# 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*
*Department of Computing Science, Umeå University, Sweden
{cklein, francisco, elmroth}@cs.umu.se
†Department of Automatic Control, Lund University, Sweden
{alessandro, manfred, jonas, martina, karlerik}@control.lth.se

***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.**

## I. Introduction

Due to their ever-increasing scale and complexity, hardware failures in cloud computing infrastructures are the norm rather than the exception [1], [2]. This is why Internet-scale interactive applications – also called *services* – such as e-commerce websites, include replication early in their design [3]. 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 [3].

However, failures in cloud infrastructures are often correlated in time and space [4], [5]. 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 [6], 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* [7]. 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 [8], 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 [9]. 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 II).

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

Our contribution is three-fold:

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

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 [10].

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

## II. Background and Motivation

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

### A. Single Replica Brownout Services

To provide predictable performance in cloud services, the **brownout** paradigm [7] relies on a few, minimally intrusive code changes (e.g., 8 lines of code) and an online adaptation strategy that controls the response time of a single-replica based service. The service programmer builds a brownout-compliant cloud service breaking the service code into two distinct subsets: Some functions are marked as *mandatory*, while others as *optional*. For example, in an e-commerce website, retrieving the characteristics of a product from the database can be seen as mandatory – a user would not consider the response useful without this information – while obtaining comments and recommendations of similar products can be seen as optional – this information enhances the quality of experience of the user, but the response is useful without them.

For a brownout-compliant service, whenever a request is received, the mandatory part of the response is always computed, whereas the optional part of the response is produced only with a certain probability given by a control variable, called the **dimmer** value. Not executing the optional code reduces the computing capacity requirements of the service, but also degrades user experience. Clearly, the user would have a better experience seeing optional content, such as related products and comments from other users. However, in case of overload and transient failure 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 [7]. 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 [11]. 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 [9]. 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.

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

Let us briefly describe the design of the controller. Denoting the dimmer value with $\Theta$ 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 $\Theta$ 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 [7], it is possible to select the dimmer value to provide some guarantees on the service behavior. The selection is based on the adaptive proportional and integral controller

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

where the value $\tilde{\alpha}(k)$ is an estimate of the unknown parameter $\alpha(k)$ computed with a Recursive Least Square (RLS) filter. The error $e(k)$ is the difference measured at time index $k$ between the setpoint for the response time and its measured value, $p_1$ is a parameter of the controller, that allows to trade reactivity for robustness. A formal analysis of the guarantees provided by the controller and the effect of the value of $p_1$ can be found in [7].

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

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

where $\varepsilon$ 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.

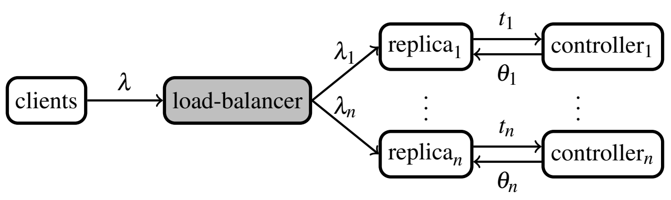


Fig. 1. Architecture of a brownout cloud service featuring multiple replicas.

Through empirical testing on two popular cloud applications, RUBiS [12] 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.

### *B. Multiple Replica Brownout-Compliant Services*

For fault tolerance, cloud services should feature multiple replicas. Fig. 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 [1].

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 $\lambda$. 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 $\Theta_i$ 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 $\Theta_i$. 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 $\lambda_i$ 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 $\Theta_i$ 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 [13] 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) [10], [13]. 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 [8]. 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 [13]. 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.

### *C. 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.

## III. Design and Implementation

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

### *A. Brownout-Compliant Load-Balancing Algorithms*

Here we discuss two brownout-compliant control-based load-balancing algorithms. Those are based on some ideas presented in [13], but with two major modifications. First, all the techniques proposed in [13] 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 [13] 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 $u_i$.

