paper IV introduces an approach based on MPC. This way, allo-

cation adjustments can be made dynamically based on the state of the application the level of replica consistency. A setup as illustrated in Figure 2.3 is considered, where an application is responding to requests issued by users. Processing these requests require that the application computes a response, performs necessary write operations and sends the response back to the users. A DR service is running alongside the application, replicating the data written by the application to a remote replica. In addition, with fixed frequency, a full image of the VM on which the application is running along with corresponding metadata is sent to the remote replica. When the networking resources are under heavy load, the system will buffer data sent by each traffic type until it can be transmitted. Based on the content of these buffers, and using a model of their dynamics, the MPC controller adjusts the bandwidth allocations of the system so as to manage the trade-off between application performance and data consistency. Using an event-based simulator, the approach is evaluated and compared to some of the most commonly used static traffic management solutions. The results show that a dynamical approach can increase the flexibility in handling said trade-off in variable scenarios.

Bibliography

Adams, K. and O. Agesen (2006). "A comparison of software and hardware techniques for x86 virtualization". *ACM SIGPLAN Notices* **41**:11, pp. 2–13.

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: *2012 IEEE Network Operations and Management Symposium (NOMS)*, pp. 204–212.

Amazon (2016). *Amazon elastic compute cloud (ec2)*. Accessed: 2016-05-16. URL: `https://aws.amazon.com/ec2/`.

Ari, I., B. Hong, E. L. Miller, S. A. Brandt, and D. D. Long (2003). "Managing flash crowds on the Internet". In: *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, pp. 246–249.

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". *ACM SIGOPS Operating Systems Review* **37**:5, pp. 164–177.

Barroso, L. A., J. Clidaras, and U. Hölzle (2013). *The datacenter as a computer: an introduction to the design of warehouse-scale machines*. 2nd edition. Morgan & Claypool Publishers.

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: *1st ACM symposium on Cloud computing (SoCC)*, pp. 241–252.

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)*. Vol. 2, pp. 1501–1506.

Docker (2016). *Docker website*. Accessed: 2016-05-16. URL: `http://www.docker.com`.

Felter, W., A. Ferreira, R. Rajamony, and J. Rubio (2015). "An updated performance comparison of virtual machines and linux containers". In: *2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172.

Foster, I. and C. Kesselman (2003). *The grid 2: blueprint for a new computing infrastructure*. Elsevier.

Foster, I., Y. Zhao, I. Raicu, and S. Lu (2008). "Cloud computing and grid computing 360-degree compared". In: *2008 Grid Computing Environments Workshop*, pp. 1–10.

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.

Garfinkel, S. and H. Abelson (1999). *Architects of the information society: 35 years of the laboratory for computer science at MIT*. MIT press.

Gong, Z., X. Gu, and J. Wilkes (2010). "Press: predictive elastic resource scaling for cloud systems". In: *2010 International Conference on Network and Service Management (CNSM)*, pp. 9–16.

Harchol-Balter, M. (2013). *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.

Herbst, N. R., N. Huber, S. Kounev, and E. Amrehn (2014). "Self-adaptive workload classification and forecasting for proactive resource provisioning". *Concurrency and Computation: Practice and Experience* **26**:12, pp. 2053–2078.

Herbst, N. R., S. Kounev, and R. Reussner (2013). "Elasticity in cloud computing: what it is, and what it is not". In: *10th International Conference on Autonomic Computing (ICAC)*, pp. 23–27.

Hubert, B., G. Maxwell, R. Van Mook, M. Van Oosterhout, P. B. Schroeder, and J. Spaans (2002). "Linux advanced routing & traffic control". In: *Ottawa Linux symposium*, p. 213.

Ji, M., A. C. Veitch, and J. Wilkes (2003). "Seneca: remote mirroring done write". In: *2003 USENIX Annual Technical Conference (ATC)*, pp. 253–268.

Keeton, K., C. Santos, D. Beyer, J. Chase, and J. Wilkes (2004). "Designing for disasters". In: *3rd USENIX Conference on File and Storage Technologies (FAST)*, pp. 59–62.

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.

Kivity, A., Y. Kamay, D. Laor, U. Lublin, and A. Liguori (2007). "Kvm: the linux virtual machine monitor". In: *Proceedings of the Linux symposium*. Vol. 1, pp. 225–230.

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)*, pp. 700–711.

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

Kleinrock, L. (1975). *Queueing systems, volume I: theory*. Wiley Interscience.

Lim, H. C., S. Babu, and J. S. Chase (2010). "Automated control for elastic storage". In: *7th International Conference on Autonomic Computing (ICAC)*, pp. 1–10.

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*, pp. 13–18.

Linuxcontainers.org (2016). *Linux containers*. Accessed: 2016-05-16. URL: `http://www.linuxcontainers.org`.

Lorido-Botran, T., J. Miguel-Alonso, and J. A. Lozano (2014). "A review of auto-scaling techniques for elastic applications in cloud environments". *Journal of Grid Computing* **12**:4, pp. 559–592.

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". *Performance Evaluation* **68**:11.

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

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

Mell, P. and T. Grance (2011). *The NIST definition of cloud computing*. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

Mitzenmacher, M. (2001). "The power of two choices in randomized load balancing". *IEEE Transactions on Parallel and Distributed Systems* **12**:10, pp. 1094–1104.

Pao, T.-L. and J.-B. Chen (2006). "The scalability of heterogeneous dispatcher-based web server load balancing architecture". In: *7th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 213–216.

Reisner, P. and L. Ellenberg (2005). "Drbd v8 - Replicated storage with shared disk semantics". In: *12th International Linux System Technology Conference (Linux-Kongress)*.

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

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.

Vogels, W. (2008). “A head in the clouds - the power of infrasctructure as a service”. In: *1st Workshop on Cloud Computing and in Applications (CCA)*.

Wood, T., E. Cecchet, K. K. Ramakrishnan, P. Shenoy, J. V.D.Merwe, and A. Venkataramani (2010). “Disaster recovery as a cloud service: economic benefits & deployment challenges”. In: *2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*.

Wood, T., H. A. Lagar-Cavilla, K. K. Ramakrishnan, P. Shenoy, and J. V.D.Merwe (2011). “Pipecloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery”. In: *2nd ACM Symposium on Cloud Computing (SoCC)*.

Paper I

Control-theoretical load-balancing for cloud applications with brownout

Jonas Dürango Manfred Dellkrantz Martina Maggio
Cristian Klein Alessandro Vittorio Papadopoulos
Francisco Hernández-Rodriguez Erik Elmroth Karl-Erik Årzén

Abstract

Cloud applications are often subject to unexpected events like flash crowds and hardware failures. Without a predictable behavior, 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.

 Originally published in *Proceedings of 53rd IEEE Conference on Decision and Control* (CDC), Los Angeles, USA, December 2014.

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 et al., 2013]. 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 et al., 2013; 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 et al., 2013; 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 et al., 2013; 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, 2006] proposed a load balancing solution using the remaining capacity of the replicas to determine how the next request should be managed. 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, 2013], 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,

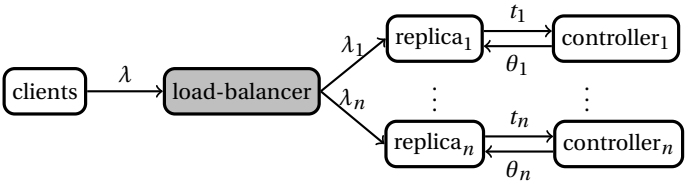

Figure 1. Architecture of a brownout-compliant cloud application featuring multiple replicas.

we tried to mimic as follows: A *Predictive* load balancer would rank the 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 [Maggio 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 et al., 2013].

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 re-

quests to complete a goal, such as buying a product [D. F. García and J. 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 standalone 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

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

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

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

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

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

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

$$\begin{aligned} \underset{w_i}{\text{maximize}} \quad & J = \sum_i w_i \theta_i = \sum_i w_i \frac{A_i - B_i w_i}{C_i + D_i w_i} \\ \text{subject to} \quad & \sum_i w_i = 1, \\ & \frac{A_i - C_i}{B_i + D_i} \leq w_i \leq \frac{A_i}{B_i} \end{aligned} \tag{11}$$

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.

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

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.

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 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 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

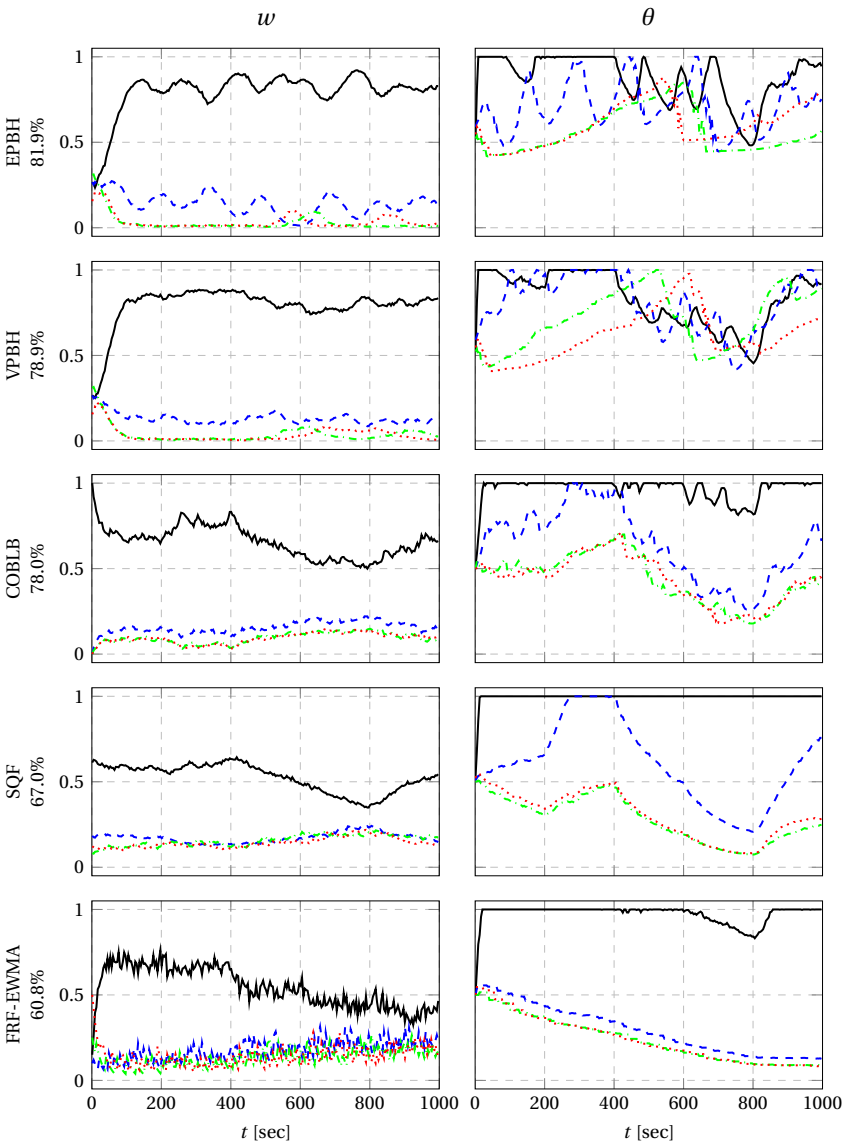

Figure 2. Results of a simulation with four replicas and clients entering and leaving the system at different time instants. The left column shows the effective weights while the right column shows the control variables for each replica. The first replica is shown in black solid lines, the second in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

