Brownout: Building More Robust Cloud Applications

Cristian Klein
Umeå University, Sweden
cristian.klein@cs.umu.se

Martina Maggio
Lund University, Sweden
martina@control.lth.se

Karl-Erik Årzén
Lund University, Sweden
karlerik@control.lth.se

Francisco Hernández-Rodriguez
Umeå University, Sweden
francisco@cs.umu.se

ABSTRACT

Self-adaptation is a first class concern for cloud applications, which should be able to withstand diverse runtime changes. Variations are simultaneously happening both at the cloud infrastructure level — for example hardware failures — and at the user workload level — flash crowds. However, robustly withstanding extreme variability, requires costly hardware over-provisioning.

In this paper, we introduce a self-adaptation programming paradigm called *brownout*. Using this paradigm, applications can be designed to robustly withstand unpredictable runtime variations, without over-provisioning. The paradigm is based on optional code that can be dynamically deactivated through decisions based on control theory.

We modified two popular web application prototypes — RUBiS and RUBBoS — with less than 170 lines of code, to make them *brownout*-compliant. Experiments show that *brownout* self-adaptation dramatically improves the ability to withstand flash-crowds and hardware failures.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Design—*Methodologies*; D.1.3 [**Software**]: Programming Techniques—*Distributed programming*

General Terms

Design, Experimentation, Theory, Performance

Keywords

Adaptive Software, Control Theory, Brownout, Cloud

1. INTRODUCTION

Many modern software applications are developed for the cloud [13]. In fact, cloud computing is expected to be one of the first 5 technologies that will drive the future economy [36]. In addition to traditional requirements, cloud applications have dynamic loads and variable number of users, therefore dynamic resource capacity requirements [56]. Moreover, they also need to be designed to robustly handle unexpected events: Unexpected peaks — also called flash crowds — may increase the volume of requests by up to 5 times [9]. Similarly, unexpected hardware failures in data centers are the norm rather than an exception [31, 50]. Also, unexpected performance degradations may arise due to workload consolidation and the resulting interference among co-located applications [47].

These phenomena are well-known, therefore, software is readily designed and deployed to cope with them. For example, techniques such as elasticity [35], replication, dynamic binding and dynamic load balancing allow to overcome unexpected events as long as resource capacity is sufficient [7, 32]. However, given the large magnitude and the relatively short duration of such unexpected events, it is often economically unfeasible or too costly to provision enough capacity. As a result, the application can **saturate**, i.e., it can become unable to serve users in a timely manner. Some users may experience high latencies, while others may not receive any service at all. Hence, the application owner may lose customers and profits. We argue that it is more profitable to downgrade user experience, thus serving a larger amount of clients.

To allow applications to more robustly handle unexpected events and avoid saturation, we propose a new programming paradigm called *brownout*. Our work borrows on the concept of brownout in electrical grids. Brownouts are intentional voltage drops often used to prevent blackouts through load reduction in case of emergency. In such a situation, incandescent light bulbs dim — emitting less light and consuming less power — hence originating the term.

We define a cloud application as **brownout**-compliant if it can gradually downgrade user experience to avoid saturation. For example, online shops usually offer end-users recommendations of similar products they might be interested in. No doubt, recommender engines greatly increase the user experience, which translates to higher owner revenue. In fact, a study found an increase of 50% on song sales when a group of users where exposed to recommendations [26]. However, due to their sophistication, such engines are highly demanding on computing resources [41]. The developer can specify that the execution of the recommender engine is optional. By selectively activating or deactivating optional components, the application's capacity requirements can be controlled at the expense of end-user experience, without compromising the functional requirements.

To lower the maintenance effort, brownouts should be automatically triggered. This would enable cloud applications to rapidly and robustly avoid saturation due to unexpected environmental changes, lowering the burden on human operators. In other words, the application should be self-adaptive [17]. Designing *brownout*-compliant applications brings the design of the runtime behavior of the application itself into the software design [5].

Contributions.

In this article we introduce a paradigm to design and develop cloud applications based on the concept of brownouts. Brownout-

ICSE '14, June 01 - 07 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/06...$15.00.
http://dx.doi.org/10.1145/2568225.2568227

compliant applications can change their resource capacity requirements and automatically adjust to changing conditions of its environment. Hence, we add a new mechanism [17] to enable design and development of self-adaptive applications. First, we discuss a model that captures the behavior of a typical cloud application with optional computation that can be activated or deactivated at runtime. Second, we synthesize a control-theoretical solution to automatically decide when to activate those optional features. Control theory allows us to provide specific guarantees on desirable properties such as user-perceived latency. This directly translates into withstanding unexpected events more robustly.

Our paper offers the following contributions.

- It proposes a model for cloud applications with optional components. Applications are extended with a dynamic parameter, the *dimmer* Θ, that monotonically affects both the end-user experience and the computing capacity required by the application (Section 2).
- It synthesizes a controller to automatically adapt the *dimmer* to cope with the incoming workload and the available resources, with formal convergence proof (Section 3).
- It shows the applicability of the brownout paradigm, extending two popular cloud benchmarks, RUBiS [57] and RUBBoS [12], with few non-intrusive changes (Section 4).
- It presents experimental results showing the behavior of the self-adaptive applications in comparison to their non-adaptive counterpart, as suggested in [10, 71]. Brownout-compliant applications more robustly withstand unexpected events like peak loads and resource shortages (Section 5).

The results show that using the brownout paradigm, applications can support more users or run on less resources, while maximizing user experience. Hence, our proposition enables cloud infrastructures to more robustly deal with unexpected peaks or unexpected failures, requiring less spare capacity. To foster further research on application brownouts and to make our results reproducible, we released all source code[1].

2. APPLICATION MODEL

In this section, we define the theoretical foundations of the *brownout* paradigm, discussing the methodology of making applications *brownout*-compliant.

Cloud applications serve multiple users through the Internet. Their computations are generally separated into independent and stateless user requests that, after being processed, provide responses [24]. An essential requirement of these applications is that responses should be produced in a time-sensitive way, otherwise unsatisfied users would abandon the service. Indeed, a study on web-user behavior found a tolerable waiting-time between 2 and 4 seconds [51]. With this model in mind, brownout can be added to applications in three steps.

Step 1. The application designer needs to identify which part of the response can be considered optional. In fact, it is good engineering practice to decompose the act of serving a request into different software components, each dealing with a different part of the response. Some of these components produce data that are necessary to satisfy the user's request, while other provide accessory information that merely improve user experience.

In a brownout-compliant application, software components are isolated to make it possible to activate or deactivate the optional computation per request. Being able to run optional computations is desirable, as they would improve end-user experience. However, in case of an unexpected event, it is preferable to deactivate optional computations, instead of saturating the application due to insufficient hardware resources and providing a response after the tolerable waiting time of the user has expired. Deciding activation of optional components for each request allows us to make more fine-grained trade-offs. For example, instead of completely deactivating optional components, the application may serve every second user with the optional part, thus avoiding saturations, but still improving the experience of some users.

Step 2. The designer needs to provide a knob to control how often the optional computations are executed. Applications export a dynamically changeable runtime parameter, a **dimmer** Θ, which monotonically affects both the average quality of the user experience and the amount of resources that the application requires. This allows a specialized component, called the controller, to monitor the application and adjust the application's behavior as needed (see Step 3). More formally, the number of times that these optional computations are executed between time k and $k+1$ (and thus the amount of resources required by the application) is proportional to Θ^k, the dimmer's value during that time interval[2]. To adhere to the stateless request model, for each request, the optional component can be activated depending on the outcome of a single Bernoulli trial with success probability Θ^k.

To clarify the concepts just introduced, we sketch an e-commerce website as an example of a brownout-compliant application. In the e-commerce website, we consider the visualization of a page containing one specific product as one request. The optional part of the response consists in displaying recommendations of similar products. For each request, besides retrieving the product information, the application runs the recommender engine with a probability Θ. Increasing Θ increases the number of times recommendations are displayed, thus improving end-user experience, but also the resource requirements of the application. In the end, the resource requirements will be roughly proportional to Θ.

