Event-Driven Application Brownout: Reconciling High Utilization and Low Tail Response Times

David Desmeurs
Department of Computing Science,
Umeå University, Sweden
Email: david.desmeurs@umu.se

Cristian Klein
SimScale GmbH, Germany[1]
Email: cklein@simscale.com

Alessandro Vittorio Papadopoulos
Department of Automatic Control,
Lund University, Sweden
Email: alessandro@control.lth.se

Johan Tordsson
Department of Computing Science,
Umeå University, Sweden
Email: tordsson@cs.umu.se

***Abstract*—Data centers currently waste a lot of energy, due to lack of energy proportionality and low resource utilization, the latter currently being necessary to ensure application responsiveness. To address the second concern we propose a novel application-level technique that we call event-driven Brownout. For each request, i.e., in an event-driven manner, the application can execute some optional code that is not required for correct operation but desirable for user experience, and does so only if the number of pending client requests is below a given threshold. We propose several autonomic algorithms, based on control theory and machine learning, to automatically tune this threshold based on measured application 95th percentile response times. We evaluate our approach using the RUBiS benchmark which shows a 11-fold improvement in maintaining response-time close to a set-point at high utilization compared to competing approaches. Our contribution is opening the path to more energy efficient data-centers, by allowing applications to keep response times close to a set-point even at high resource utilization.**

***Index Terms*—graceful performance degradation; event-driven control; machine learning; tail response time;**

I. Introduction

Data-center energy efficiency is an increasing concern. Already this sector consumes more than 1.3% of the world's electricity and is among the sectors with the fastest power increase [1]. Even worse, most of this energy is being wasted due to two reasons: lack of energy proportionality and low utilization. Current hardware lacks energy proportionality [2], i.e., the performance per watt is not linear with the utilization. This is due to the fact that some components, such as memory, disk, and even parts of the CPU, are consuming energy even when in idle. Lack of energy proportionality is problematic in itself, if hardware is operated at high utilization. Unfortunately, servers are mostly utilized between 10% and 50% [2]. The main reason is to leave headroom, so that applications may remain responsive, despite fluctuations in the number of users accessing them, i.e., the arrival rate. However, even assuming a constant average arrival rate, due to the variance in inter-arrival time, applications need some headroom to maintain responsiveness, as proved in queuing theory [3].

A promising method to increase hardware utilization without sacrificing application responsiveness is Brownout [4]. The method is inspired from brownouts in electrical grids, which are intentional or unintentional voltage drops meant to avoid blackouts by reducing energy consumption. Similarly, Brownout for cloud applications is a minimally-intrusive software engineering paradigm, which avoids application overload by selectively disabling some computations. In essence, a developer marks some content as essential, i.e., required to satisfy an end-user's request, and some as non-essential, i.e., it can be disabled but is highly desirable to serve. For example, for an e-commerce application the product description is essential, while recommendations of similar products can be seen as non-essential. In Brownout, a feedback controller monitors application response time and periodically recomputes the ratio of requests to serve with non-essential content during the next time period.

Brownout has shown to be a useful concept to avoid overloads. However, when Brownout is active, i.e., some non-essential content is dropped, the response time is often far off the desired set-point, and may present spikes. This is mostly due to two reasons: the periodic nature of the controller and the fact that controller decisions are based on response time alone. Experience with the related problem of load balancing [5] suggests that improvements can be achieved by taking a control decision for each request, i.e., in an event-driven manner, and by basing these decisions also on queue-length.

In this paper we present an improved approach to Brownout. We start by proposing a software architecture that externalizes the decision on serving optional content to a proxy, which implements the feedback control loop (Section III). By running isolated from the business logic, the proxy can more precisely monitor the number of requests in transit (commonly called queue-length) and response times of requests. Furthermore, the proxy decides for each request, i.e., in an event-driven manner, whether it should be served with optional content or not. In contrast to the previous approach where decision making was periodic, this allows the application to keep response times close to a set-point and CPU utilization higher.

The potential improvements of this architecture depend on autonomic algorithms that decide which requests to serve with optional content. We investigate solutions based on feedback control (Section IV-A) and on machine-learning with offline training (Section IV-B). Since each of these has advantages and disadvantages, we propose a combined approach, where the feedback loop is complemented with machine-learning as

[1]work partially done while working at Umeå University, Sweden

feedforward signal (Section IV-C). We evaluate all approaches through a common experiment based on the popular RUBiS cloud application benchmark and compare the results with the initial, periodic Brownout (Section V). Our results show a 11-fold improvement in maintaining response time close to a set-point at high utilization compared to the initial periodic Brownout [4] without lowering utilization. Although this improvement may vary a lot according to certain parameters such as the length of the pending request queue, as it is directly used to make the decision to serve optional content or not. For instance, we observed a 4-fold improvement in a scenario with low average queue-length.