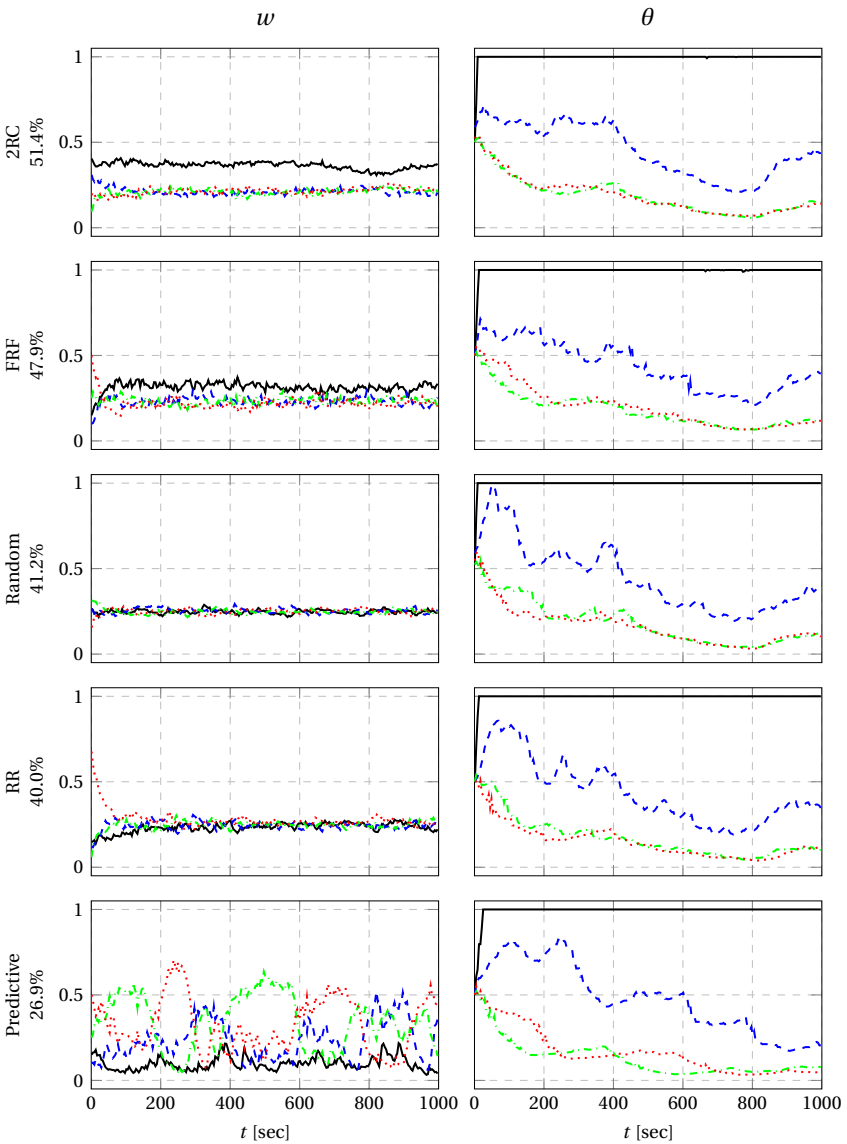

Figure 2. (continued) Results of a simulation with four replicas and clients entering and leaving the system at different time instants. The left column shows the effective weights while the right column shows the control variables for each replica. The first replica is shown in black solid lines, the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

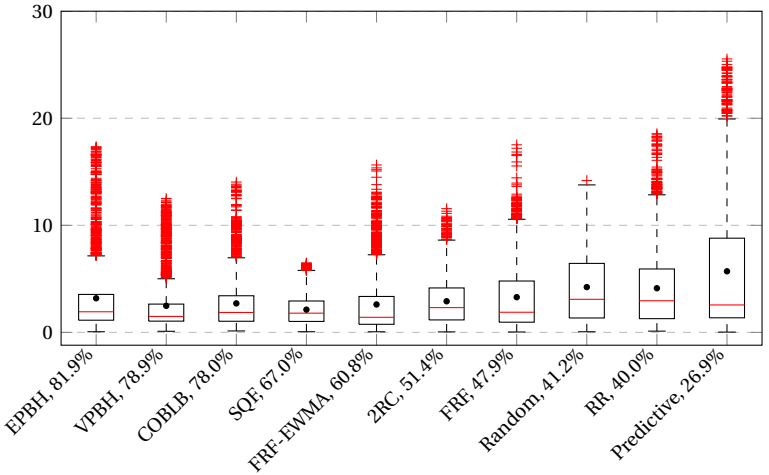

Figure 3. Box plots of the maximum response time in all the replicas for every control interval. Each box shows from the first quartile to the third. The red line shows the median; outliers are represented with red crosses while the black dots indicate the average value (also considering the outliers).

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. 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

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 information 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: *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*.

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

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

Barroso, L. A., J. Clidaras, and U. Hölzle (2013). *The datacenter as a computer: an introduction to the design of warehouse-scale machines*. 2nd edition. Morgan & Claypool Publishers.

BIG-IP (2013). *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: *1st ACM symposium on Cloud computing (SoCC)*, pp. 241–252.

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)*. Vol. 2, pp. 1501–1506.

Cardellini, V., M. Colajanni, and P. S. Yu (2003). "Request redirection algorithms for distributed web systems". *IEEE Transactions on Parallel and Distributed Systems* **14**:4.

Casolari, S., M. Colajanni, and S. Tosi (2009). "Self-adaptive techniques for the load trend evaluation of internal system resources". In: *5th International Conference on Autonomic and Autonomous Systems (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: *Real-Time and Embedded Technology and Applications Symposium (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: *American Control Conference*.

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. DOI: 10.1109/TCC.2013.4.

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: *3rd USENIX Conference on Hot topics in Cloud Computing (HotCloud)*.

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

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: *39th IEEE Conference on Decision and Control (CDC)*.

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)*, pp. 700–711.

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: *2012 International Green Computing Conference (IGCC)*. DOI: 10.1109/IGCC.2012.6322266.

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". *Performance Evaluation* **68**:11.

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

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 colocations". In: *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 248–259.

Mitzenmacher, M. (2001). "The power of two choices in randomized load balancing". *IEEE Transactions on Parallel and Distributed Systems* **12**:10, pp. 1094–1104.

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, pp. 223–240.

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: *7th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 213–216.

Patterson, R. H., G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka (1995). "Informed prefetching and caching". In: *15th ACM Symposium on Operating Systems Principles (SOSP)*.

Ranjan, S., R. Karrer, and E. Knightly (2004). "Wide area redirection of dynamic content by internet data centers". In: *23rd Conference of the IEEE Communications Society (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 Transactions on Autonomous and Adaptive Systems (TAAS)* **4**:2, 14:1–14:42.

Schroeder, B., A. Wierman, and M. Harchol-Balter (2006). "Open versus closed: a cautionary tale". In: *3rd Conference on Networked Systems Design & Implementation (NSDI)*.

Stankovic, J. A. (1985). "An application of bayesian decision theory to decentralized control of job scheduling". *IEEE Transactions on Computers* **34**:2.

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

Tomás, L. and J. Tordsson (2013). “Improving cloud infrastructure utilization through overbooking”. In: *2013 ACM Cloud and Autonomic Computing Conference (CAC)*. DOI: 10.1145/2494621.2494627.

Wang, L., V. Pai, and L. Peterson (2002). “The effectiveness of request redirection on cdn robustness”. In: *5th Symposium on Operating Systems Design and Implementation (OSDI)*.

Wolf, J. L. and P. S. Yu (2001). “On balancing the load in a clustered web farm”. *ACM Transactions on Internet Technology* **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: *IEEE International Conference on Communications (ICC)*.

Paper II

Improving cloud service resilience using brownout-aware load-balancing

Cristian Klein Alessandro Vittorio Papadopoulos
Manfred Dellkrantz Jonas Dürango Martina Maggio
Karl-Erik Årzén Francisco Hernández-Rodriguez Erik Elmroth

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 et al., 2013; 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, 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

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

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 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 θ affects 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 selec-

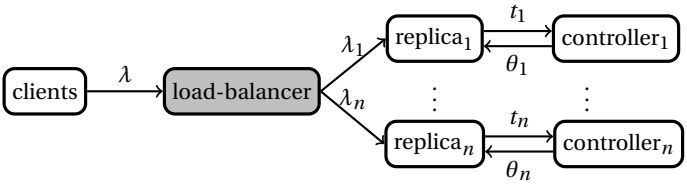

Figure 1. Architecture of a brownout cloud service featuring multiple replicas.

tion 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 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

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

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 et al., 2013].

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 smoothen 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

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

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 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 [Amazon, 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; Z. Shen et al., 2011; Zheng et al., 2009; Stewart and K. 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

[3] `https://github.com/cloud-control/brownout-lb-lighttpd`
[4] 2100 MHz, 16 cores per processor, no hyper-threading.
[5] `https://github.com/cloud-control/brownout-lb-rubis`

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 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.

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

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 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

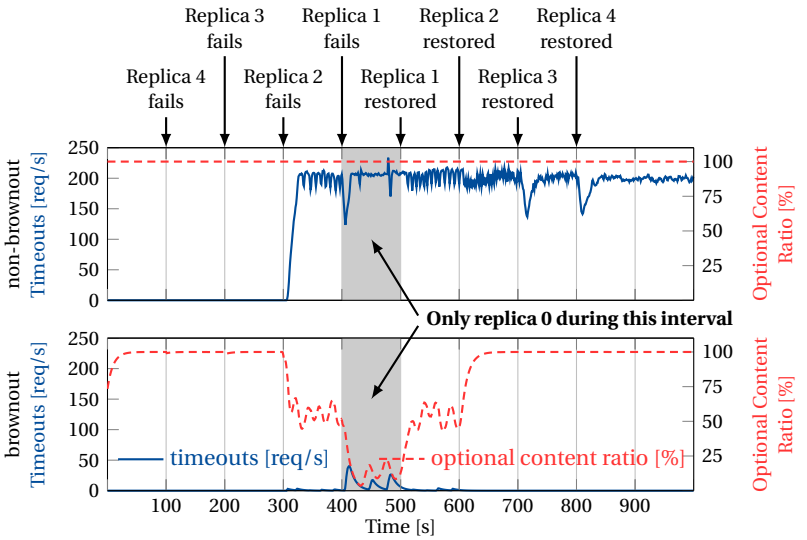

Figure 2. Experimental results comparing resilience without and with brownout. Configuration: 5 replicas, each having 4 cores.