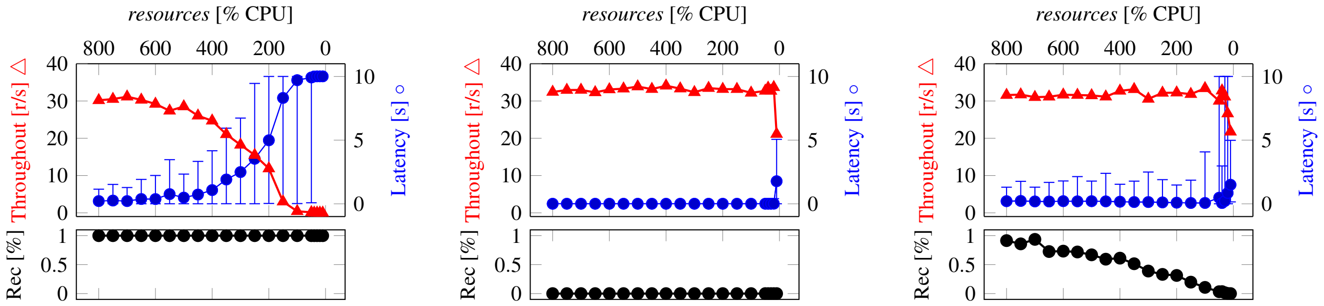
To show that the brownout paradigm can handle different performance measures and different application-engineering concepts, we propose another example. A music-streaming service, such as Spotify [43], serves two types of audio files: songs that the user chooses to listen to and ads. Since they are simultaneously requested by a large number of users, songs can often be served in a peer-to-peer fashion, without consuming the service owner's bandwidth. In contrast, ads are personalized for each user and premium users do not have to listen to them at all. Ads, therefore, need to be served from the servers operated by the owner. In this case, we consider the streaming of a song as a single request, which may be preceded by an ad. Since it is the heart of the business model, the service owner would like to serve as many ads as possible. However, it might be better to make serving ads optional and stop serving them in case of insufficient bandwidth than to damage the service's reputation due to interrupted song streaming. Thus, the associated performance is the delivered streaming bandwidth, directly affecting user satisfaction, while the dimmer Θ is the probability of serving an ad before streaming a song, affecting the service's profit.

Step 3. For reduced burden on the human operator, the application should be self-adaptive. A new component, called the **controller**, is added to achieve this. Its goal is to adjust the dimmer exported during the second step as a function of the current performance, e.g., response time, to avoid saturation.

Let us illustrate the interest of self-adaptation through numerical examples obtained with our extended version of the RUBiS e-

[1] `https://github.com/cristiklein/brownout`

[2] In the entire paper, we use superscripts to indicate time indexes.



(a) with all recommendations (b) with no recommendations (c) self-adaptive

Figure 1: Stress-testing RUBiS extended with our recommender engine. The number of users is kept constant and the amount of CPU allocated is reduced. Throughput reduction and response time increase show the point where the application is saturated.

commerce website. Let us assume that a constant number of users is accessing the cloud application. Due to various unexpected phenomena — such as cooling failure, hardware failure, performance interference, etc. — the amount of resources allocated to the application may be arbitrarily reduced. Since a usual application serves recommendations to all users, it would become saturated when the allocated CPU is lower than 400% (i.e., 4 cores). Effectively, the application cannot respond to user requests as quickly as they are issued, therefore, some users experience high latencies, while others perceive the applications as unavailable (Fig. 1a). The application **less robustly** withstands unexpected capacity reductions.

To deal with the lack of capacity and restore the application's responsiveness, a system administrator could decide to completely disable recommendations (Fig. 1b). This way, the application would be responsive as long as it has at least 20% CPU. Effectively, by disabling recommendations, the application can withstand a further capacity reduction of 20 times (i.e., from 400% to 20%).

Nevertheless, adopting such a manual solution has several deficiencies. First, it requires an administrator to constantly monitor the application and react. Second, no users would receive recommendations, even if there would be extra capacity to serve, for example, every second user with recommendations. Therefore, manually deactivating or activating optional code is not a viable solution.

Avoiding manual intervention, a self-adaptive application would dynamically adjust the dimmer as available capacity allows (Fig. 1c). Effectively, as the amount of CPU allocated to the application is reduced, the application would transition from serving recommendations to all users, to serving recommendations to some users, to serving no recommendations at all. The application **more robustly** withstands unexpected resource capacity reductions. Self-adaptation would require some "headroom", in our case the amount of CPU needs to be above 40%, instead of 20% without recommendations. However, we believe the headroom is small when compared to the self-adaptation benefits.

We obtained similar figures with an increase in the number of users and constant resources, as happens with flash crowds — sudden increases of popularity of the application when linked from a high-profile website. With no recommendations, RUBiS saturates with 20 times more users than if recommendations are always enabled, whereas a self-adaptive approach allows a 10 fold increase in users. Hence, the brownout paradigm enables cloud applications to better withstand unexpected events, either load increase or hardware capacity reductions, by gradually reducing user experience.

From an engineering point-of-view, the brownout paradigm also encourages modularity and separation of concerns. Existing applications only need to be augmented with the choice of *which* code is optional and *how* to deactivate it, while a separate controller can take care of *when* to execute the optional components. In the next section, we discuss how to build a controller that automatically selects the dimmer for a vast class of applications, providing formal guarantees on its behavior.

3. CONTROLLER DESIGN

Cloud applications usually run in virtualized environments, where a hypervisor multiplexes hardware resources among multiple Virtual Machines (VMs). This makes avoiding saturation more challenging due to the inherent performance unreliability of the underlying VM. Moreover, CPU utilization cannot reliably measure used capacity: At low utilization, the VM is given the impression that it runs isolated, without being informed of the amount of CPU that is allocated to a different VM. This only becomes noticeable at higher utilizations, reported as "steal-time", i.e., the time that the VM would have had something to execute, but the hypervisor decided to run a different VM instead [23]. This unreliability introduces the need for a different indicator to detect saturation conditions, such as response time. However, the relationship between response-time and saturation is non-linear, since many applications behave like queues [2, 54]. Therefore, off-the-shelf controllers, like PIDs, should be carefully tuned and coupled with online corrections. In this section we present the synthesis of a controller for average and maximum latency and formally prove its limits. We employ terminology and notations used in control theory [45].

The first controller keeps the average response time around a given setpoint. Using a very primitive, yet useful, model we assume that the average response time of the web application, measured at regular time intervals, follows the equation

$$t^{k+1} = \alpha^k \cdot \Theta^k + \delta t^k \qquad (1)$$

i.e., the average response time t^{k+1} of all the requests that are served between time index k and time instant $k+1$ depends on a time varying unknown parameter α^k and can have some disturbance δt^k that is a priori unmeasurable. α^k takes into account how the dimmer Θ selection affects the response time, while δt^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. Our controller design should aim for canceling the disturbance δt^k and selecting the value of Θ^k so that the average response time would be equal to our setpoint value.

As a first step of the design, we assume that we know α^k and its value is constant and equal to α. We will later substitute an estimation of its current value in the controller equation, to make sure that the behavior of the closed loop system is the desired one. The Z-transform of the system that we want to control before the feedback loop is closed, called the **plant**, described by Eq. (1) is

$$z \cdot T(z) = \alpha \cdot \Theta(z) + \Delta T(z) \tag{2}$$

where z^{-1} is the unit delay operator, $T(z)$ is the Z-transform of the time series t^k, $\Theta(z)$ relates to Θ^k and $\Delta T(z)$ transforms δt^k. We cannot control the disturbance $\Delta T(z)$, therefore, we are only interested in the transfer function from the input (the dimmer) to the output (the measured average response time), which is

$$P(z) = \frac{T(z)}{\Theta(z)} = \frac{\alpha}{z}. \tag{3}$$

Every closed loop system composed by one controller and a plant has the generic transfer function

$$G(z) = \frac{C(z) \cdot P(z)}{1 + C(z) \cdot P(z)} = \frac{Y(z)}{R(z)} \tag{4}$$

where $C(z)$ is the transfer function from the error to the control signal. In this case, the error is the difference between the setpoint and the measured value and the control signal is the dimmer value in the next time interval. $P(z)$ is the plant transfer function [45] — in our case Eq. (3). $Y(z)$ and $R(z)$ are respectively the output and the input of the closed loop system, in our case the measured and the desired average response time. The function $G(z)$ represents the response of the controlled system with the feedback loop closed.