*1) 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 [14]. 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 $\Theta_i$, piggy-backed by each replica $i$ through a previous response, and on the queue lengths $q_i$, using the formula

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

where $\gamma \in (0,1)$ is a filtering constant, $\gamma_P$ and $\gamma_I$ 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 $\gamma$ and $\gamma_P$ are fixed to a selected value, increasing the integral gain $\gamma_I$ 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 $\Theta_i$, therefore greatly improving performance at the cost of a more aggressive control action. On the contrary, decreasing $\gamma_I$ would smooths the control action, possibly resulting in performance loss due to a slower reaction time. The choice of the integral gain allows to exploit the trade-off between performance and robustness. For the experiments we chose $\gamma_I = 5.0$.

*2) 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 $\Theta_i$ 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

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

where $\gamma_e$ 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.

## B. Implementation

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

## IV. 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.

## A. Experimental Setup

Experiments were conducted on a single physical machine equipped with two AMD Opteron$^{\text{TM}}$ 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 [15]. Each replica is deployed with all its tiers – web server and database server – inside its own Virtual Machine (VM), as is commonly done in practice [16], e.g., using a LAMP stack [17]. 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 [12], an eBay-like e-commerce prototype, that is widely-used for cloud benchmarking [18]–[24]. RUBiS was already brownout-compliant, thanks to a previous contribution [7] 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 [25]–[27] and, in case of RUBiS, would only require an engineering effort. The load-balancer, i.e., `lighttpd` extended with our brownout-aware algorithms, was deployed inside the privileged VM in Xen, i.e., Dom0, pinned to a dedicated core.

To generate the workload, we had to choose between three system models: open, closed or partly-open [28]. 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

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

[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

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 [9].

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

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

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

To qualify the resilience of the service, we chose two metrics that measure how well the service is performing in hiding failures, or, otherwise put, how strongly the user is affected by failures. The **timeout rate** represents the number of requests per second that were not served by the service within 4 seconds, due to overload. In production, a request that timed out will make a user unhappy. She may leave the service to join other competitors, thus incurring long-term losses to the service provider. The **optional content ratio** represents the percentage of requests served with optional content. Serving a request with optional content, such as recommendations of similar products, may increase the service provider's revenue by 50% [8]. 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.

## *B. 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.

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

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

Even worse, when enough replicas are restored to make capacity sufficient, the non-brownout service still does not