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 ser-

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%

vice 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 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 scenar-

ios. 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.

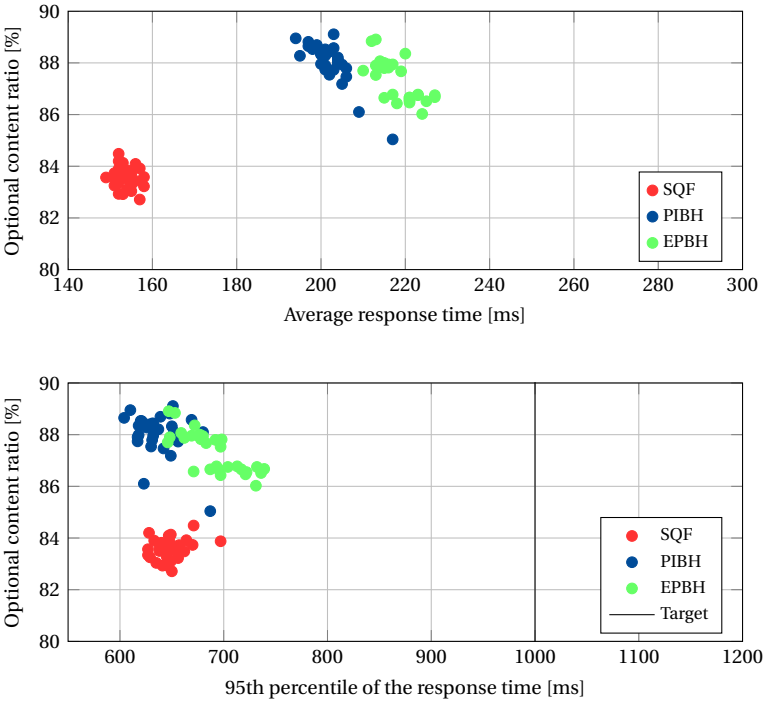

Figure 3. Comparison of SQF and brownout-aware load-balancing algorithms when two replicas have 1 core and three replicas have 8 cores.

"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 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

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** ($p < 10^{-15}$)

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**($p < 10^{-9}$)

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

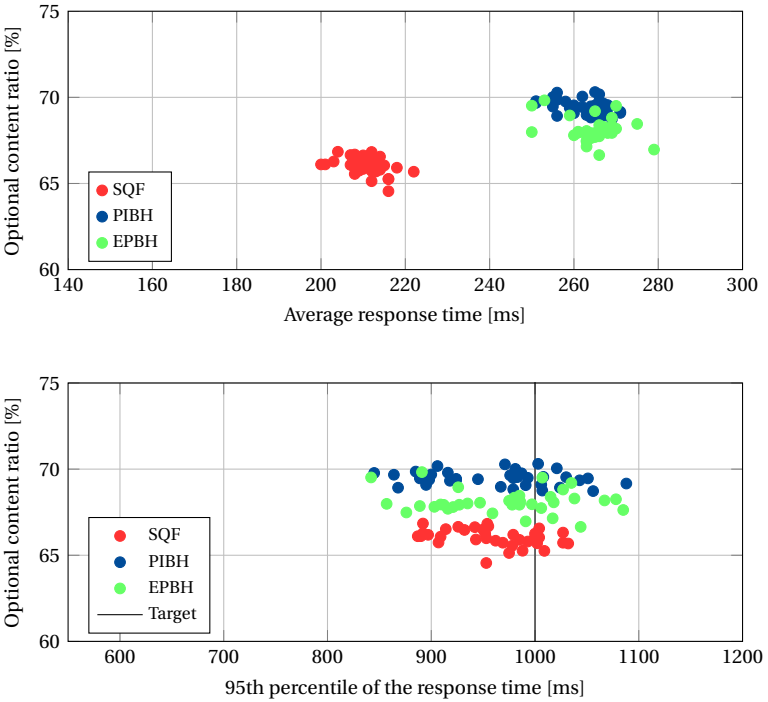

Figure 4. Comparison of SQF and brownout-aware load-balancing algorithms when three replicas have 1 core and two replicas have 8 cores.

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** ($p < 10^{-15}$)

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 set-

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**($p = 0.2265$)

point (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 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 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* [Y. 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 et al., 2013; 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 [M. 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.

References

Ali-Eldin, A., J. Tordsson, and E. Elmroth (2012). “An adaptive hybrid elasticity controller for cloud infrastructures”. In: *2012 IEEE Network Operations and Management Symposium (NOMS)*, pp. 204–212.

Amazon (2013). *Tutorial: installing a LAMP web server*. URL: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html`.

Ardekani, M. S., P. Sutra, and M. Shapiro (2013). “Non-monotonic snapshot isolation: scalable and strong consistency for geo-replicated transactional systems”. In: *32nd IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 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". *ACM SIGOPS Operating Systems Review* **37**:5, pp. 164–177.

Barroso, L. A., J. Clidaras, and U. Hölzle (2013). *The datacenter as a computer: an introduction to the design of warehouse-scale machines*. 2nd edition. Morgan & Claypool Publishers.

Chen, Y., S. Iyer, X. Liu, D. Milojicic, and A. Sahai (2007). "SLA decomposition: translating service level objectives to system level thresholds". In: *4th International Conference on Autonomic Computing (ICAC)*. 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: *32nd IEEE International Symposium on Reliable Distributed Systems (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: *1st ACM symposium on Cloud computing (SoCC)*. 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". *ACM SIGOPS Operating Systems Review* **41**:6. DOI: 10.1145/1323293.1294281.

Diegues, N. L. and P. Romano (2013). "Bumper: sheltering transactions from conflicts". In: *32nd IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 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: *4th ACM Symposium on Cloud Computing (SoCC)*. DOI: 10.1145/2523616.2523627.

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. 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: *53rd IEEE Conference on Decision and Control (CDC)*.

Fallahi, N., B. Bonakdarpour, and S. Tixeuil (2013). "Rigorous performance evaluation of self-stabilization using probabilistic model checking". In: *32nd IEEE International Symposium on Reliable Distributed Systems (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: *2010 International Conference on Network and Service Management (CNSM)*, pp. 9–16.

Guan, Q. and S. Fu (2013). "Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures". In: *32nd IEEE International Symposium on Reliable Distributed Systems (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". *Performance Evaluation* **64**:9-12. DOI: 10.1016/j.peva.2007.06.012.

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

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)*, pp. 700–711.

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: *2012 International Green Computing Conference (IGCC)*. DOI: 10.1109/IGCC.2012.6322266.

Lin, Y. and S. S. Kulkarni (2013). "Automated multi-graceful degradation: a case study". In: *32nd IEEE International Symposium on Reliable Distributed Systems (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: *3rd Conference on Networked Systems Design & Implementation (NSDI)*.

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: *2013 ACM Cloud and Autonomic Computing Conference (CAC)*. 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: *2nd ACM Symposium on Cloud Computing (SoCC)*, p. 5.

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

Stewart, C., T. Kelly, and A. Zhang (2007). "Exploiting nonstationarity for performance prediction". In: *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. DOI: 10.1145/1272998.1273002.

Stewart, C. and K. Shen (2005). "Performance modeling and system management for multi-component online services". In: *2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 71–84.

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

Vasić, N., D. Novaković, S. Miučin, D. Kostić, and R. Bianchini (2012). "DejaVu: accelerating resource allocation in virtualized environments". In: *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 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: *11th IEEE/ACM International Conference on Grid Computing (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: *2009 USENIX Annual Technical Conference (ATC)*, pp. 18–28.

Paper III

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 at *10th International Workshop on Feedback Computing*, Seattle, USA, April 2015. The article has been reformatted to fit the current document.

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 IaaS service model, cloud providers rent resources to customers in the form of physical or 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 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 a priori 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 provision-

ing 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-Botran et al., 2014]. 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., 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., 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.

[Berekmery 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 [Mao and Humphrey, 2012], VM startup times can vary heavily, both depending on application and infrastructure.

In [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.

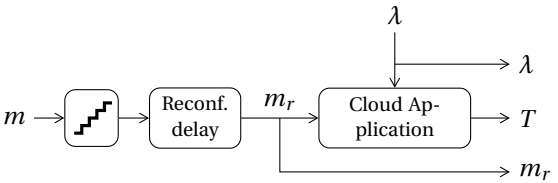

Figure 1. Schematic diagram of the cloud application as a dynamic mapping from desired amount of resources m via deployed resources m_r to the performance metric T. λ is the incoming load of the application and is assumed to measurable. The signal m is also subject to quantization before being sent to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

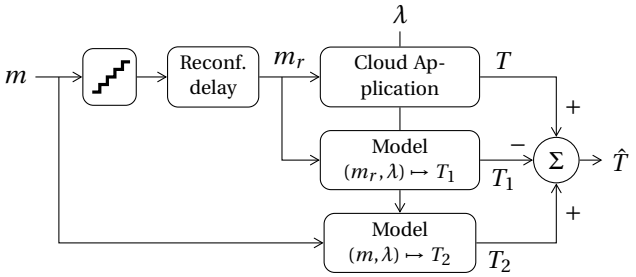

Figure 2. Smith-inspired delay-compensator for cloud applications. The delayed model uses the measured m_r from the cloud application instead using an implementation of a estimate of the delay.

pensation 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 Section 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.

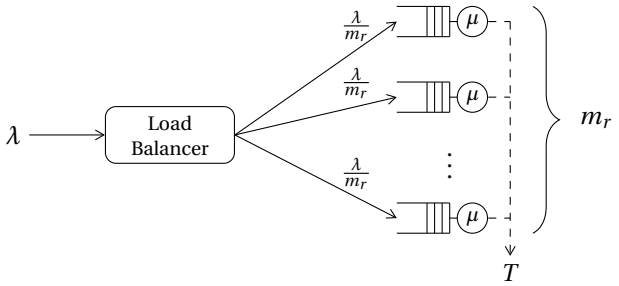

Figure 3. Schematic diagram of the load balancing of m_r running VMs.