By keeping applications responsive, despite having them operate at high utilization, our contribution is opening up the path to more energy efficient data-centers without requiring energy proportional hardware. Furthermore, our contribution may be of interest to other designers of autonomic control loops, to better understand the trade-off between event-driven and periodic decision making.

II. Background and Motivation

Admission control is a well-proven technique to ensure good resource utilization while avoiding system overload. A common way to guarantee web server performance is to drop and/or reorder certain requests when overloads (situations with many requests with very high response times) are detected by using diverse techniques related to request scheduling and admission control [6]–[9]. For instance, in [8], requests are sorted into classes, and each class has a weight corresponding to possible income for the application owner. The income is then maximized by an optimization algorithm. In [9], admission control is based on user-sessions, which can be admitted, rejected, or deferred. Another related work is the implementation of an autonomic risk-aware overbooking architecture that, by adaptive admission control, is able to increase resource utilization in cloud data centers by accepting more virtual machines than physical available resources [10].

Other works involving partial execution of requests exist. In [11], computations at the end of requests can be dropped to improve performance and meet given task deadlines. For that purpose, a set of scheduling algorithms are proposed and evaluated. Similar work is realized in [12], also with requests being time bounded implying lower answer quality but better performance. In [13], the quality of result is measured, i.e., there is a decline in quality when requests are not fully executed. In order to do so, certain requests are run twice, one time request execution completes, i.e., deliver all components, and a second time, requests are faster thanks to neglected data.

In the field of big data, dropping certain computations is also considered. In [14], a system named *ApproxHadoop*, an extension of Hadoop[1], is designed with three mechanisms proposed as a general approximation of MapReduce[2]. One of them is task dropping where only a subset of tasks are executed. An error bound is estimated for MapReduce jobs and once a certain target is achieved, remaining tasks are dropped. In [15], SQL queries are approximated to obtain error and response time constraints in order to improve performance. For that purpose, samples are created and then carefully selected based on an analysis of the data and past queries. As a result, a trade-off between accuracy and response time is made to be able run fast certain queries on large amounts of data. In [16], accuracy of queries is also investigated with a distributed system named *DICE* based on data cube exploration. This system includes a master/slave architecture where the master distributes queries to each slave and the slaves may either speculatively execute the queries or use cached data. As a result, queries are run faster with a certain level of approximation.

In cloud data centers, the Brownout paradigm [4] enables graceful user experience degradation with admission control carried out by removing optional contents from requests (as opposed to dropping requests). The initial version of this paradigm uses a controller that only takes into account the response time. The controller outputs a $dimmer$ value, such that $0 \leq dimmer \leq 1$, which is used to decide whether to serve optional contents or not. The $dimmer$ is periodically updated based on an error computed with a given set-point and the 95th percentile of the measured response times during the last control period. Then, for each following incoming request, the probability that optional contents are served depends on the $dimmer$ value. By using this process, the initial periodic Brownout has shown to successfully avoid overloads, however response times that significantly deviate from the set-point and occasional spikes have been observed. This is due to the periodic nature of its controller that makes the decision whether to serve optional contents based on response time only.

Initial experiments showed that basing decisions (to serve optional contents or not) on the queue-length, that is, the number of pending requests in a web server (a proxy in our case), leads to closer response times to the aimed set-point compared to what has been achieved with the initial periodic Brownout. Therefore, in this paper, the primary goal is to investigate an event-based approach where an event is triggered by the arrival of a new request. Hence we design and implement algorithms that make the decision, for each request, of whether to serve optional contents based on the queue-length in order to keep response times close to a set-point. Another design goal is to maximize the number of times optional contents are served. Deactivating optional contents reduces the user experience and should thus be applied only when necessary. For example, a study found that recommendations, which can be marked as optional, can increase sales of songs by 50% [17], which makes optional contents desirable, e.g., in e-commerce applications. By maximizing the number of optional contents served, CPU utilization is implicitly maximized as well (although keeping a small headroom can be adequate).