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

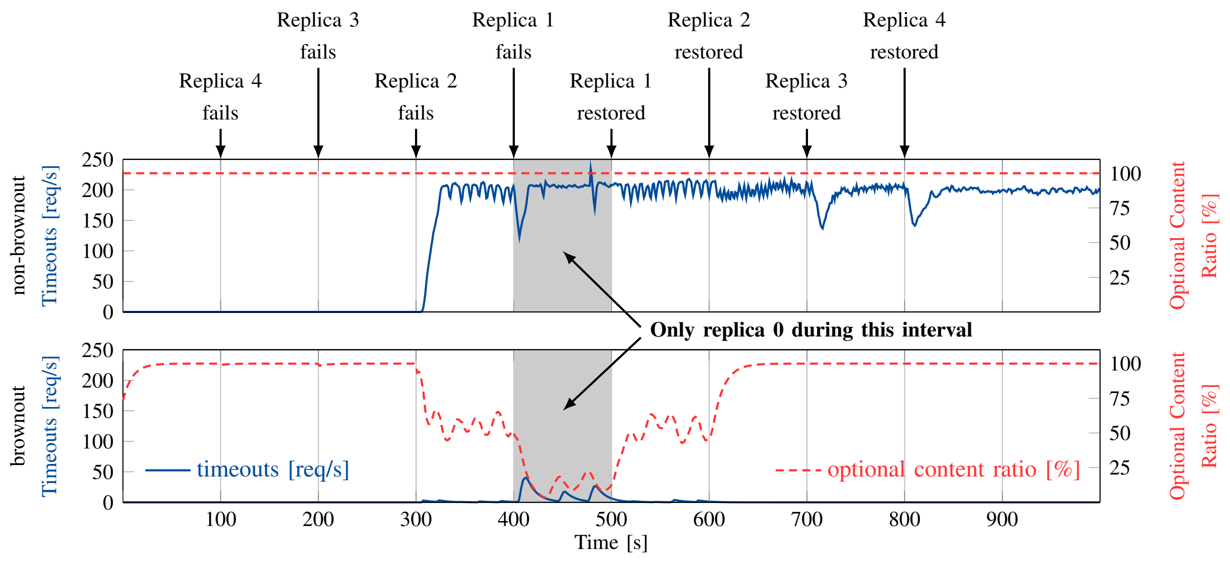


Fig. 2. Experimental results comparing resilience without and with brownout. Configuration: 5 replicas, each having 4 cores.

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

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

TABLE I
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% |

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

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

## C. 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 [13]. 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 [29] 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 [30]. As another example, previous elasticity (auto-scaling) decisions may have resulted in heterogeneously sized replicas [31]. 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**.

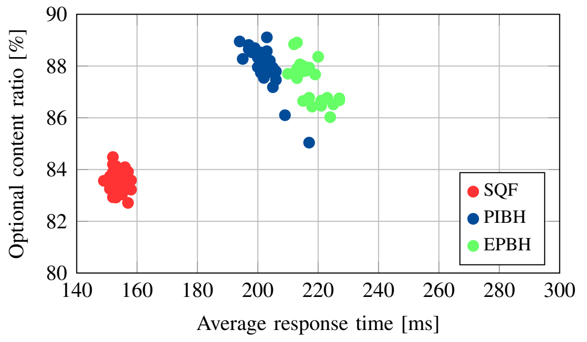
*1) "2×1+3×8 cores" Scenario:* The first scenario consists of a constant request rate of 400 requests per second. The service consists of 5 replicas, two of which are allocated 1 core, while the other three are allocated 8 cores. This scenario leaves the service with insufficient capacity to serve all 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.

Fig. 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 II. 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 ratio, at the expense of somewhat higher response times. Slightly increasing the average response time (Fig. 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 (Fig. 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 Fig. 3, all three algorithms obtain a similar distribution of response times. In Table III 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.

*2) "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 Fig. 4, analogously to the previous scenario, while pair-wise comparison of algorithms is presented in Table IV. PIBH and EPBH outperform SQF with respect to optional content ratio by 5.17% and 3.13%, respectively.

Again, this is achieved without interfering with the self-adaptation of the replicas: 95th percentile response times are distributed similarly for all three algorithms close to the target. This is also proven by the paired t-test presented in Table V, 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



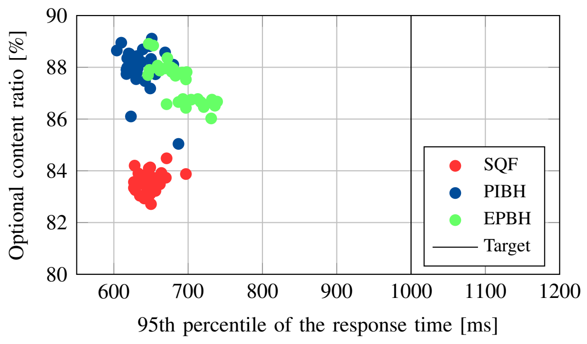
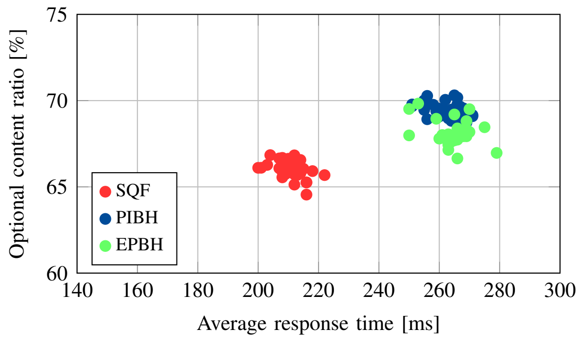


Fig. 3. Comparison of SQF and brownout-aware load-balancing algorithms when two replicas have 1 core and three replicas have 8 cores.