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], it is shown how the system given by Equation (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

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

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

The linearization yields the following system:

$$\begin{aligned} \Delta\dot{x} &= -\frac{\alpha}{\mu\,{T_{\text{ref}}}^2}\Delta x - \alpha\frac{(T_{\text{ref}}\,\mu-1)^2}{{T_{\text{ref}}}^2\,\lambda_0}\Delta m + \alpha\frac{T_{\text{ref}}\,\mu-1}{T_{\text{ref}}\,\lambda_0}\Delta\lambda \\ \Delta T &= \mu^{-1}\Delta x. \end{aligned} \tag{2}$$

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

$$G_p(s) = \frac{\partial g}{\partial x}\left(s - \frac{\partial f}{\partial x}\right)^{-1}\frac{\partial f}{\partial m}\bigg|_{\substack{x=x_0\\ m=m_0\\ \lambda=\lambda_0}} = -\frac{A}{s+a} \tag{3}$$

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

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

$$G_c(s) = K_p + \frac{K_i}{s} \tag{4}$$

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

$$G_1(s) = \frac{G_c\,G_p}{1+G_c\,G_p} = \frac{A(K_p s + K_i)}{s^2 + s(a - AK_p) - AK_i}. \tag{5}$$

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

$$G_2(s) = \frac{G_p}{1+G_c\,G_p} = -\frac{As}{s^2 + s(a - AK_p) - AK_i}. \tag{6}$$

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:

$$K_i < 0$$

$$K_p \leq 0$$

$$-4AK_i \leq (a - AK_p)^2.$$

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 imple-

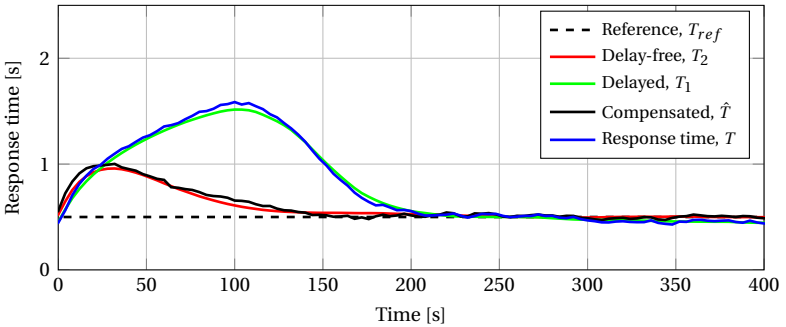

Figure 4. Response time results from simulation of step up. The compensated response times reach the reference much before the actual response times.

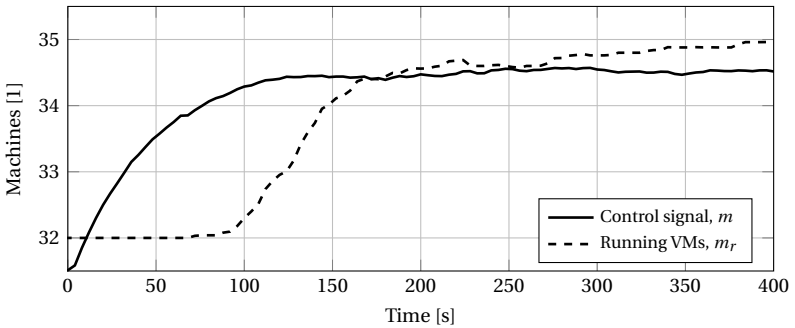

Figure 5. Control signals from simulation of step up. The controller manages to respond to the change in load with little overshoot, which is important.

mentation of the model described in Section 3.1. The PI controller derived in section 3.3 is implemented in discrete time as such:

$$\begin{aligned} e_k &= T_{\text{ref}} - \hat{T}_k \\ i_k &= i_{k-1} + K_i\, h\, e_k \\ m_k &= K_p\, e_k + i_k \end{aligned} \tag{7}$$

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 $T_{\text{ref}} = 0.5$ 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 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 Figures 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 $m_0 = T_{\text{ref}}\lambda_0/(T_{\text{ref}}\mu - 1) = 31.5$ 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 $T_{\text{ref}} = 0.5$ 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 $T_{\text{lower}} = 0.35$ s, $T_{\text{upper}} = 0.6$ 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 10 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.

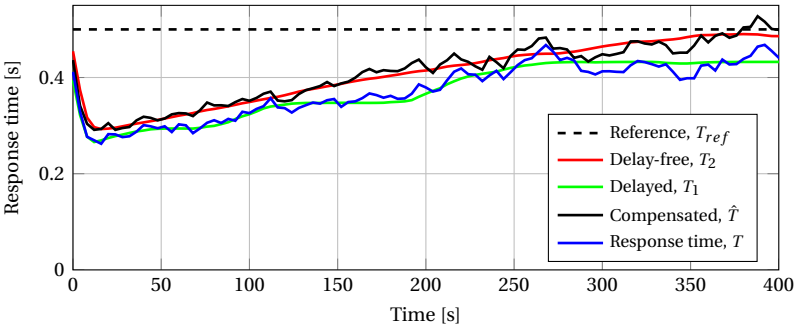

Figure 6. Response time results from simulation of step down. The difference between delayed and delay-free is that the delay-free model has no quantization.

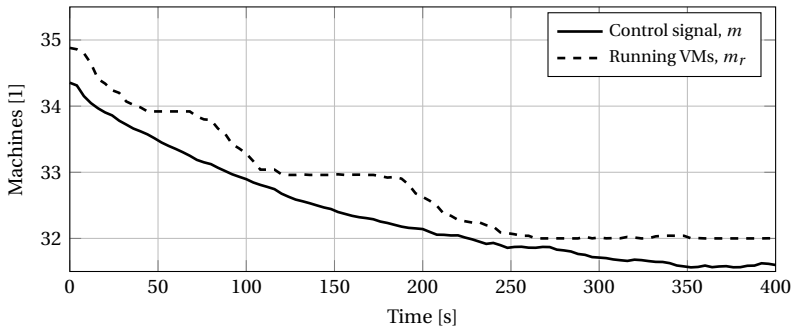

Figure 7. Control signals from simulation of step down. The controller gradually turns off machines to find the equilibrium.

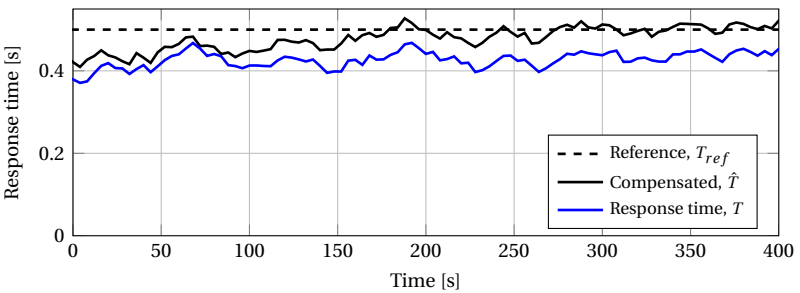

Figure 8. Steady state with $\lambda = 630$. The controller finds the lowest number of machines to come below T_{ref} and then compensates for the difference.

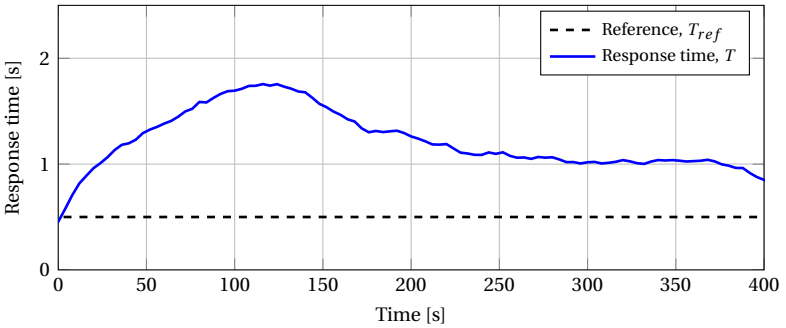

Figure 9. Response times for the step up scenario when using the threshold controller with cooldown.

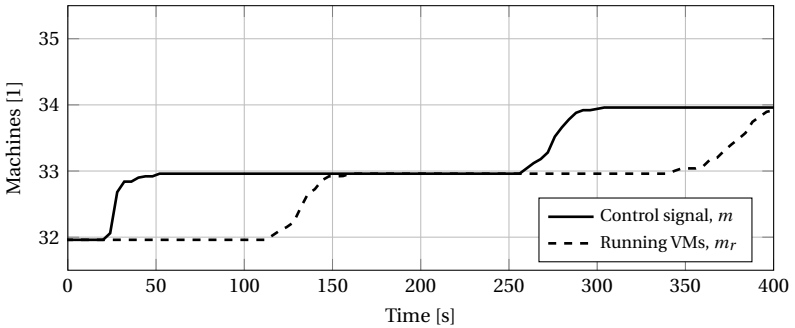

Figure 10. Number of machines for the step up scenario when using the threshold controller with cooldown.

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. Conclusion

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.

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: *2012 IEEE Network Operations and Management Symposium (NOMS)*, pp. 204–212.

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

Berekmery, M., D. Serrano, S. Bouchenak, N. Marchand, B. Robu, et al. (2014). "A control approach for performance of big data systems". *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 (ICT)*. Vol. 2, 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: *2010 International Conference on Network and Service Management (CNSM)*, pp. 9–16.

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

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*, pp. 13–18.

Lorido-Botran, T., J. Miguel-Alonso, and J. A. Lozano (2014). "A review of auto-scaling techniques for elastic applications in cloud environments". *Journal of Grid Computing* **12**:4, pp. 559–592.

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

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

Rider, K. L. (1976). "A simple approximation to the average queue size in the time-dependent M/M/1 queue". *Journal of the ACM* **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 (SoCC)*, 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: *IEEE INFOCOM*. Vol. 1, pp. 255–262.

Paper IV

A control theoretical approach to non-intrusive geo-replication for cloud services

Jonas Dürango William Tärneberg Luis Tomás Johan Tordsson Maria Kihl Martina Maggio

Abstract

Complete data center failures may occur due to disastrous events such as earthquakes or fires. To attain robustness against such failures and reduce the probability of data loss, data must be replicated in another data center sufficiently geographically separated from the original data center. Implementing geo-replication is expensive as every data update operation in the original data center must be replicated in the backup. Running the application and the replication service in parallel is cost effective but creates a trade-off between potential replication consistency and data loss and reduced application performance due to network resource contention. We model this trade-off and provide a control-theoretical solution based on Model Predictive Control to dynamically allocate network bandwidth to accommodate the objectives of both replication and application data streams. We evaluate our control solution through simulations emulating the individual services, their traffic flows, and the shared network resource. The MPC solution is able to maintain the most consistent performance over periods of persistent overload, and is quickly able to indiscriminately recover once the system return to a stable state. Additionally, the MPC balances the two objectives of consistency and performance according to the proportions specified in the objective function.

Submitted to the *55th IEEE Conference on Decision and Control (CDC)*, Las Vegas, December 2016.

1. Introduction

Today, there is an ever increasing reliance on cloud services for business critical operations. Outsourcing operational applications to one vendor expose businesses to potential revenue losses incurred by for example downtime due to failures [Patterson et al., 2002]. In cloud computing, failures are the norm rather than an exception. Failures are unpredictable and may happen at any time, as exemplified by the cascading power blackout that swept cities from Detroit to New York City in 2003 [Barron, 2003]. The interruption in business continuity and the information lost when storage devices or a complete DC hosting an application fail can even put entire enterprises out of business [Keeton et al., 2004]. Two out of five enterprises that experience a disaster are out of business within five years [R. Witty, 2001] from the outage. Furthermore, cost estimations for data unavailability can reach millions of Euros per hour [Ji et al., 2003]. These events and revelations have incited the development of DR schemes that provide reduced interruption of service in case of disasters.