[1]https://hadoop.apache.org/

[2]http://research.google.com/archive/mapreduce.html

III. Architecture

Brownout can be deployed within a proxy for a web server. This proxy should be able to measure the arrival rate of incoming requests, the response time of each request, and the ongoing queue-length of pending requests. By using this information, Brownout is executed in a control loop each time period (e.g., every second) to update a **queue threshold value**. For each client request, the proxy piggybacks a boolean value that indicates if the current queue-length is below the threshold. The web server (logic layer) enables optional contents for requests with this flag set. Figure 1 outlines this architecture. In this paper we assume that requests are handled by a server deployed in a Virtual Machine (VM), as we consider Brownout to be used with virtualization in cloud data centers, even though it is not a requirement. Optional content may take time to generate, in particular if an interaction with a database is necessary. Requests without optional contents are thus leaner, resulting in lower response time. Subsequently there are three types of possible VM utilization:

- *Low utilization*. When a VM is lowly utilized, the 95th percentile response time (RT_{95}) is considered well below the set-point, even if optional content is served for all requests.
- *High utilization*. When a VM is highly utilized, optional contents are not always served in order to achieve application responsiveness. In this case RT_{95} varies around a set-point fixed in the Brownout algorithm, with the algorithm aiming to keep RT_{95} as close as possible to the set-point.
- *Overload*. When a VM is overloaded, application responsiveness is not achieved implying very high RT_{95}, far above the set-point, despite that no optional content is served.

If there is more than one VM, the proxy also acts as load balancer distributing requests to all VMs. In this case, the measurement of arrival rates, response times and queue-length must be separated, that is, distinct measurements for each VM. However, we henceforth consider only one VM as the focus of this paper is on the performance of our event-driven Brownout algorithms. Part of an envisaged future work is to include the event-driven Brownout algorithms with load balancing algorithms, just as it was done with the initial periodic Brownout [5]. Figure 1 represents the deployment of Brownout in a proxy interacting with a single VM.

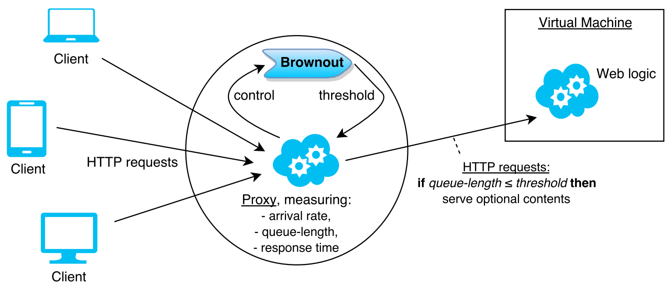

Fig. 1. Event-driven Brownout architecture. A proxy runs Brownout and interacts with an application, in this case a web server running inside a VM.

IV. Design of Autonomic Algorithms

In this section we describe the autonomic algorithms developed for event-driven Brownout. First we investigate a feedback controller approach, then Machine Learning Algorithms (MLAs) based on offline training, and finally a combination of the controller and a selected MLA based on online training.

A. Feedback Controller

To be able to dynamically set the queue threshold value used to decide whether optional contents should be served, we employ control techniques. Proportional-Integral-Derivative (PID) and PI controllers are widely used in practice, with more than 95% of all industrial control problems being solved by PID control [18], and control theory is a useful approach for self-managing systems [19]–[21].

1) PI Controller: We design a PI controller that sets the threshold to stabilize 95th percentile response times (RT_{95}) around a set-point. Measuring the 95th percentile response time instead of the average allows to produce consistent response times [22], and, overall, more timely responses for the users, hence improving their satisfaction [23]. The PI controller is outlined in Algorithm 1.

Algorithm 1 PI controller with filter and anti-windup
```
1:  $threshold \leftarrow 0$
2:  $previousError \leftarrow 0$
3:  $y \leftarrow 0$ // used to filter the process variable ($RT_{95}$)
4:  while server is running do
5:      if filter is activated then
6:          $y \leftarrow p * y + (1-p) * RT_{95}$ // p is the filter parameter
7:          $error \leftarrow setPoint - y$
8:      else
9:          $error \leftarrow setPoint - RT_{95}$
10:     end if
11:     $threshold \leftarrow threshold + K_p * (error -$
12:             $previousError) + K_i * error * controlPeriod$
13:     // Saturation, using $AR\_EWMA$, which is the expo-
14:     // nential weighted moving average of the arrival rate
15:     $threshold \leftarrow$min(max($threshold$, 0), $AR\_EWMA$)
16:     $previousError \leftarrow error$
17:     sleep for $controlPeriod$
18: end while
```

Given the error measured during each control time period ($controlPeriod$), the PI controller outputs a feedback, which is the threshold value. The parameters K_p and K_i, as well as the filter parameter p (with $0 \leq p < 1$), are tuned offline. The reason we use a PI controller instead of a PID controller, which includes a derivative part, is that the K_d parameter associated with the derivative part can be difficult to tune and often does not significantly improve performances [24]. It is estimated that only around 20% of all deployed controllers use a derivative part [25].