The next step in controller design consists in deriving an equation for $C(z)$. One possible strategy is to choose $C(z)$ so that some properties on $G(z)$, the response of the controlled system, are satisfied — in control terms, to select the "shape" of the response. For example, we want the steady state gain of $G(z)$ in Eq. (4) to be one, since we want the output to be equal to the setpoint. Also, we want to introduce a stable pole in the closed loop system, to control the speed of the response — in order for the system to be stable the pole should lay within the unit circle, in order to also avoid oscillations its value should be between zero and one. Assuming that we want to introduce the stable pole in p_1, our desired closed loop transfer function looks like

$$G(z) = \frac{C(z) \cdot P(z)}{1 + C(z) \cdot P(z)} = \frac{1 - p_1}{z - p_1} \tag{5}$$

and substituting the plant transfer function of Eq. (3) into Eq. (5) we can derive the expression $C(z) = \frac{(1-p_1) \cdot z}{\alpha(z-1)}$ for the controller, which turns to be a PI controller with specific constant values. By applying the inverse Z transform on $C(z)$, we obtain

$$\Theta^{k+1} = \Theta^k + \frac{1 - p_1}{\alpha} \cdot e^{k+1} \tag{6}$$

where e^{k+1} is the difference measured at time $k+1$ between the setpoint for the response time and its measured value. This equation can be used to implement a controller that selects the dimmer parameter. We also add anti-windup to the controller. The choice of the pole p_1 depends on the type of behavior that we want to enforce for the closed-loop system, as explained later in this section.

During the control synthesis phase, we assumed α to be constant. However, we know that its value changes over time (due to performance interference) and we should take into account those variations. We should therefore provide an estimation of its current value α^k as $\tilde{\alpha}^k$ to be used in the controller. We can use many methods to estimate it online, while the application is running. The most simple is to take past measurements, compute the average response time t^{k+1}, pretend the disturbance δt^k is negligible and compute $\tilde{\alpha}^k$ based on Eq. (1). Once a first estimation is available, it is also possible to assign a weight to new data points and choose

$$\tilde{\alpha}^{k+1} = (1 - \mu) \cdot \tilde{\alpha}^k + \mu \cdot \frac{t^{k+1}}{\Theta^k} \tag{7}$$

where μ is a discount factor that defines how trustworthy the new observations are.

The same methodology can be used also to control maximum response time, that has become the focus of recent research [20]. The above designed controller can easily be adapted to this purpose, by using $e^{k+1} = \bar{l}_{\max} - l^k_{\max}$ in Eq. (6), where $\bar{l}_{\max}$ is the desired maximum response time and $l^k_{\max}$ is the maximum response time measured from time index k to $k+1$.

In what follows, we use the controller for maximum latency. It is to be expected that latencies can have high variations, but their maximum should be kept close to the setpoint of the controller.

Control-theoretical guarantees.

Control theory allows us to provide some formal guarantees on the system. Our main aim is to close a loop around a cloud application, constraining the application to have a behavior that is as predictable as possible. Without any feedback strategy, the application can have transient behaviors, depending on the input that it receives. For example, when the number of users suddenly increases, latencies can raise due to saturation. We would like to enforce robustness on the application behavior, no matter how much environmental variations the application is subject to. In control terms, the uncontrolled variations that the system is exposed to are *disturbances* and our aim is to follow the setpoint, *rejecting* them.

The controller will not be able to reject all types of disturbances. For example, when even setting the dimmer to zero results in a too high latency it means that the control system cannot achieve the desired value. However, if the goal is feasible, i.e., if the controller can set a dimmer value whose operating conditions fulfill the requirements, it will find it due to its stability property. To enforce stability — since the closed loop system has the form given by Eq. (5) — we should simply make sure that the pole p_1 belongs to the open interval $(-1, 1)$. To avoid oscillations, the pole should also be positive [45].

We now analyze how the pole position can compensate the undesired effects of introducing an estimator for α^k in the control algorithm. The controller acts based on an estimation of the effect of its action, $\tilde{\alpha}^k$ in Eq. (7). If this estimation is incorrect, the controller acts based on some false assumption. However, the presence of the feedback loop helps in detecting those errors and reacting to them in a satisfactory way.

The value given to the pole p_1 can be use to trade off responsiveness (how fast the controller reacts to disturbances, maybe not correctly estimated) and safety (how sure the controller is that the action taken will not damage the system). The closer p_1 is to one, the slower the system responds, but the better it rejects measurement noise or other disturbances. Effectively, the controller will only make small corrections at every iteration. In contrast, values of p_1 close to zero will make the system respond quickly, but also be more sensitive to disturbances, making large corrections, that risk being based on transient disturbances instead of long-term trends. Some values for p_1 can be suggested, depending on the reliability of the measurements and the variability of the incoming requests. However, selecting a value for p_1 is best done based on empirical testing as shown in Section 5.2.

To complete the trade off analysis we show the entity of incorrect estimation that each possible value given to p_1 is able to withstand. Assume we estimate α^k as $\tilde{\alpha}^k$ but the real values is $\tilde{\alpha}^k \cdot \Delta\alpha^k$. This

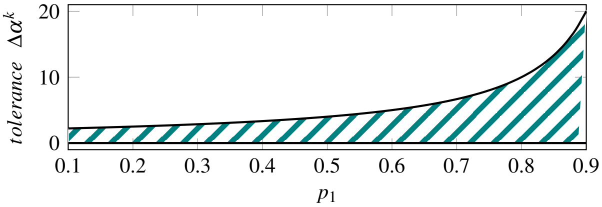


Figure 2: Multiplicative tolerance on the estimation error.

multiplicative perturbation is often used to quantify how wrong an estimation can be. If the system tolerates a $\Delta\alpha$ equal to 10, it means that although the estimated value might be 10 times smaller or larger than the real one, the system will converge anyway due to the presence of the feedback loop.

We test what is the maximum perturbation that our system is able to cope with. In other words, we want to find the values of $\Delta\alpha^k$ for which our plant is still stable. The plant transfer function $P(z)$ is $\frac{\alpha}{z}$, therefore it becomes $P_{mod}(z) = \frac{\bar{\alpha}\cdot\Delta\alpha}{z}$. The controller transfer function is $C(z) = \frac{(1-p_1)\cdot z}{\bar{\alpha}(z-1)}$. The closed loop transfer function $G_{mod}(z)$ of Eq. (5) becomes therefore

$$G_{mod} = \frac{C(z)\cdot P_{mod}(z)}{1+C(z)\cdot P_{mod}(z)} = \frac{(1-p_1)\cdot\Delta\alpha}{z+\Delta\alpha(1-p_1)-1} \qquad (8)$$

which is stable only if the two denominator poles are inside the unit circle. Since the gain is one, it still hold that, if the system is stable, the setpoint will be reached. The stability conditions derived by setting the denominator roots within $(-1,1)$ is $0 < \Delta\alpha^k < \frac{2}{1-p_1}$, which means exactly that choosing the value of the pole p_1 defines how safely the controller acts with respect to model perturbations.

Fig. 2 plots the allowed perturbation. The highlighted region is the safety zone. Setting the pole to 0.9 means that the estimation can be 20 times wrong, while setting the pole to 0.1 means that the tolerated estimation error is only twice or half of the real value.

In conclusion, there is a fundamental trade off between the controller reactivity and the safety with respect to perturbations that the controller can withstand. This trade off can be exploited carefully choosing the pole p_1. In our experimental results we demonstrate what this means in practice, with statistical analysis, exploring the range of possible choices for p_1 and their effects.

4. IMPLEMENTATION

To demonstrate the easiness of applying the brownout paradigm to existing cloud applications, we extended two well-known cloud benchmarks: RUBiS [57] and RUBBoS [12].

RUBiS is an extensively used benchmark that implements an auction website, similar to eBay. It has been widely used in cloud research, e.g. in [16, 29, 61, 63, 64, 66, 74]. We built a brownout-compliant RUBiS version, extending the PHP implementation and in particular the `ViewItem.php` page. We show how we applied the three steps presented in Section 2: selecting an optional component, adding the dimmer, closing the feedback loop.

With respect to selecting an optional component, the existing RUBiS implementation seems a fairly minimalistic auction website. Therefore, to make our evaluation more realistic, instead of selecting an optional component from the existing code, we decided to extend RUBiS with a simple recommender engine, that works as follows: When the user views an item j, the engine retrieves the set of users U_j that bid on the same item in the past. Then, the engine composes the set of recommended items R_j, retrieving other items that the users in U_j have bid on. The items in R_j are ordered by popularity, i.e., the number of bids on them, and the top 5 are returned. While the described engine is not sophisticated, it does serve as a reasonable example of an optional component that a cloud application may enable or disable at runtime. Clearly, such a recommender engine adds a great value to the user experience, thus increasing the owner's revenue. However, it is also resource hungry, as already discussed in Section 2 and Fig. 1. Nevertheless, in our extension, the recommender engine is well isolated, which allows us to easily enable and disable it per-request.

Modification	RUBiS	RUBBoS
Recommender	37	22
Dimmer	3	6
Reporting response time to controller	5	5
Controller	120	120
Total	*165*	*153*

Table 1: Effort in logical Source Lines of Code (SLOC) to apply brownout to two popular cloud applications.