Current DR schemes typically achieve redundancy by mirroring all relevant data on the application's primary operational node to one or multiple secondary replicas. The replicas are persistently standing by to assume the responsibility for hosting the applications, in the event the primary fails. In order to be tolerant to disasters severe enough to bring down an entire DC, such as a fire or an earthquake, replicas are kept geographically separated, known as geo-replication. As a result, the applications can stay available even as the primary replica is lost or becomes unreachable [Ji et al., 2003]. However, such DR solutions increase the overall network traffic from the primary node shared between the replication service and the applications. The additional traffic can lead to network contention between the occupants on the primary node during high loads. To mitigate this potential contention, the system administrators typically assign a static quota for the network bandwidth allotted to the replication service traffic. As an example, the Distributed Replicated Block Device (DRBD) replication tool documentation recommends a 30% bandwidth allotted to the replication service traffic[1].

Such solutions are inherently inflexible, as they do not cope well with irregular traffic patterns and the heterogeneous objectives of the different streams, manifested in their different goals. The replication service will seek to maintain the replicas as closely synchronized as possible to minimize potential data loss and unavailability in case of a failure. It does so by attempting to minimize the delay imposed to each write operation. On the other hand, the application traffic needs to be served at a certain rate to meet performance objectives, e.g., end-user response time. Therefore, there is an inherent *trade-off* between data consistency and delivered application performance, which strongly depends on the available bandwidth. We argue in this paper that these conflicting goals can be

[1] https://drbd.linbit.com/en/users-guide/s-configure-sync-rate.html

managed using a dynamic bandwidth allocation approach.

In this paper, we propose a dynamic bandwidth allocation solution for DR systems based on MPC. We propose a two fold solution: (1) differentiating between different traffic flows and concurrently use different replication modes; and (2) an MPC solution that dynamically adjusts resource allocations for the different traffic flows over time based on the prevailing conditions in an attempt to meet their individual performance objectives. Our proposed solution dynamically adapts to changes in egress (outgoing) traffic and provides a cost-optimized scheduling of bandwidth, in order for the replication service traffic to be handled with low latency at the same time as the ordinary application traffic can keep its performance objectives. The fundamental principal is a holistic one, we allow the controller to compromise the throughput of one the services while maintaining its performance goals to meet the objectives of achieving better overall system performance and cost-efficiency, with the resources at hand. We validate our strategy with a simulator that executes a variety of workloads and measures the amount of data loss in case of a disaster. Finally, to evaluate the replication performance of our solution, we formulate a performance metric that captures the momentary disaster recovery readiness of the system.

2. Related work

In this section we provide an overview of fault tolerance and disaster tolerance techniques for cloud services, and of replication challenges in general. We begin by discussing prevalent replication techniques, their inherit challenges, and the current state of the research in that area. We then tie it into the problem we are addressing.

Making DR cost-efficient is a significant research area [Ji et al., 2003]. More and more companies are focusing on recovery plans, in an attempt to achieve what is generally known as business continuity [Wood et al., 2010]. The main principal of business continuity is to offer application owners the assurance that their applications will have as few service interruptions as possible. To achieve this, many business services utilize (1) fault tolerance techniques and (2) disaster recovery techniques. Fault tolerance techniques, such as Remus [Cully et al., 2008] or COLO [Dong et al., 2013] are used to recover from sporadic failures by synchronizing what a VM is doing into a secondary copy of the VM.

Other work highlight the importance of data replication and try to reduce the incurred cost of replication. One example is [Cidon et al., 2015]. In this case, the main objective is data durability and how to protect against both independent and correlated node failures by means of a tiered replication scheme that splits the cluster into a primary and a backup tier. Regarding disaster recovery techniques, in [Wood et al., 2010] the authors propose to use a public cloud to recover in case of a disaster instead of a backup site.

Replication incurs additional operations during the normal execution of the DC. In general, in response to client-issued requests, applications continuously write data onto their attached virtual disks. As part of the DR solution, a replication service is then responsible for mirroring the write operations at the secondary replica. Such mirroring can be carried out by either synchronous or asynchronous write operations. Synchronous writes provide a higher degree of data consistency between replicas as each write operation at the primary replica has to be verified to have been carried out also at the secondary replica before completing. Pipecloud [Wood et al., 2011] is a synchronous backup strategy that addresses the impact of replication latency on performance by efficiently overlapping replication with application processing for multi-tier servers. However, as write operations must await response from the backup site before completing, synchronous backup guarantees consistency at the expense of collocated services sharing the same resource.

In a MAN or WAN setting however, bandwidth limitations and high latency can make replication unacceptably slow, as the network connectivity between replicas becomes a performance bottleneck. To avoid this, performance can be improved at the expense of consistency guarantees by using asynchronous replication. In this case, the primary replica is essentially allowed to pull ahead of the secondary replica by completing write operations when they have been made to the local file system, without waiting for the secondary replica. The replication service is then responsible for carrying out the write operations at the secondary replica. This clearly creates some inconsistencies, until the write operations have been propagated to the secondary replica, but at the same time avoids performance bottlenecks. For instance, in SnapMirror [Patterson et al., 2002], batches of updates are periodically sent to the backup site, aiming at trading off cost and performance. SnapMirror's asynchronous solution does, however, not offer continuous mirroring but only guarantees that the copies are in sync at the backup instants. The degree of replica consistency is thus proportional to the delay incurred by the intermediate network and the availability of shared resources.

One frequently employed service for replicating file systems in Linux systems is DRBD [Reisner and Ellenberg, 2005] and DRBD Proxy, which have support for both synchronous and asynchronous replication modes. The DRBD [Reisner and Ellenberg, 2005] asynchronous replication mode sends data continuously but only waiting for the acknowledge that the packages has reached the send-TCP buffer in the local server, unlike synchronous mode that waits for the acknowledgement of the write operation at the remote location. DRBD is our choice as an enabling technology for the design of our DR solution.

As regards to the interference between the replication service traffic and the normal DC operation, besides the well known techniques to differentiate traffic flows at routing level (such as DiffServ or IntServ[2]), there are tools available for

[2] https://tools.ietf.org/html/draft-ietf-diffserv-rsvp-02

traffic sharing, allowing to differentiate traffic per process or flow at server level. For instance, Dusia et al. present a network quality of service guaranteeing approach [Dusia et al., 2015] capable of prioritizing some processes (in their case containers) by making use of the Linux traffic control (TC) utility. However, they define an static setting, not aware of current buffer status or data flow needs.

3. System architecture model

In this section we outline the system architecture model used in this paper. Emphasis is given to describing traffic streams and system components through which they can be managed. We consider applications that are hosted as typically for cloud applications, i.e. in a primary replica executing in either a VM or container that in turn is hosted on a PM in a DC. Requests from clients are received by the application, in turn prompting the computation of responses that are returned to the issuers.

3.1 Dynamic control for concurrent flows transmission

Apart from mirroring the data that the application writes during runtime, the DR service also needs to transfer the information regarding the VMs running the application, including the VMs image and meta-data on its current state such as virtual disks attached (known as volumes) and network configurations. Since such configurations usually are not frequently updated, the transfer of the corresponding data to the remote site is usually initiated at set time intervals. Figure 1 provides an overview of the observed system.

To reduce cost, the replication service is not given a dedicated interface for its traffic. Instead, the network resources are shared between the application and replication service. This introduces inherent conflicting goals, since the QoS of an application is typically directly related to the rate at which it can serve requests from clients. On the other hand, the degree to which the application remains disaster tolerant is subject to the rate at which the application write operations can be mirrored to the remote replica and how expediently the transfer of VM images and related meta-data can be completed. By not considering this trade-off, high load situations can lead to unacceptable service degradation or disaster tolerance. On the other hand, existing solutions to addressing said trade-off can prove too inflexible in a dynamic setting where traffic conditions are subject to unpredictable changes. Hence the need for a dynamic solution.

3.2 Flows differentiation and traffic model

The solution proposed in this paper builds on differentiating the three traffic flows described previously, denoted *Application*, *Replication service*, and *VM image*, and taking into account their different time-variant features when managing them. Note that the approaches considered here are agnostic to and de-

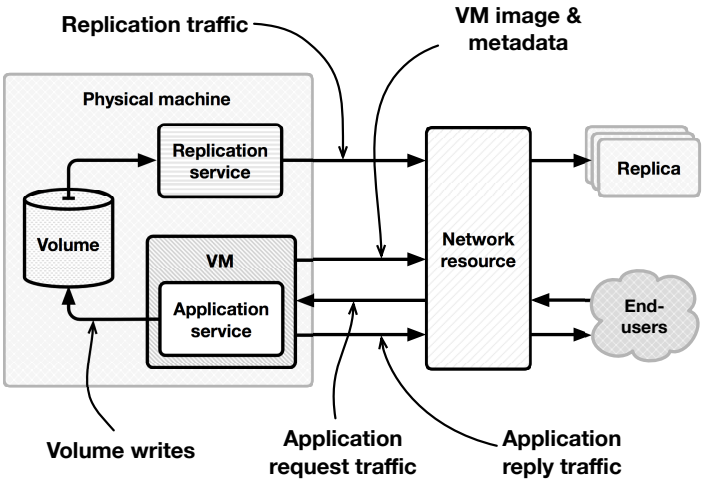

Figure 1. System abstraction for the set-up considered from the primary replica's viewpoint. The application replies to incoming client requests, while the replication service is responsible for mirroring the data written by the application as well as the state of the VM at the remote replica. The application and replication service need to share a common network resource.

coupled from the actual application and the replication service. This is to make the approaches as general and as portable as possible, and to facilitate easier deployment in a future testbed. Separating the solution from the nature of the application means that traffic sent by the *Application* and *Replication service* from the traffic management component's point of view can be seen as exogenously generated. The *Application* and *Replication service* traffic streams generally have time-varying rates at which data need to be sent. In this paper, we assume the traffic streams to be non-homogeneous Poisson processes [Harchol-Balter, 2013]. The *VM image* flow is also exogenous to the traffic management, but for reasons outlined earlier it can be assumed to instead arrive in bulks at given time intervals. An illustration of the structure of the traffic management solutions discussed here is given in Figure 2.