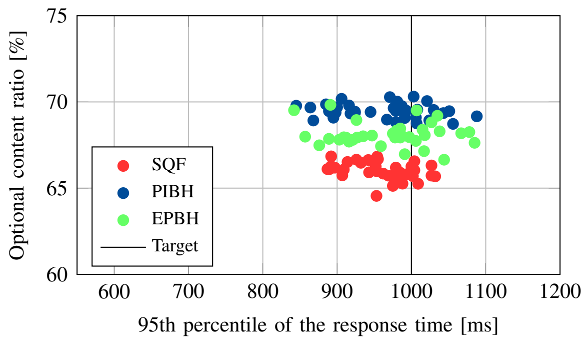


Fig. 4. Comparison of SQF and brownout-aware load-balancing algorithms when three replicas have 1 core and two replicas have 8 cores.

TABLE II
IMPROVEMENT IN AMOUNT OF OPTIONAL CONTENT SERVED, AFTER 120000 REQUESTS (SUMMARY OF FIG. 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 III
IMPROVEMENT IN AMOUNT OF 95TH PERCENTILE OF THE RESPONSE TIME (SUMMARY OF FIG. 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** ($p < 10^{-9}$) |

TABLE IV
IMPROVEMENT IN AMOUNT OF OPTIONAL CONTENT SERVED, AFTER 120000 REQUESTS (SUMMARY OF FIG. 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** ($p < 10^{-15}$) |

TABLE V
IMPROVEMENT IN AMOUNT OF 95TH PERCENTILE OF THE RESPONSE TIME (SUMMARY OF FIG. 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** ($p = 0.2265$) |

quantity is increased, but on average still lower than the setpoint (set to 1 second). The same holds for the average response time, which is slightly increased with respect to the previous scenario.

## *D. 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 [10]. 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.

## V. 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* [32], 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 [33], 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 [34]. 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 [2], 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 [25]–[27]. 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 [1], [3]. 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 [35] or price of energy [36]. 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 [10]. Our previous simulation results [13] 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.

## VI. 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 [37], that would allow to autonomously manage the number of replicas, together with the traffic routing, to obtain a cloud service that is both resilient and cost-effective. Finally, some control parameters were chosen empirically based on the many tests we have conducted. Ongoing work will quality the robustness of the system given the chosen parameters in a more systematic way and for a larger scenario space.

## Acknowledgment

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

## References

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

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

[3] J. Hamilton, "On designing and deploying internet-scale services," in *LISA*, USENIX, 2007, 18:1–18:12.

[4] M. Gallet *et al.*, "A model for space-correlated failures in large-scale distributed systems," in *Euro-Par*, 2010. DOI: 10.1007/978-3-642-15277-1_10.

[5] N. Yigitbasi *et al.*, "Analysis and modeling of time-correlated failures in large-scale distributed systems," in *GRID*, 2010. DOI: 10.1109/GRID.2010.5697961.

[6] E. Chuah *et al.*, "Linking resource usage anomalies with system failures from cluster log data," in *SRDS*, 2013. DOI: 10.1109/SRDS.2013.20.

[7] C. Klein *et al.*, "Brownout: building more robust cloud applications," in *ICSE*, 2014. DOI: 10.1145/2568225.2568227.

[8] D. Fleder *et al.*, "Recommender systems and their effects on consumers," in *Electronic Commerce*, 2010. DOI: 10.1145/1807342.1807378.

[9] F. F.-H. Nah, "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour and Information Technology*, vol. 23, no. 3, 2004.

[10] V. Gupta *et al.*, "Analysis of join-the-shortest-queue routing for web server farms," *Perform. Eval.*, vol. 64, no. 9-12, 2007. DOI: 10.1016/j.peva.2007.06.012.

[11] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, 2007. DOI: 10.1145/1323293.1294281.

[12] (2014). Rice university bidding system, [Online]. Available: http://rubis.ow2.org.

[13] J. Dürango *et al.*, "Control-theoretical load-balancing for cloud applications with brownout," in *Conference on Decision and Control (CDC)*, IEEE, 2014.