Having selected the recommender engine as the optional component, we can proceed by adding an externally-modifiable parameter to control its activation: the dimmer. Since PHP scripts are executed independently for each request, adding a central coordinator to decide which requests are to be served with recommendations and which not may lead to contention, thus reduced scalability. Therefore, we chose the dimmer Θ to represent the per-request probability that the recommender engine is activated. By working with a probabilistic instead of deterministic behavior, we enable each invocation to take an independent decision. Each invocation of a script reads the dimmer value from a file, called the *dimmer file*, then a single Bernoulli trial is performed to decide whether recommendations are served or not: The script generates a random number $r \in [0,1]$ and if $r < \Theta$, recommendations are displayed, otherwise not. The operating system caches this small dimmer file, therefore, exporting the dimmer using this procedure is done with low overhead and is minimally intrusive for the codebase (Table 1).

Finally, we need to close the feedback loop to avoid overload. We chose as performance criterion the user-perceived latency as it seems to have a great influence on web user satisfaction [65] and is highly suitable to predict saturation (Fig. 1). To this end, the beginning and the end time of each "view item" request are recorded and, by subtracting the two, the response time can be measured. While this quantity is slightly different than the user-perceived latency — due to network latencies, context switches and other external factors — it should be reasonably close. Each invocation of the view-item page sends the measured response-time to a well-known local UDP port, on which the controller listens and stores these measurements. The controller's algorithm is periodically activated to decide on a new value of the dimmer based on Eq. (6). This value is atomically written to the dimmer file, using the `rename` system call, and is then used by the PHP scripts during the next control interval. Adding latency reporting and implementing the controller takes little effort as can be seen in Table 1.

RUBBoS is a bulletin-board prototype website modeled after Slashdot and has also been used as a benchmark in cloud computing research [19, 37, 46, 69].

Adding brownout-compliance to RUBBoS can be done similarly to RUBiS described above, focusing on the "view story" page. However, concerning the first step of our methodology, optional code choice, RUBBoS offers more flexibility. Indeed, we have identified two parts that can be considered optional. First, the existing code features a comment section that can easily be disabled. While the comments section is an essential part of a bulletin-board, users are better served without it, than not at all, in case of overload. Second, we extended RUBBoS with a recommender engine, that suggests other stories that might be related or interesting for the reader, based on common commentators.

The activation of the two optional components, the comment section and the recommender engine, can be linked to the value of the dimmer Θ in several ways. In our implementation, for each invocation, the view story script stochastically decides how to serve the page, using a cascade scheme as follows. A first Bernoulli trial is performed with probability Θ for success. In case of a successful outcome, comments are served and a second Bernoulli trial is performed with the same probability. In case the second trail succeeds, then recommendations are also served alongside the comments. As a result, the probability of serving a page with comments is Θ, while for serving a page both with comments and recommendations the probability is Θ^2.

The feedback loop is identical to the RUBiS one. We reused the controller written for RUBiS and implemented the same response time reporting mechanism through local UDP. As with RUBiS, the code changes were minimal as shown in Table 1.

Experience with two popular cloud applications showed that brownout can, in general, be added to existing applications with minimal intrusion and little effort, once the optional components have been identified. As with most self-adaptation techniques, brownout is a cross-cutting concern [17]. Therefore, large projects would benefit from using aspect-oriented programming [39] to clearly separate brownout-compliance code from core concerns.

In the next section, we do extensive experimentation, to further show the benefits of our paradigm, and to provide an in-depth evaluation of the behavior of the resulting self-adaptive applications.

5. EVALUATION

In this section, we report results obtained using real-life experiments to show the potential of the brownout paradigm in comparison to a non-adaptive approach and to test the behavior of the self-adaptive application under different conditions. In what follows we first describe the experimental setup, then we do fine-grained analysis on the time-series results of a limited number of runs and, finally, we test the system statistically under a variety of conditions.

Experimental Setup.

Experiments were conducted on a single physical machine equipped with two AMD Opteron$^{\text{TM}}$ 6272 processors[3] and 56 GB of memory. To simulate a typical cloud environment and also to dynamically change the resource allocation, we decided to deploy each application, with all its tiers, inside its own VM, as is commonly done in practice [62], e.g., using a LAMP stack [58]. We use Xen [6] as hypervisor to get fine-grained resource allocations to VMs [44]. Initial experiments revealed that CPU is the major bottleneck, both for RUBiS and RUBBoS. Therefore, each VM was configured with a static amount of memory, 4 GB, and a variable number of virtual CPUs depending on the experiment. Allocating 800% CPU means that the application had exclusive access to 8 cores of the physical machine, while 50% signifies accessing to a single core of the physical machine, for half of the time. Combined multiplexing of the physical cores, both in space and in time, is common in today's virtualized data-centers [30].

To emulate the users' behavior, we have found the clients provided by RUBiS and RUBBoS insufficient for our needs. Specifically, they do not allow to change the number of concurrent users and their behavior at run-time. Moreover, they report statistics for the whole experiment and do not export the time series data, preventing us from observing the application's behavior during transient phases. Last, these tools cannot measure the number of requests that have been served with recommendations or comments, which represents the quality of the user-experience.

We therefore developed a custom tool, `httpmon`, to emulate web users. Its behavior is similar both to the tools provided by RUBiS and RUBBoS, and to the TPC-W benchmark specification [27]. Among others, it allows to dynamically select a **think-time** and a **number of users** and maintains a number of client threads equal to the number of users. Each client thread runs an infinite loop, which waits for a random time and then issues a request for an item or a story. The random waiting time is chosen from an exponential distribution, whose rate parameter is given by the reciprocal of the think-time. A master thread collects information from the client threads and periodically prints statistics for the previously elapsed second of execution. In particular, it records the maximum **user-perceived latency** — which is the time elapsed from sending the first byte of the HTTP request to receiving the last byte of the HTTP response — and the **ratio of recommendations** or **comments** — the number of requests that have been served executing the optional code for recommendations or comments divided by the total number of requests. Note that, due to the stochastic nature of our implementation (see Section 4), this may be slightly different from the dimmer Θ, which is the output of the controller. During all our experiments, `httpmon` was executed on a dedicated core, to reduce its influence on the application under test.

5.1 Time-series analysis

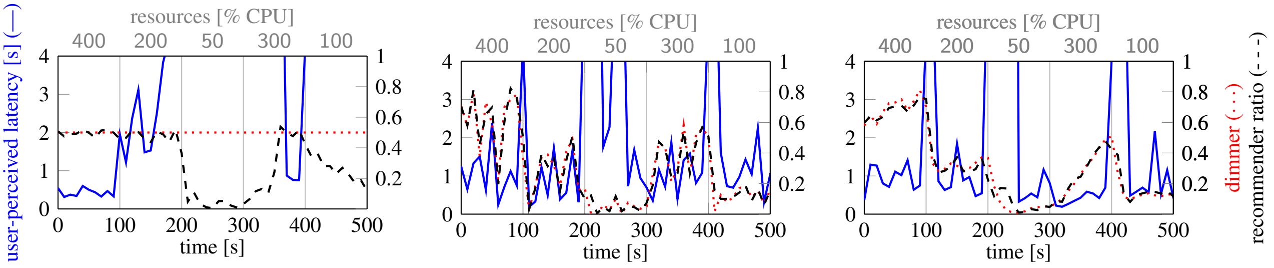
Thoroughly testing a system should be done under a variety of conditions, applying statistical analysis on the result. However, statistical testing alone may hide details about the behavior of the system in transient phases. Therefore, we first show a few, selected experiments in the form of time-series.

We report three sets of experiments, each focusing on a different time-varying aspect. First, we vary the resources that the application can use. Second, we vary the application load — the number of connected users. As a third experiment, we vary both these quantities together to emulate a real execution environment. Since previous research suggests that unexpected peaks vary considerably in nature [9], we manually chose values for load and resources that exposed the application to extreme conditions.

The following figures, presenting each a single experiment, are structured as follows. The bottom x-axis represents the time elapsed since the beginning of the experiment. Every experiment consists of 5 intervals, each of them lasting 100 seconds. The experimental parameter that is changed for every interval and its value are reported on the top x-axis. Three different metrics are plotted. First, the maximum user-perceived *latency* is shown in continuous blue lines and its y-axis is depicted on the left side of the plot. Second, the right y-axis hosts two different curves related to the user experience. The first one, the *dimmer*, is the output of the controller and is shown in dotted red lines. The second one, the *comments* or *recommendation ratio*, depicted in dashed black lines, is the average ratio of pages served with optional content (number of pages served with optional content over total number of pages served). To improve the readability of the graphs, the values are aggregated over 10-second intervals.