When there is data ready for transmission, the system needs to decide which differentiated traffic stream will get access to the network resource. In Linux and other operating systems, egress network traffic can be managed, policed, and shaped through a QDisc [Dusia et al., 2015], which acts as a scheduler on the outgoing interface. By default most systems do no traffic differentiation and the QDisc acts as a simple FIFO buffer. This can be disadvantageous in some circumstances as it allows one traffic stream to grab an unproportionally large share of the bandwidth by sending packets at a high rate. A more competent alterna-

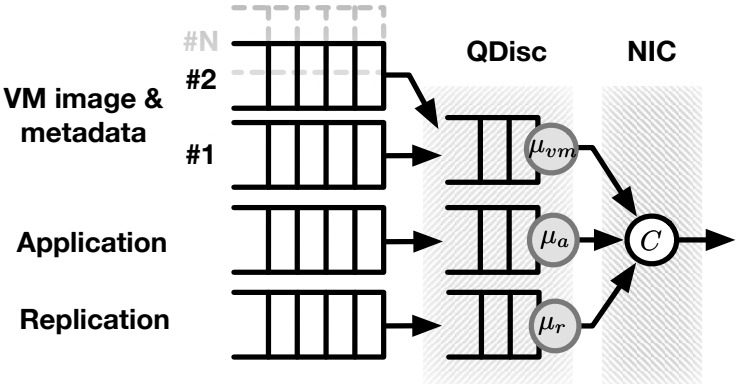

Figure 2. Structural overview of system components relevant to traffic management. The *Replication service*, *Application* and individual *VM image* copy streams are differentiated. Multiple VM image copies are kept differentiated from each other. Streams try to deposit packets in target QDisc buffers and will wait if the target buffer is full.

tive, QDiscs such as Hierarchical Token Bucket (HTB) [Devera and Cohen, 2002] employ filters to give the system administrator large freedom in managing outgoing traffic per traffic type. Combining differentiation to classify traffic streams and HTB filters, individual traffic streams can be allotted a share of the network bandwidth over a certain time period, and bandwidth sharing hierarchies can be constructed. This way, each traffic stream is guaranteed to receive its share of the bandwidth and is not vulnerable to bandwidth hoarding streams in the way the FIFO solution is. Yet another commonly used approach for traffic management is to prioritize traffic streams, where higher ranked streams persistently pre-empts lower ranked streams. This is supported by default in the QDisc structure in most Linux distributions, pfifo_fast, which combines FIFO scheduling with three priority levels. Traffic streams are given a priority, and packets are buffered in three FIFO buffers, one for each priority level. The QDisc then schedules packets for transmission from buffers in falling priority order. This type of traffic management is ideal for allowing interactive latency-sensitive applications transmitting relatively little data to still access the network resource while larger bulk-type transfers also are active.

It is worth noting that by their nature, buffer sizes of QDiscs are relatively small, typically ranging from kilobytes to a few megabytes. Accordingly, the QDisc buffers cannot be expected to be able to accommodate all traffic at all times, in particular during sudden bursts and when the system is intermittently overloaded. In a real system, when a buffer fills up, the transportation layer would incur *back-pressure* in the QDisc, thereby forming a closed loop. In this paper we do not explicitly consider back-pressure as it goes against the application agnos-

tic approach taken. Instead each traffic stream is kept in its own infinite buffer that deposits its content into the target QDisc buffer. For *VM image* traffic this has some implications. In our set-up, whenever the DR system schedules a VM image copy, it is treated as a bulk arrival of packets that are deposited in its own buffer, which in turn tries to make deposits in the target QDisc buffer. If a copy is scheduled to start before a previous one is finished, two streams would try to deposit in the same target buffer. Effectively, this halves the potential bandwidth available to each VM image copy stream until one of them is finished. It is again worth noting that as the proposed method is agnostic to the hosted services it cannot control the arrival of VM image copies.

4. Control design

This section describes our proposal for dynamically adjusting bandwidth shares to the traffic streams identified in Section 3. It is based on the HTB filter approach, where traffic streams are differentiated and allocated a guaranteed minimum share of the available network bandwidth. Typically, the guaranteed minimum share is set statically depending on some knowledge of the system requirements. In contrast to that, our MPC controller dynamically adjusts the guaranteed minimum share for each of the traffic stream using feedback of the current state of the system.

Let $\lambda_i(k)$, $i \in I = \{a, r, vm\}$ denote the amount of data that the DC is requested to transmit in the sampling interval $[k, k+1]$ for each of the streams: *Application* (a), *Replication service* (r) and *VM image* traffic (vm). These requests are considered to be exogenous to the traffic management system. Also, based on the total available network bandwidth for the primary replica, let C denote the total amount of data that can be sent during a sampling period, C being dependent on the DC network link.

We denote with u_i the control signal that we use, which is the fraction of network bandwidth reserved for each of the traffic streams. We actuate that via a minimum share of the bandwidth $Cu_i(k)$, $u_i = \{u_i | u_i \geq 0, \sum u_i = 1\}$. It is important to notice that u_i is only a minimum guaranteed share, which helps us avoiding wasting available bandwidth[3]. Bandwidth left unused by streams that did not have enough data to transmit can then be used by other streams, thus maximizing total bandwidth utilization. On the contrary, if the allocated bandwidth for a stream is insufficient to complete the transmission, the exceeding data is buffered.

Let $x_i(k)$ denote the buffer levels at time k for each traffic stream, i.e., the data that is ready to be sent at time k for each stream. For each of the streams, $\forall i \in I$,

[3] In other words, the traffic shaping is *work preserving*, meaning that shares are only enforced if there is enough data to transmit for each traffic stream.

we can then define the following linear integrator dynamics for the system:

$$x_i(k+1) = x_i(k) + \lambda_i(k) - Cu_i(k) - d_i(k). \tag{1}$$

In Equation (1), the disturbance terms $d_i(k)$ model actions that are not in direct relationship with the control signal, for example taking into account the situation in which buffers are emptied because there was not enough traffic in one of the other streams. The *actual sent traffic* for traffic stream i in the time interval $[k, k+1]$ is therefore

$$\mu_i(k) = Cu_i(k) + d_i(k). \tag{2}$$

We assume the buffer levels to be measurable[4]. Indeed, in real implementations, a measurement of the amount of data sent per traffic stream $\mu_i(k)$ is usually also available. Using that, measurements of the data arrival processes $\lambda_i(k)$ can be reconstructed as follows:

$$\lambda_i(k) = x_i(k+1) - x_i(k) + \mu_i(k). \tag{3}$$

We model the arrival processes as standard input disturbances. VM image traffic is modeled as impulses arriving at fixed intervals while for Application and Replication service traffic one of two possible disturbance models is used. In one case, traffic is assumed to be slowly varying, with the following state space representation:

$$\begin{aligned} z_i(k+1) &= z_i(k) + e_i(k) = Fz_i(k) + e_i(k) \\ \lambda_i(k) &= z_i(k) + v_i(k) = Gz_i(k) + v_i(k). \end{aligned} \tag{4}$$

In the second case, we extend the previous model with a local linear trend, with corresponding state space representation

$$\begin{aligned} z_i(k+1) &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} z_i(k) + e_i(k) = Fz_i(k) + e_i(k) \\ \lambda_i(k) &= \begin{pmatrix} 1 & 0 \end{pmatrix} z_i(k) + v_i(k) = Gz_i(k) + v_i(k), \end{aligned} \tag{5}$$

where $e_i(k) \sim N(0, \Sigma_e)$ and $v_i(k) \sim N(0, \Sigma_v)$. A Kalman filter is used to estimate the states of the disturbance models, which are then used as initial conditions for predicting future traffic by the MPC controller. Among the traffic streams, *VM image* stands out in the sense that the marginal benefit from allocating bandwidth to it is zero up until the point the transfer of a full image is completed. Bandwidth spent on servicing an image transfer without completing it is therefore essentially wasted. For this reason, we augment the system description with

[4] The lack of distinction between traffic stream buffers and QDisc buffers in Equation (1) is due to the fact that traffic streams are differentiated. This means that they are the only actor depositing packets in their target QDisc buffers. Therefore it is possible to aggregate data residing in each QDisc buffer and model it as one larger, measurable, buffer.

an integral state i_{vm} for the VM image buffer to incentivize the controller to finish image transfers. By setting $F_d = \text{diag}(F, F)$ we get a complete state space description of the system, augmented with corresponding disturbance and integral states, as

$$\begin{aligned} x(k+1) &= \begin{pmatrix} I & G_d & 0 \\ 0 & F_d & 0 \\ Z & 0 & 1 \end{pmatrix} x(k) - C \cdot \begin{pmatrix} I \\ 0 \\ Z \end{pmatrix} u(k) - \begin{pmatrix} I \\ 0 \\ Z \end{pmatrix} d(k) \\ G_d &= \begin{pmatrix} G^T & 0 & 0 \\ 0 & G^T & 0 \end{pmatrix}^T, \; Z = \begin{pmatrix} 0 & 0 & -1 \end{pmatrix}, \end{aligned} \tag{6}$$

with the state vector $x = \begin{pmatrix} x_a & x_r & x_{vm} & z_a & z_r & i_{vm} \end{pmatrix}$. We then use Equation (6) in our MPC controller design to predict the evolution of the system, assuming that the disturbances d_i are zero-mean and uncorrelated. In the design of the MPC controller, we use a standard quadratic cost function with penalties on buffer sizes and control signal variations $\Delta u_i(k) = u_i(k) - u_i(k-1)$

$$J = \sum_{k=1}^{H_p} \sum_{i \in I} \left(q_i \, x_i^2(k+1) + r_i \, \Delta u_i^2(k+1) \right) + q_n i_{vm}(k+1). \tag{7}$$

H_p is here the prediction horizon, while q_i and r_i are the penalties on buffer lengths and control signal variations, respectively and q_n the penalty on the integrator state. Neither buffer lengths nor control signals can be negative, so those properties enter as natural constraints to the problem. The controller formulation then takes the form

$$\begin{aligned} \text{minimize} \quad & \sum_{k=1}^{H_p} \sum_{i \in I} \left(q_i x_i^2(k+1) + r_i \Delta u_i^2(k+1) \right) + q_n i_{vm}(k+1) \\ \text{subject to} \quad & \text{Equation (6)}, \\ & 0 \le x_i \le \bar{x}_i, \\ & u_i \ge 0, \\ & \sum_{i \in I} u_i \le 1, \end{aligned} \tag{8}$$

where the upper limits $\bar{x}_i$ on buffer levels represent a tunable maximal amount of buffered data we can tolerate for each traffic type. The controller then selects $\Delta u_i(k+1)$ and therefore $u_i(k+1)$ in order to minimize the cost function. In turn, this allows us to trade data consistency (bandwidth share given to the *Replication service* and the *VM image* traffic) and performance (bandwidth assigned to the *Application*).

5. Evaluation

This section discusses the evaluation of the proposed solution and the comparison of the results obtained with the MPC controller, and comparing them with the other alternatives we introduced in Section 3. We tested the different strategies with a simulator and in many different scenarios, two of which are reported in the following. To compare the results, we have identified three metrics that summarize the behavior of the system and permit a comparison of the solutions. At the end of this section, we present some general conclusions that can be drawn from the experiments shown in this paper and from our experience with other scenarios.