[14] I. D. Landau *et al.*, *Digital control systems: design, identification and implementation*. Springer, 2006.

[15] P. Barham *et al.*, "Xen and the art of virtualization," in *SOSP*, ACM, 2003. DOI: 10.1145/945445.945462.

[16] K. Sripanidkulchai *et al.*, "Are clouds ready for large distributed applications?" *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, 2010. DOI: 10.1145/1773912.1773918.

[17] (2013). Tutorial: installing a LAMP web server, [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html.

[18] Z. Gong *et al.*, "PRESS: predictive elastic resource scaling for cloud systems," in *CNSM*, IEEE, 2010. DOI: 10.1109/CNSM.2010.5691343.

[19] Z. Shen *et al.*, "CloudScale: elastic resource scaling for multi-tenant cloud systems," in *SoCC*, ACM, 2011. DOI: 10.1145/2038916.2038921.

[20] W. Zheng *et al.*, "JustRunIt: experiment-based management of virtualized data centers," in *ATC*, USENIX, 2009, pp. 18–28.

[21] C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," in *NSDI*, USENIX, 2005, pp. 71–84.

[22] N. Vasić *et al.*, "DejaVu: accelerating resource allocation in virtualized environments," in *ASPLOS*, ACM, 2012. DOI: 10.1145/2189750.2151021.

[23] C. Stewart *et al.*, "Exploiting nonstationarity for performance prediction," in *EuroSys*, ACM, 2007. DOI: 10.1145/1272998.1273002.

[24] Y. Chen *et al.*, "SLA decomposition: translating service level objectives to system level thresholds," in *ICAC*, IEEE, 2007. DOI: 10.1109/ICAC.2007.36.

[25] N. L. Diegues and P. Romano, "Bumper: sheltering transactions from conflicts," in *SRDS*, IEEE, 2013. DOI: 10.1109/SRDS.2013.27.

[26] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *SoCC*, ACM, 2010. DOI: 10.1145/1807128.1807152.

[27] M. S. Ardekani *et al.*, "Non-monotonic snapshot isolation: scalable and strong consistency for geo-replicated transactional systems," in *SRDS*, IEEE, 2013. DOI: 10.1109/SRDS.2013.25.

[28] B. Schroeder *et al.*, "Open versus closed: a cautionary tale," in *NSDI*, USENIX, 2006.

[29] B. Welch, "The generalization of 'student's' problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1-2, 1947. DOI: 10.1093/biomet/34.1-2.28.

[30] L. Tomás and J. Tordsson, "Improving cloud infrastructure utilization through overbooking," in *CAC*, ACM, 2013. DOI: 10.1145/2494621.2494627.

[31] M. Sedaghat *et al.*, "A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling," in *CAC*, ACM, 2013. DOI: 10.1145/2494621.2494628.

[32] Y. Lin and S. S. Kulkarni, "Automated multi-graceful degradation: a case study," in *SRDS*, 2013. DOI: 10.1109/SRDS.2013.17.

[33] T. Do *et al.*, "Limplock: understanding the impact of limpware on scale-out cloud systems," in *SoCC*, 2013. DOI: 10.1145/2523616.2523627.

[34] N. Fallahi *et al.*, "Rigorous performance evaluation of self-stabilization using probabilistic model checking," in *SRDS*, 2013. DOI: 10.1109/SRDS.2013.24.

[35] M. Lin *et al.*, "Online algorithms for geographical load balancing," in *IGCC*, IEEE, 2012. DOI: 10.1109/IGCC.2012.6322266.

[36] J. Doyle *et al.*, "Stratus: load balancing the cloud for carbon emissions control," *TCC*, vol. 1, no. 1, 2013. DOI: 10.1109/TCC.2013.4.

[37] A. Ali-Eldin *et al.*, "An adaptive hybrid elasticity controller for cloud infrastructures," in *NOMS*, IEEE, 2012. DOI: 10.1109/NOMS.2012.6211900.