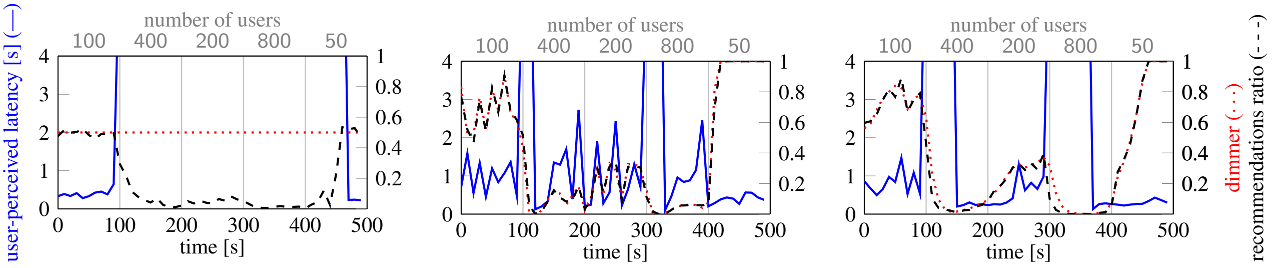
Ideally, the controller should maximize the dimmer while keeping the latency close to the setpoint. Also, the recommendation ratio should closely follow the dimmer.

[3] 2100 MHz, 16 cores per processor, no hyper-threading.



(a) non-adaptive, $\Theta = 0.5$ (b) self-adaptive, $p_1 = 0.5$ (c) self-adaptive, $p_1 = 0.9$

Figure 3: RUBiS behavior in a non-adaptive configuration and two self-adaptive configurations, varying the resource allocation.



(a) non-adaptive, $\Theta = 0.5$ (b) self-adaptive, $p_1 = 0.5$ (c) self-adaptive, $p_1 = 0.9$

Figure 4: RUBiS behavior in a non-adaptive configuration and two self-adaptive configurations, varying the number of users.

Constant Load and Variable Resources.

In this set of experiments, we keep the load constant — 100 concurrent users with a think-time of 3 seconds — and vary the amount of resources allocated to RUBiS. In a cloud data-center, this situation arises when a physical machine is over-subscribed and co-located VMs suddenly requires more resources. In fact, when multiple VMs share the same machine, performance interference may occur [52]. This is either cause by the hypervisor's decision to time-share a CPU core, also called "steal time" [23], or due to the congestion of a hardware resource, such as memory bandwidth or CPU cache. Also, CPU throttling due to cooling failures or load redistribution due to the failure of a different physical machine may cause similar VM capacity reductions.

The controller was configured with a control period of 1 second to allow a quick reaction, a target latency of 1 second to allow a safety distance to the tolerable waiting time of 2 seconds recommended in [51], a discount factor $\mu = 1$ and a sliding measure window of 5 seconds. This means that the controller's input is the error between the desired value of 1 second and the maximum latency measured over the last 5 seconds. First, we test the behavior of the non-adaptive system when no controller is present. Emulating the non-brownout-compliant version of the application can simply be done by forcing the output of the controller to maximum. However, we found that the resulting system behaves poorly, therefore, for fairer comparison, we set the dimmer to 0.5, which means that requests are served with recommendations only half of the time. Second, we compare these results to the case of brownout-compliant application, when the controller's pole p_1 is set to 0.5 and 0.9.

Figure 3 plots the results of our test. Figure 3a shows that the non-adaptive application performs quite well in the first interval, up to 100 seconds, when resources are abundant as the CPU allocation is set to 400%. Indeed, the latency is below 2 seconds and the recommendation ratio is closely following the dimmer, 0.5. However, during the next interval, when resources are slightly insufficient as the amount of allocated CPU is halved, the latency starts increasing because the application is unable to serve requests fast enough and saturates. In the next time interval, when even fewer resources are available, the system becomes unresponsive and some users experience huge latencies, up to 10 seconds[4]. The ratio of recommendations is very low, as requests that would potentially receive recommendations are abandoned by the client due to timeouts. In the next interval, more resources are allocated to the web application as its CPU is increased to 300%. The application "catches up" with serving previous requests and the latency decreases. However, this process takes 60 seconds, during which the application seems unresponsive. In the last interval, resources are insufficient and the application becomes again unresponsive. The results show that if resource capacity fluctuates and temporarily becomes insufficient, a non-adaptive approach may result in users experiencing unpredictable latencies.

Figure 3b plots the results of the self-adaptive application with the controller configured with the pole $p_1 = 0.5$. In contrast to the non-adaptive system, the self-adaptive application is perceived as more predictable from the user's perspective. Effectively, it managed to maintain the latency below 2 seconds whenever possible, despite a factor 8 reduction of resources, from a CPU of 400% to one of 50%. This is due to the dimmer adjustment that follows the available resource capacity. Furthermore, the ratio of recommendations closely follows the dimmer.

We discuss here the few deviations from the desired behavior, where the latency increases above the tolerable waiting time. The highest deviations occur as a result of an overload condition, when the CPU allocation is reduced, around time instant 100, 200 and 400. This is in accordance with theory, since the controller needs some time to measure the new latencies and correspondingly select the new dimmer value. Nevertheless, the system quickly recovers from such conditions, in less than 20 seconds. Around time instant 50, 240 and 480, the controller seems to be too aggressive. It tends

[4] We limit plots to 4 seconds to ease comparison among scenarios.

to increase the dimmer quickly, violating therefore the 2 seconds tolerable latency.

Let us now study how the self-adaptive application behaves when the controller is tuned for more stability. Figure 3c plots the results with the same controller configured with $p_1 = 0.9$. As predicted by theory, it reacts at a slower pace, with small adjustments at every iteration. Its output seems more steady and it generally does a better job at keeping the latency around the setpoint of 1 second. By using this controller, the likelihood of having latencies above the tolerable waiting time is decreased. However, this configuration also takes more time to recover from overload conditions. Compared to the previous configuration, it required twice as much time to react to the resource reduction at time instant 200. Also, during recovery, the recommendation ratio differs slightly from the dimmer's value. This happens because the responses arriving at the client have a high latency and were actually started at a time when the dimmer was higher. However, considering that the resources were reduced instantaneously by a factor of 4, the slower recovery is unlikely to be a problem in a production environment.

Summarizing, adding brownout self-adaptivity to a cloud application may considerably improve its flexibility with respect to resource allocation. Effectively, the application behaves more robustly and can withstand large reduction in resource allocation, proportional to the resource requirements of the optional components.

Constant Resources and Variable Load.

In this second set of experiments, we keep the resources constant, setting the CPU allocation to 400%, and vary the number of users accessing brownout-compliant RUBiS. In a real data-center, this situation may happen due to flash crowds — sudden increase in popularity when the page is linked from another high-profile website. However, it can also be the result of load redistribution due to a failing replica or denial-of-service attacks. The controller is configured identically to the previous set of experiments.

Let us now discuss the results. Figure 4a shows the results of the non-adaptive version of RUBiS, when the system cannot keep up with the load increase. Even after the number of users is significantly decreased, such as at 400 seconds, the application requires a significant time to recover, up to 62 seconds.

In contrast, Figure 4b and Figure 4c shows the results with the self-adaptive version of RUBiS. Despite an 8-fold increase in the number of users from 100 to 800, the application is more responsive, adjusting the dimmer to adapt to the load increase. Regarding the adaptation time, in the worst interval, when the number of users was spontaneously increased by a factor of 4 at time 300, the controllers required respectively 22 seconds and 66 seconds when $p_1 = 0.5$ and 0.9. As in the previous experiment, the controller with a pole of 0.5 is more aggressive, quickly increasing the dimmer and risking latencies above the tolerable level. In contrast, setting the pole to 0.9 produces a more conservative controller, which does smaller adjustments of the dimmer's value. In any case, the brownout-compliant cloud application is more robust and avoids saturation when the number of users increases.

Variable Load and Resources.

To reproduce a realistic setup, we studied how a brownout-compliant application behaves when both the available resource capacity and the number of users are varying. We present the results of an experiment conducted with RUBBoS, incidentally showing another brownout-compliant application with two optional components, comments and recommendations. However, to improve graph readability, we only present the ratio of requests served with comments. Concerning controller configuration, we present results with the pole $p_1 = 0.9$ and, to further show that the controller behaves as theoretically designed, we chose to reduce the target latency (setpoint) to 0.5 seconds.