5.1 Simulation Framework

In order to evaluate our proposed solution we have designed an event-based simulator using Python and SimPy[5]. The simulator is based on the system model detailed in Section 3. It includes implementations of a set of alternative traffic management solutions, the foundations of which are outlined in Section 3, together with the MPC controller introduced in section 4. The policies that complement our solution are the following:

- In *FIFO*, all traffic streams deposit packets in a shared QDisc buffer that is served by the network resource in a FIFO manner. When the buffer is full, the packet waits until further space in the buffer is available. This particular strategy mirrors a system's default behavior when no deliberate traffic shaping effort has been made by the system administrator.
- The *STATIC* solution implements a static bandwidth assignment, similar to the HTB filter approach described in Section 3. Each traffic stream is guaranteed a set share of the network bandwidth at all times. In our case we devote 30% of the bandwidth to the *Replication service* traffic, following the guidelines for DRBD. For the *VM image* traffic, we calculate the amount of bandwidth necessary to finish a session copy before the next is initiated. The remaining bandwidth is devoted to the application traffic.
- The *PRIO* strategy relies on priorities, assigned to each traffic stream. The priorities are fixed and assigned by the system administrator, based on a ranking of which traffic stream would benefit most from receiving prioritized access to the network. In the simulator, we have given the highest priority to the *Application* traffic in order for it to be minimally negatively impacted by the presence of *Replication service* traffic. The second highest priority is given to the *Replication service* traffic, with *VM image* traffic having the lowest priority. As previously described, each priority level has its own FIFO buffer in the QDisc that is served by the network resource only if higher prioritized buffers are empty.

[5] http://simpy.readthedocs.org

All these solutions are work preserving, thereby maximizing bandwidth utilization. The total traffic is therefore the same with all the solutions, the difference being how much of the shared resource is allocated to the different traffic types.

5.2 Performance metrics

In order to evaluate the behavior of each bandwidth allocation strategy, we perform simulations recording a set of relevant performance metrics. The set of metrics assesses the behavior of the traffic shaping solutions along different axes: the performance delivered to the application, the traffic needed for replication purposes and the data loss in case a disaster happens, it being data that has been buffered for replication but never sent out.

For *Application* and *Replication* streams, we observe the mean level i_μ and 95th percentile $i\_\lambda^{0.95}$, $\forall i \in \{App, Rep\}$ of buffered traffic over the entire experiment. The *VM image* transfer process is evaluated based on the average vm_σ_μ and 95th percentile vm_σ_λ of the time passed since the creation of the most recent *VM image* available at the secondary replica. This reflects the state to which a system could roll back in case a disaster happens. We also observe the mean vm_μ and 95th percentile vm_λ of the transfer times for the completed *VM image* transfers.

Together with statistics, we evaluate the application performance based on the *waiting time* spent in the system by each packet that belongs to the *Application* stream. To provide a measure of the effort required to restore a service following a disaster, at each point in time we take the last available *VM image* at the backup site and sum the amount of write operations that have been made since the timestamp associated with the image. This gives us an indication of the amount of data that should be recreated in case a replica should be fired up at a third site. We refer to this metric as the *Disaster Recovery Overhead (DRO)*. Finally, we measure the amount of data currently in the replication buffer. This data is considered lost at the moment of a failure (*Data loss*) since it has not been transferred to the replication site.

5.3 Experiment 1

Scenario: In this first experiment, we explore the behavior of the system in a 3-hour long experiment. In this experiment, the traffic mix changes slowly, producing periods in which the DC is overloaded and periods in which the network capacity is enough to serve the incoming traffic and the replication traffic. More specifically, the significant contributor to overload alternates between *Application* traffic and *Replication service* traffic. This traffic composition models the normal operation of a DC with which daily patterns (for example, a news website usually receives more visits during the lunch break).

The total available bandwidth to the system C is 100 Mbps, and the *Application* outputs on average 62.5 Mbps, while *Replication service* operations are made

Table 1. Statistics for experiment 1

Application	MPC	FIFO	STATIC	PRIO
App_μ [MB]	1616	3077	2076	183
$App\_\lambda^{0.95}$ [MB]	6988	8185	8100	1245
Replication				
Rep_μ [MB]	3442	2336	3168	2976
$Rep\_\lambda^{0.95}$ [MB]	6989	7944	9863	6807
VM image				
vm_σ_μ [sec]	1203	377	607	5388
vm_σ_λ [sec]	2275	650	931	10126
vm_μ [sec]	935	81	321	5211
vm_λ [sec]	1881	90	433	9992

on average at 32.5 Mbps. Every 10 minutes, a *VM image* copy is initiated with a fixed size of 375 MB, which corresponds to an average rate of 5 Mbps over a 10 minute period. The first plot in Figure 3 shows the *Application*, *Replication service* streams and the moments in which *VM images* are transmitted.

We configure STATIC and PRIO as outlined in Section 5.1. For the MPC, we set the state penalties to $(q_a,\ q_r,\ q_{vm},\ q_n) = (600,\ 250,\ 50,\ 1)$ and for control signal variations $(r_a,\ r_r,\ r_{vm}) = (10^6,\ 10^6,\ 10^6)$ with the prediction horizon $H_p = 30$, corresponding to 5 minutes as the sampling time is 10 seconds. Lastly, FIFO is configuration-free.

Results: Figure 3 illustrates the metrics for experiment 1, while Table 1 presents the resulting statistics of the experiment's outcome. The second plot in Figure 3 shows the waiting time for the read operations. In many time instants the DC is overloaded and there is not enough bandwidth to transmit the data belonging to all the streams. In this case, the time that each packet belonging to the *Application* stream waits in the system becomes larger. The FIFO strategy results in a significant increase of the waiting time, therefore reducing application performance. With FIFO, the system recovers only after a large enough period of under load is experienced. Conversely, when employing STATIC as traffic shaping mechanism the *Application*'s performance is quickly able to recover as soon as the arrival rate for the read traffic does not exceed the static capacity allotted to it. The PRIO is also penalized during overload — for example in the time interval $t = [1800, 4200]$. However, the *Application* penalty is only due to the overload generated by the *Application* traffic itself, which temporarily exceeds the capacity of the DC. The proportionality of the overload contributed by the *Applica-*

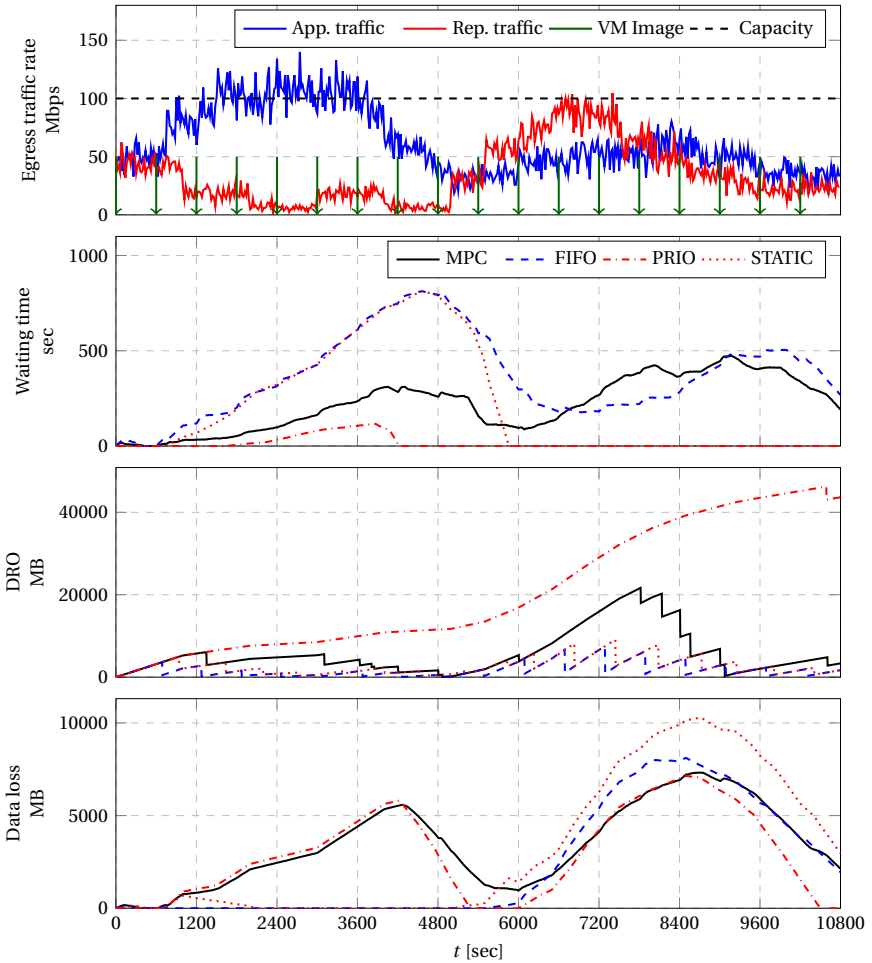

Figure 3. Results from experiment 1. Top figure shows the rates at which *Application*, *Replication* and *VM image* traffic arrives. Next figure shows the recorded waiting time before being sent for *Application* traffic. Thereafter the DRO is shown, while the potential data loss in case of a disaster is shown at the bottom.

tion is a common factor for both the STATIC and PRIO methods, although the waiting time is penalized to different extents. The MPC solution here provides a middle-ground with acceptable buffering proportional to the aggregate overload. The MPC is able to indiscriminately accommodate both types of overload (read arrival rate exceeding the capacity and total arrival rate exceeding the capacity) with a consistent level of *Application* performance.

The third plot in Figure 3 shows the DRO. As can be seen, FIFO and STATIC are able to accommodate the replication traffic and provide good replication performance. On the contrary, the priority given to the read traffic for PRIO comes at a significant DRO. Not only does the overhead fast exceed any other method but is divergent in this time-frame. The MPC solution is able to quickly recover also in terms of DRO. The last plot of Figure 3 shows the amount of data that is not recoverable in case a disaster happens at a specific time. When the read traffic is generating the overload conditions, only the PRIO and MPC method suffer from the possibility of data loss. On the contrary, when the write traffic is higher than the static channels allocated for FIFO and STATIC, the data loss of all the alternatives are comparable. In Table 1 it is possibile to see that while PRIO is the best in terms of *Application* performance, the MPC controller is the second best (App_μ, $App\_\lambda^{0.95}$), with STATIC and FIFO not being a good fit to handle the read traffic. While FIFO is on average good for *Replication* performance (Rep_μ), it is not consistently better (the 95th percentile $Rep\_\lambda^{0.95}$ is higher than with PRIO and MPC). The MPC solution is better at exploiting the trade-off between different traffic conditions, and is able to trade consistency for performance and viceversa. FIFO and STATIC the best at transfering the *VM images*, while PRIO is not capable of handling this part of the traffic (vm_σ_μ, vm_σ_λ, vm_μ, vm_λ).