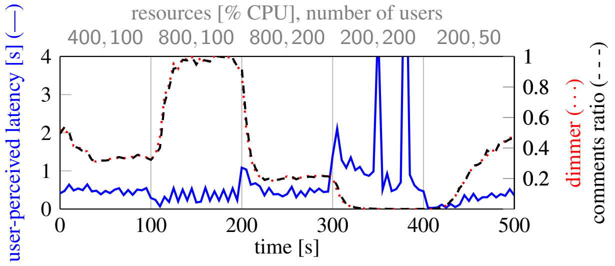


Figure 5: RUBBoS (brownout-compliant, $p_1 = 0.9$, setpoint 0.5 seconds) behavior varying both resources and users.

Figure 5 shows the results of the experiment. As can be observed, except for the fourth interval, the controller successfully manages to keep the maximum latency around 0.5 seconds. In the fourth interval, the dimmer is kept as close as possible to zero to serve the maximum number of requests in a reasonable time. In general, the dimmer is increased when the conditions allow for it, such as during the second and fifth interval, and decreased when resource capacity is insufficient or the load is too high, during the remaining intervals.

The time-series results show that the self-adaptive applications behaves as intended. The controller adapts the dimmer both to the available capacity and number of users as expected, and keeps the perceived latencies close to the setpoint. Moreover, the advantages that the brownout paradigm brings to a previously non-adaptive applications can clearly be observed from the results.

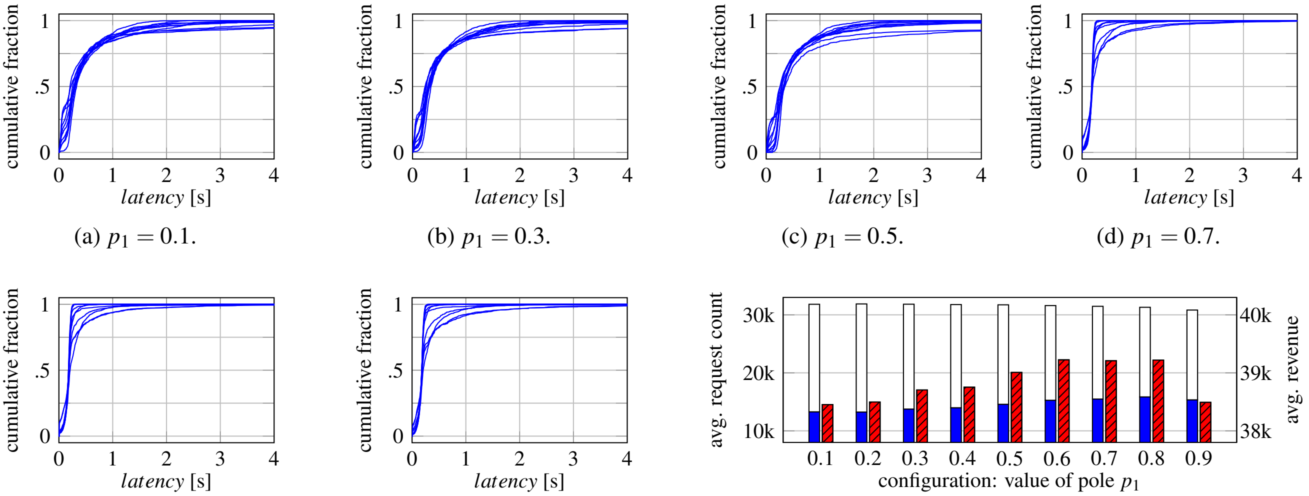
The experiments opened up two questions. First, we have shown that the pole allows to choose between a more aggressive and a more conservative controller. Which one is "better", i.e., which one serves more requests of any kind and which one serves more requests with optional components enabled? Second, we have chosen to compare the self-adaptive approach with a non-adaptive one with the dimmer $\Theta = 0.5$. Is it possible that other static dimmer values compare more favorably? In what follows we show experimental results that answer these two questions.

5.2 Statistical analysis

In this section, we present statistical results to show that our controller is able to tackle a variety of scenarios with invariant benefits in the cloud application behavior.

We focus our experiments here on unreliable resource capacity. Our intuition is that flash-crowds are somewhat avoidable, for example, by gradually rolling out an application to an increasing number of users, as Google did when launching GMail or Facebook when launching Graph Search. In contrast, for the foreseeable future, hardware is increasingly unreliable as its scale and complexity is constantly increasing, therefore, any cloud provider eventually has to deal with unreliable resource capacity. Moreover, performance of the cloud applications may be degraded due to colocation [47]. Since both the stress-test (Fig. 1) and time-series results (Figs. 3 and 4) suggest that the system behaves similarly in responding to an increasing number of users as to a reduction in resources, we believe these results are representative also for the flash-crowd scenario.

The experimental setup is as follows. Each particular application configuration, whether non-adaptive or self-adaptive, is subjected to 10 distinct test-cases. Each test-case consist of 10 time intervals,



(a) $p_1 = 0.1$.

(b) $p_1 = 0.3$.

(c) $p_1 = 0.5$.

(d) $p_1 = 0.7$.

(e) $p_1 = 0.8$.

(f) $p_1 = 0.9$.

(g) Summary over all self-adaptive configurations.

Figure 6: Statistical results for self-adaptive system. Figs. 6a to 6f show empirical cumulative distribution of latencies for each configuration. Each line represents the results of a test-case. Fig. 6g shows, for each configuration, over all test-cases, average served requests (bar with no fill), how many of them were on average served with recommendations (solid, blue bar) and average revenue (red, patterned bar).

each having a duration of 100 seconds, but a different amount of CPU allocated to the application. The first four test-cases allocated CPU with small uniform random deviation of 50% around 100%, 200%, 400% and 600%. The next four test-cases allocate CPU uniform randomly between 50% and 800%. The final two test-cases validate the system in extreme cases, in which the amount of available resources alternates between 50% and 800% CPU, starting with the lower and higher amount, respectively. We present results with RUBiS, with the controller configured similarly as in the previous section, except for a setpoint of 0.5 seconds.

For any particular configuration of the application, we are interested in the following metrics: the user-perceived latency represented as an empirical cumulative distribution function, the total number of requests served and the total number of requests served with optional content, such as recommendations. To provide a single metric and ease comparison, we compute the **revenue** of the application owner. Each served request values 1 monetary unit and each served recommendation adds 0.5 monetary units. This revenue model is based on a study that found that recommendations increased sales by 50% [26].

Let us first focus on testing the various self-adaptive configurations. Fig. 6 show the results for various pole configurations ranging from 0.1 (more aggressive) to 0.9 (less aggressive). The first take-away point is that every configuration does a reasonable job in keeping the latencies below the setpoint, therefore, we gain confidence that our controllers are adjusting their output correctly. Analyzing the results more in detail, more aggressive controllers serve slightly more requests than conservative ones, however, they serve fewer recommendations and perform poorer in maintaining low latencies (Figs. 6a to 6f). When combining the two dimensions by looking at revenue (Fig. 6g), values for p_1 between 0.6 and 0.8 maximize application owner income. In the end, we determine that the pole $p_1 = 0.8$ is our best configuration, since it maximizes average revenue over the test-cases, while at the same time keeping low latencies.

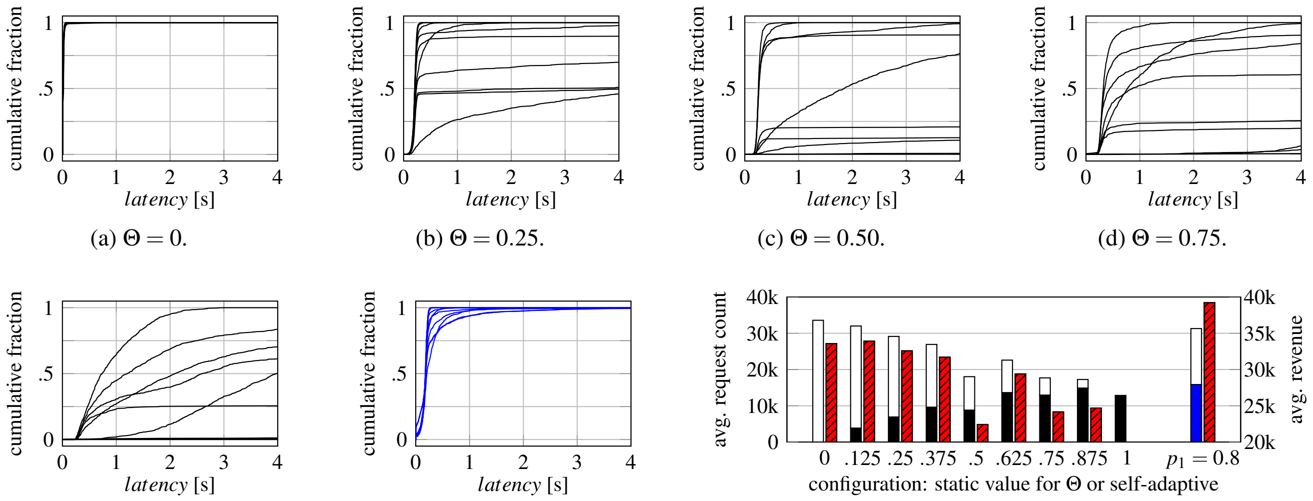
Having found the best configurations for the self-adaptive application, let us compare it to the non-adaptive version. Fig. 7 shows the results for various statically chosen dimmer values with the same resource availability patterns. The results of the best brownout-compliant configuration is replotted, for the reader's convenience in Figs. 7f and 7g. As expected, low dimmer values maintain low latencies, but deliver no recommendations. High dimmer values risk saturating the application, which leads to timeouts and, as a result, reduces the number of requests served. In contrast, the self-adaptive approach serves recommendations when capacity allows and disables them otherwise. As a result, when it comes to owner revenues, self-adaptation manages to outperform all static configurations (Fig. 7g).

To sum up, the results show that adding brownout compliance to cloud applications increases their robustness in case of flash-crowds, unexpected hardware failures or unexpected performance interference. This directly translates into increased revenue for the application owner and, hence, increased profits.

6. RELATED WORK

Self-adaptation is playing a key role in the development of software systems [17, 38, 42] and control theory has proved to be a useful tool to introduce adaptation in such systems [11, 21, 25, 70]. Although many attempts have been made to apply control theory to computing systems [53], the research is still in a preliminary stage and the achievable benefits are yet to be clearly defined [34, 76].

We were inspired by the idea that there might be multiple code alternatives for the same functionality [3, 22] and not all the code that a software application is executing is necessary, and some of its code might be skipped when necessary [49]. A similar concept has been proposed in the web context. Degrading the static content of a website was first proposed in [1] and has been subsequently extended to dynamic content [55]. However, this work tend to propose controllers that keep CPU usage below a certain threshold, that should be determined through guesswork or prior knowledge of the platform. This solution works for web servers running on bare-metal hardware but it is unsatisfactory in cloud environments. On one hand, resources may be inefficiently utilized, due to the fact that the threshold needs to be set low enough to leave head-room in case of performance interference from co-located VMs.



(a) $\Theta = 0$.

(b) $\Theta = 0.25$.

(c) $\Theta = 0.50$.

(d) $\Theta = 0.75$.

(e) $\Theta = 1$.

(f) self-adaptive, $p_1 = 0.8$.

(g) Summary: non-adaptive and best self-adaptive configurations.

Figure 7: Statistical results for non-adaptive system. Figs. 7a to 7e show latency distribution for each non-adaptive configuration, while Fig. 7f shows our selected self-adaptive behavior. Each line represents a test-case. Fig. 6g shows, for each configuration, over all test-cases, number of served requests (bar with no fill), how many of them with recommendations (solid bar) and revenue (red, patterned bar).

On the other hand, CPU usage is not a reliable measure for spare capacity in virtualized environments, due to hypervisor preemption of virtual machines, also called steal time [23]. We use measured latency, which requires a carefully-designed controller, in order to avoid this limitation.

Other proposals exploit partial execution of computation to avoid overload. Response generation is interrupted after a certain time budged has been exhausted [33, 40], which is, however, only applicable to a narrow class of applications. Although in some specific cases, this may lead to better results, our aim is to provide a widely-applicable programming paradigm that can tackle the variety of fluctuations happening in a cloud environment. Therefore, we propose a general solution that can be applied both to existing and new cloud applications with minimal impact on their design.

An approach complementary to ours is elasticity [4, 14, 15, 29, 35, 52, 60, 68]. In cloud computing, elasticity means deciding the right amount of resources each application needs avoiding under- or over-provisioning. For example, Sharma et al. [59] propose a system that tries to minimize the cloud tenant's deployment cost while reacting to workload changes. The system takes into account the cost of each instance, the possibility of horizontal and vertical scaling and the transition time between configurations. However, elasticity offers no solution if the underlying infrastructure's capacity is exhausted, taking only the application's point-of-view.

Although some work deal with performance differentiation for multiple classes of clients [48], to our knowledge, the closest cloud application to a brownout-compliant one is Harmony [18]. Harmony adjust the consistency-level of a distributed database as a function of the incoming end-user requests, so as to minimize resource consumption. In this case, the motivation to introduce the adaptation mechanism lies in limiting the amount of money that the user is charged for. This is a specific example of how a cloud application can be compliant with our paradigm, while we propose a general technique and build a sufficiently broad control strategy to realize adaptivity for a vast class of applications.

Related to methodology, cloud research in general relies either on analytical models [8, 16, 63, 72, 73, 75] or on running actual experiments and building empirical traffic profiles and signatures [28, 52, 64, 66, 67]. Our system uses an analytical model to infer performance from measurements taken from the actual system. Zheng et al. [74] argue that running actual experiments is cheaper than building accurate models to validate research proposals. We build on this assumption, validating our technique on a small scale experimental testbed.

7. CONCLUSION

In this paper, we introduced the brownout paradigm for cloud applications. We discussed a model for applications with optional code, i.e., computations that can be activated or deactivated per client request. We described our experience with two widely-used, cloud benchmark applications, RUBiS and RUBBoS, to show the ease of applying our approach to existing cloud applications. We synthesized a controller for a wide range of applications, which selects the dimmer parameter based on incoming load and available capacity. We proved the correctness of the resulting system using control-theoretical tools. We implemented the framework and tested it with real-life experiments.

The results show that self-adaptation through brownout can allow applications to support more users or run on fewer resources than their non-adaptive counterparts. Hence, our proposition enables cloud applications to more robustly deal with unexpected peaks or unexpected failures, without requiring spare capacity.

Future work include extending the contribution to applications spanning multiple machines and combining brownout-compliance with other mechanism like horizontal/vertical elasticity and migration. We believe that brownouts open up a new level of flexibility in cloud platforms.

Acknowledgment.

This work was partially supported by the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control and through the LCCC Linnaeus Center. Also, we received partial support from the EU project VISION Cloud under grant agreement no. 257019, the Swedish Government's strategic effort eSSENCE and from the ELLIIT Excellence Center.

References

[1] T. F. Abdelzaher and N. Bhatti. "Web content adaptation to improve server overload behavior". In: *WWW*. 1999, pp. 1563–1577.

[2] M. Andersson, J. Cao, M. Kihl, and C. Nyberg. "Performance modeling of an Apache web server with bursty arrival traffic". In: (2003). URL: `http://lup.lub.lu.se/record/532529/file/625324.pdf`.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *PLDI*. 2009.

[4] D. Ardagna, B. Panicucci, and M. Passacantando. "A game theoretic formulation of the service provisioning problem in cloud systems". In: *WWW*. 2011, pp. 177–186.

[5] L. Baresi and C. Ghezzi. "The disappearing boundary between development-time and run-time". In: *FoSER*. 2010, pp. 17–22.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. "Xen and the art of virtualization". In: *SOSP*. 2003, pp. 164–177.

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

[8] M. Bhadauria and S. A. McKee. "An approach to resource-aware co-scheduling for CMPs". In: *ICS*. 2010, pp. 189–199.

[9] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. "Characterizing, modeling, and generating workload spikes for stateful services". In: *SOCC*. 2010, pp. 241–252.

[10] Y. Brun. "Improving impact of self-adaptation and self-management research through evaluation methodology". In: *SEAMS*. 2010, pp. 1–9.

[11] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. "Software Engineering for Self-Adaptive Systems". In: 2009. Chap. Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70.

[12] *Bulletin Board Benchmark*. URL: `http://jmob.ow2.org/rubbos.html`.

[13] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility". In: *Future Generation Computer Systems* 25.6 (2009), pp. 599–616.

[14] E. Caron, F. Desprez, and A. Muresan. "Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients". In: *J. Grid Comput.* 9.1 (2011), pp. 49–64.

[15] L. Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, and W. Binder. "Achieving application-centric performance targets via consolidation on multicores: myth or reality?" In: *HPDC*. 2012, pp. 37–48.

[16] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. "SLA Decomposition: Translating Service Level Objectives to System Level Thresholds". In: *ICAC*. 2007, pp. 3–13.

[17] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. "Software Engineering for Self-Adaptive Systems: A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Springer Berlin / Heidelberg, 2009, pp. 1–26.