5.4 Experiment 2

Scenario: In this second scenario we show long periods of stable traffic levels interpreaded with abrupt changes with resulting high and low network load. This could for example correspond to an application switching between different operating modes — *e.g.*, computing statistics and applying changes to the data. In contrast to the previous experiment, the overload in this scenario is less extreme. Here, the contribution to the contention is more uniform across the traffic types. The simulation experiment is run for total duration of two hours, and the total available bandwidth to the system C is 100 Mbps. *Application* traffic arrives at an average rate of 71 Mbps, *Replication service* operations to be replicated at 22 Mbps and *VM image* copies are again initiated every 10 minutes with an image size of 375 MB. The various policies are configured as done for the previous experiment. The STATIC shares are equal to the traffic rates, as if the operator could perfectly know the traffic composition. For the MPC we use the penalties $(q_a, q_r, q_{vm}, q_n) = (10^4, 2{\cdot}10^3, 3{\cdot}10^4, 1)$ and $(r_a, r_r, r_{vm}) = (5{\cdot}10^6, 5{\cdot}10^6, 5{\cdot}10^6)$, while the prediction horizon is again $H_p = 30$.

Results: The results from this experiment are summarised in Figure 4 and Table 2. The second plot in Figure 4 shows the waiting time for the read requests, while the third plot shows the DRO. Both FIFO and STATIC sacrifice the *Application's* performance in favour of significant *Replication service* traffic. As a result, both of these methods persistently achieve the lowest disaster recovery overhead. In this scenario the *Application* traffic generally does not exceed the capacity C.

Table 2. Statistics for experiment 2

Application	MPC	FIFO	STATIC	PRIO
App_μ [MB]	32	386	306	0.33
$App\_\lambda^{0.95}$ [MB]	100	1089	1083	1.03
Replication				
Rep_μ [MB]	222	23	18	86
$Rep\_\lambda^{0.95}$ [MB]	681	223	84	401
VM image				
vm_σ_μ [sec]	939	369	642	1231
vm_σ_λ [sec]	1720	643	975	3019
vm_μ [sec]	680	75	374	1003
vm_λ [sec]	1460	79	429	2748

As a result the PRIO method almost fully accommodates the *Application* across the entire observed time period. However, the PRIO method reaches a very high DRO, which recovers only when the *Application* traffic is significantly below the system capacity, with significant lag.

In the observed scenario, the MPC method is able to maintain a negligible *Application* performance degradation under periods of overload. The MPC solution's ability to balance the two objectives is made clear by the small sacrifice in *Application* performance for a significant reduction disaster recovery overhead. This specific ability to negligibly sacrifice *Application* performance also contributes to accelerating the recovery of the momentary disaster recovery overhead once the system return to an aggregate stable load, proportionally regardless to the composition of the load. Table 2 confirms these results.

5.5 Summary of findings

The primary objective of this evaluation is to determine the effectiveness of a dynamic solution to cope with the different type of traffic combinations and changes that happens in a real DC. The second aim of the experimental analysis is to determine how our MPC solution fits as a means to this end.

After trying a multitude of workloads, we can conclude that some of these workloads highlight features and weaknesses of all the different traffic scheduling solution that we have described. The FIFO solution has proven to be effective at accommodating all applications needs for streams when under loaded, and indiscriminately penalising when over loaded. Especially, in the scenarios we have rendered, a large portion of the traffic is *Application* traffic, which makes FIFO

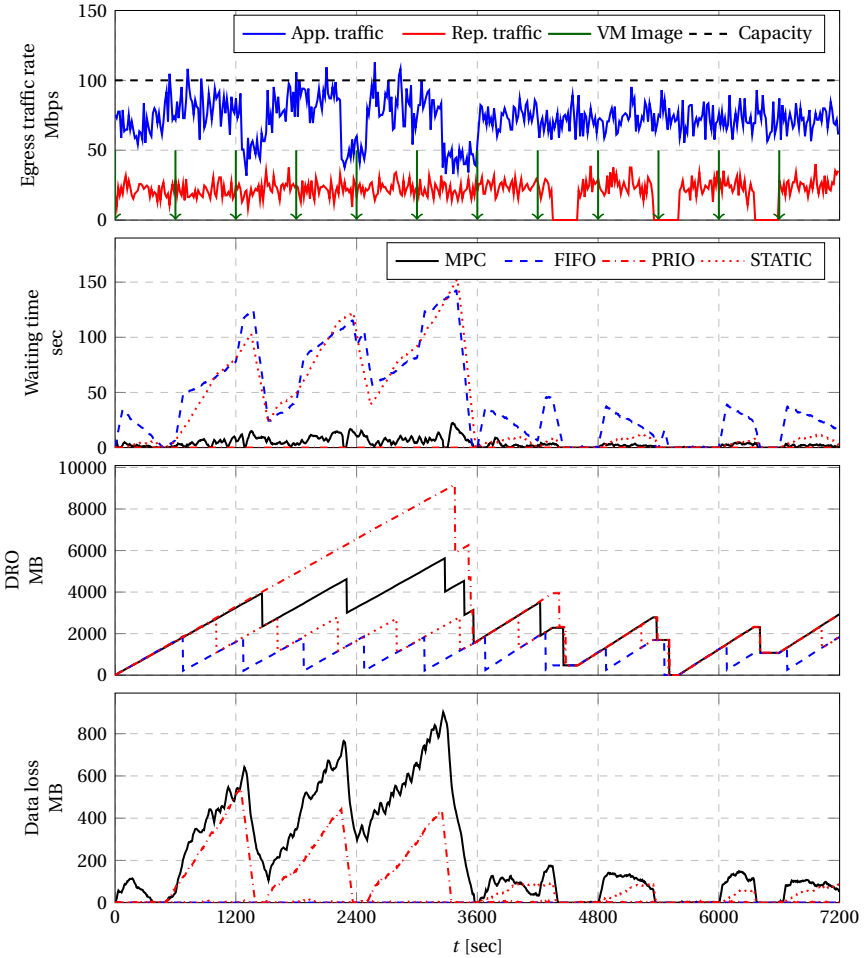

Figure 4. Results from experiment 2. Figures as in Figure 3.

unsuitable since the end users will suffer from buffering thus inhibiting the end-to-end performance of the *Application*. On the other hand, with FIFO, the replication traffic and the *VM image* traffic are able to indiscriminately gain access to the the shared resource and are therefore served with a reasonable and fair delay which is in line with what queuing theory tells us [Harchol-Balter, 2013]. Furthermore, the STATIC solution manages to isolate the *VM image* traffic and guarantee that the images are transferred timely, but is sensitive to any changes in the other traffic streams. Its evident inability to accommodate the individual objectives of the tenants make FIFO unsuitable for this system.

The PRIO solution is inherently the most successful in terms of accommodating *Application* performance, but fails at accommodating the other tenant's objectives. In most of the scenarios we have run, it performs well for the *Replication service* traffic but fails at containing and recovering the momentary disaster recovery overhead in a timely manner.

From the experiments above, we can conclude that our dynamic method is the best to achieve the most desirable balance between *Application* performance and disaster fault tolerance readiness in an intermittently overloaded system. Furthermore, the MPC method is able to capture the trade-off between delivering acceptable *Application* performance and accommodating the *Replication service*. The MPC solution is able to maintain the most consistent performance over periods with persistent overload, and is quickly able to indiscriminately recover once the system return to a stable state. Additionally, the MPC is able to persistently balance the two objectives according to the proportions specified in the objective function.

6. Conclusion and future work

In this paper we design an MPC controller to determine the amount of bandwidth to be allocated to different streams in a cloud computing infrastructure. Our investigation starts from the detection of an inherent trade-off between data consistency in case of disasters and performance delivered by applications to end users.

In fact, the outgoing bandwidth in the data center is used concurrently both to replicate the changes operated by the users in the secondary backup, targeting consistency, and to respond to the user requests, targeting performance. The available outgoing bandwidth is however limited. So, while one would want to serve the user requests timely, it is also important to ensure that the amount of data loss in case of a disaster is limited.

We have developed a dynamic solution for this problem, in the form of an MPC controller, that we compared to the static solutions that are currently the best practice. The result of our investigation is that a dynamic solution is more flexible and it is capable of exploiting the mentioned trade-off. Future work includes implementation and evaluation of our solution in a real environment.

References

Barron, J. (2003). *The blackout of 2003: the overview; power surge blacks out northeast, hitting cities in 8 states and canada; midday shutdowns disrupt millions.* URL: `http://www.nytimes.com/2003/08/15/nyregion/blackout-2003-overview-power-surge-blacks-northeast-hitting-cities-8-states.html`.

Cidon, A., R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer (2015). "Tiered replication: a cost-effective alternative to full cluster geo-replication". In: *2015 USENIX Annual Technical Conference (ATC)*, pp. 31–43.

Cully, B., G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield (2008). "Remus: high availability via asynchronous virtual machine replication". In: *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 161–174.

Devera, M. and D. Cohen (2002). "Htb linux queuing discipline". URL: `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`. Manual-user guide.

Dong, Y., W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan (2013). "Colo: coarse-grained lock-stepping virtual machines for non-stop service". In: *4th ACM Symposium on Cloud Computing (SoCC)*.

Dusia, A., Y. Yang, and M. Taufer (2015). "Network quality of service in docker containers". In: *IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 527–528.

Harchol-Balter, M. (2013). *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.

Ji, M., A. C. Veitch, and J. Wilkes (2003). "Seneca: remote mirroring done write". In: *2003 USENIX Annual Technical Conference (ATC)*, pp. 253–268.

Keeton, K., C. Santos, D. Beyer, J. Chase, and J. Wilkes (2004). "Designing for disasters". In: *3rd USENIX Conference on File and Storage Technologies (FAST)*, pp. 59–62.

Patterson, R. H., S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara (2002). "Snapmirror: file-system-based asynchronous mirroring for disaster recovery". In: *1st USENIX Conference on File and Storage Technologies (FAST)*.

R. Witty, D. S. (2001). *Disaster recovery plans and systems are essential*. URL: `https://www.gartner.com/doc/340749/disaster-recovery-plans-systems-essential`.

Reisner, P. and L. Ellenberg (2005). "Drbd v8 - Replicated storage with shared disk semantics". In: *12th International Linux System Technology Conference (Linux-Kongress)*.

Wood, T., E. Cecchet, K. K. Ramakrishnan, P. Shenoy, J. V.D.Merwe, and A. Venkataramani (2010). "Disaster recovery as a cloud service: economic benefits & deployment challenges". In: *2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*.

Wood, T., H. A. Lagar-Cavilla, K. K. Ramakrishnan, P. Shenoy, and J. V.D.Merwe (2011). "Pipecloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery". In: *2nd ACM Symposium on Cloud Computing (SoCC)*.