[18] H.-E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Perez. "Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage". In: *CLUSTER*. 2012.

[19] W. Dawoud, I. Takouna, and C. Meinel. "Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning". In: *Global Trends in Computing and Communication Systems*. Vol. 269. Communications in Computer and Information Science. 2012, pp. 11–25.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (2007), pp. 205–220.

[21] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. E. Kaiser, and D. Phung. "A control theory foundation for self-managing computing systems". In: *IEEE J.Sel. A. Commun.* 23.12 (Sept. 2006), pp. 2213–2222.

[22] P. C. Diniz and M. C. Rinard. "Dynamic feedback: an effective technique for adaptive computing". In: *PLDI*. 1997, pp. 71–84.

[23] C. Ehrhardt. *CPU time accounting*. Last accessed: Aug. 2013. URL: `http://www.ibm.com/developerworks/linux/linux390/perf/tuning_cputimes.html`.

[24] R. T. Fielding and R. N. Taylor. "Principled design of the modern Web architecture". In: *ACM Trans. Internet Technol.* 2.2 (May 2002), pp. 115–150.

[25] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. "Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements". In: *ASE*. 2011, pp. 283–292.

[26] D. Fleder, K. Hosanagar, and A. Buja. "Recommender systems and their effects on consumers: the fragmentation debate". In: *EC*. 2010, pp. 229–230.

[27] D. F. García and J. García. "TPC-W E-Commerce Benchmark Evaluation". In: *Computer* 36.2 (Feb. 2003), pp. 42–48.

[28] D. Gmach, J. Rolia, and L. Cherkasova. "Selling T-shirts and Time Shares in the Cloud". In: *CCGrid*. 2012, pp. 539–546.

[29] Z. Gong, X. Gu, and J. Wilkes. "PRESS: PRedictive Elastic ReSource Scaling for cloud systems". In: *CNSM*. 2010.

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

[31] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. "Failure recovery: when the cure is worse than the disease". In: *HotOS*. 2013, pp. 8–14.

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

[33] Y. He, Z. Ye, Q. Fu, and S. Elnikety. "Budget-based control for interactive services with adaptive execution". In: *ICAC*. 2012, pp. 105–114.

[34] J. L. Hellerstein. "Why feedback implementations fail: the importance of systematic testing". In: *FEBID*. 2010, pp. 25–26.

[35] N. R. Herbst, S. Kounev, and R. Reussner. "Elasticity in Cloud Computing: What It Is, and What It Is Not". In: *ICAC*. 2013.

[36] N. Irwin. *These 12 technologies will drive our economic future*. May 2013. URL: `http://www.washingtonpost.com/blogs/wonkblog/wp/2013/05/24/these-12-technologies-will-drive-our-economic-future/`.

[37] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Q. Wang, J. Park, and C. Pu. "Expertus: A Generator Approach to Automate Performance Testing in IaaS Clouds". In: *CLOUD*. 2012, pp. 115–122.

[38] J. O. Kephart. "Research challenges of autonomic computing". In: *ICSE*. 2005, pp. 15–22.

[39] G. Kiczales and M. Mezini. "Aspect-oriented programming and modular reasoning". In: *ICSE*. 2005, pp. 49–58.

[40] J. Kim, S. Elnikety, Y. He, S.-W. Hwang, and S. Ren. "QACO: Exploiting Partial Execution in Web Servers". In: *CAC*. 2013.

[41] J. A. Konstan and J. Riedl. "Recommended to you". In: *IEEE Spectum* (Oct. 2012).

[42] J. Kramer and J. Magee. "Self-Managed Systems: an Architectural Challenge". In: *FOSE*. 2007, pp. 259–268.

[43] G. Kreitz and F. Niemela. "Spotify – Large Scale, Low Latency, P2P Music-on-Demand Streaming". In: *P2P*. 2010, pp. 1–10.

[44] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. "Supporting soft real-time tasks in the Xen hypervisor". In: *VEE*. 2010, pp. 97–108.

[45] W. Levine. *The Control handbook*. CRC Press, 1996.

[46] S. Malkowski, Y. Kanemasa, H. Chen, M. Yamamoto, Q. Wang, D. Jayasinghe, C. Pu, and M. Kawaba. "Challenges and Opportunities in Consolidation at High Resource Utilization: Non-monotonic Response Time Variations in n-Tier Applications". In: *CLOUD*. 2012, pp. 162–169.

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

[48] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. "Maestro: quality-of-service in large disk arrays". In: *ICAC*. 2011, pp. 245–254.

[49] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. "Quality of service profiling". In: *ICSE*. 2010, pp. 25–34.

[50] M. Nagappan, A. Peeler, and M. Vouk. "Modeling cloud failure data: a case study of the virtual computing lab". In: *SECLOUD*. 2011, pp. 8–14.

[51] F. F.-H. Nah. "A study on tolerable waiting time: how long are Web users willing to wait?" In: *Behaviour and Information Technology* 23.3 (2004), pp. 153–163.

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

[53] T. Patikirikorala, A. Colman, J. Han, and L. Wang. "A systematic survey on the design of self-adaptive software systems using control engineering approaches". In: *SEAMS*. 2012, pp. 33–42.

[54] D. C. Petriu, M. Alhaj, and R. Tawhid. "Software performance modeling". In: *SFM*. Berlin, Heidelberg: Springer-Verlag, 2012.

[55] J. Philippe, N. De Palma, F. Boyer, and e. O. Gruber. "Self-adaptation of service level in distributed systems". In: *Softw. Pract. Exper.* 40.3 (Mar. 2010), pp. 259–283.

[56] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *SOCC*. 2012.

[57] *Rice University Bidding System*. URL: `http://rubis.ow2.org`.

[58] A. W. Services. *Tutorial: Installing a LAMP Web Server*. Sept. 2013. URL: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html`.

[59] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. "A Cost-Aware Elasticity Provisioning System for the Cloud". In: *ICDCS*. 2011, pp. 559–570.

[60] U. Sharma, P. Shenoy, and D. F. Towsley. "Provisioning multi-tier cloud applications using statistical bounds on sojourn time". In: *ICAC*. 2012.

[61] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. "CloudScale: elastic resource scaling for multi-tenant cloud systems". In: *SOCC*. 2011.

[62] K. Sripanidkulchai, S. Sahu, Y. Ruan, A. Shaikh, and C. Dorai. "Are clouds ready for large distributed applications?" In: *SIGOPS Oper. Syst. Rev.* 44.2 (2010).

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

[64] C. Stewart, T. Kelly, and A. Zhang. "Exploiting nonstationarity for performance prediction". In: *EuroSys*. 2007, pp. 31–44.

[65] N. Tolia, D. G. Andersen, and M. Satyanarayanan. "Quantifying Interactive User Experience on Thin Clients". In: *Computer* 39.3 (Mar. 2006), pp. 46–52.

[66] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini. "DejaVu: accelerating resource allocation in virtualized environments". In: *ASPLOS*. 2012, pp. 423–436.

[67] A. Verma, L. Cherkasova, and R. H. Campbell. "ARIA: automatic resource inference and allocation for MapReduce environments". In: *ICAC*. 2011, pp. 235–244.

[68] S. Vijayakumar, Q. Zhu, and G. Agrawal. "Automated and dynamic application accuracy management and resource provisioning in a cloud environment". In: *GRID*. 2010, pp. 33–40.

[69] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, M. Kawaba, and C. Pu. "Response Time Reliability in Cloud Environments: An Empirical Study of n-Tier Applications at High Resource Utilization". In: *SRDS*. 2012, pp. 378–383.

[70] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. "A survey of formal methods in self-adaptive systems". In: *C3S2E*. 2012, pp. 67–79.

[71] D. Weyns, M. U. Iftikhar, S. Malek, and J. Andersson. "Claims and Supporting Evidence for Self-Adaptive Systems – A Literature Study". In: *SEAMS*. 2012.

[72] M. Woodside, T. Zheng, and M. Litoiu. "Service System Resource Management Based on a Tracked Layered Performance Model". In: *ICAC*. 2006, pp. 175–184.

[73] Q. Zhang, L. Cherkasova, and E. Smirni. "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications". In: *ICAC*. 2007, pp. 27–37.

[74] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. "JustRunIt: experiment-based management of virtualized data centers". In: *USENIX Annual Technical Conference*. 2009, pp. 18–28.

[75] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. "1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center". In: *ICAC*. 2008, pp. 172–181.

[76] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. "What does control theory bring to systems research?" In: *SIGOPS Oper. Syst. Rev.* 43.1 (Jan. 2009), pp. 62–69.