With a PI controller, the tuning parameters K_p and K_i determine how fast the controller reacts to significant changes. With high K_p and K_i, we observe that RT_{95} is further away from the set-point compared to when the arrival rate does not significantly change. Therefore, if only the PI controller is used to determine threshold values, a tradeoff must be found between RT_{95} close to the set-point but slow reactions to significant arrival rate changes, or RT_{95} further away from the set-point but with faster reactions to significant changes in arrival rate.

2) Windup and Anti-Windup Solution: To avoid the windup phenomena [18], [24], we need to bound the threshold. The obvious lower boundary is 0 as there is no need for the threshold to be lower than 0, the queue-length of incoming requests can never be negative. The upper boundary is less obvious. It is important to serve optional contents whenever possible so the user experience is not deteriorated, therefore the threshold should be high enough during low utilization in order to always serve optional contents, as the queue-length would always be below the threshold. We observe that taking the arrival rate as upper boundary produces a suitable anti-windup solution. Indeed it is unlikely that the queue-length for the next second will be higher than the previous measured arrival rate. However, for accuracy, the arrival rate exponentially weighted moving average (AR_EWMA in Algorithm 1) is taken as upper boundary to smooth the arrival rate that may fluctuate a lot.

3) Process Variable Filter: When the hardware configuration (e.g., number of CPU cores) allows a high number of requests to be treated at the same time, we observe that the queue-length is high on average, and so is the threshold when the VM is highly utilized. In contrast, when a low number of requests is served at the same time, the queue-length and therefore the threshold are much lower. As the threshold is an integer, very low values of threshold and arrival rate can lead to oscillations in RT_{95} due to discretization issues. If these oscillations are too significant, we observe that RT_{95} can be far away from the set-point, which is undesired. To counteract this problem, a filter on the process variable, i.e., RT_{95}, can be added. We use a low-pass filter, including a p parameter that determines how much noise is canceled. The filter is activated or deactivated based on a certain number of past threshold values, e.g., during the past 300 seconds. The filter is activated if the average threshold is below a value considered low, e.g., 10.

4) Bootstrap: Right after the proxy is initiated, the threshold is set to 0, and it needs some time to increase to appropriate values. As a result, some optional contents are not served during the first seconds no matter the workload, which is unwanted in case of low utilization. To avoid this problem, optional contents can always be served before any control, and then the initial threshold can be set to the first measured arrival rate. As a result, optional contents are always served at the beginning, and if the VM is overloaded, the threshold quickly decreases thanks to the controller in order to stabilize RT_{95} around the set-point. This technique is useful when the workload is unknown, which is common in real-world scenarios. This technique can also be applied when the controller is combined with a machine learning algorithm (Section IV-C). A machine learning algorithm does not produce useful outputs until it has been properly trained, implying that during that time, only the controller is used to update the threshold.

B. Machine Learning Approaches

Techniques from statistical machine learning have shown to be effective for feedback control in cloud data centers [26], [27]. As Algorithm 1 shows satisfying results in some situations according to how the tuning parameters are determined, learning from these results can lead to improvements, i.e., avoiding slow controller reactions to significant arrival rate changes, or RT_{95} being too far away from the set-point. Therefore we investigate other approaches to find appropriate thresholds with MLAs. Here the described MLAs are trained offline with data from **favorable states**.

1) Producing Favorable States: A favorable state is based on a set of measured data. A state S can be represented as a tuple:

$$S := (RT_{95}, arrival\text{-}rate, \%\text{-}optional\text{-}content, ...). \quad (1)$$

States are generated by measuring the arrival rate, RT_{95}, and amount the of optional content obtained using the PI controller with diverse workloads. Next, the favorable states are extracted, by keeping only the data points for which RT_{95} deviates by less than 10% from the set-point. Then, during offline training, MLAs are trained to later be able to reproduce favorable states.

2) Initial Approaches: We first investigated a classification algorithm, the perceptron, which is based on a neural network that classifies and associates favorable states with threshold values. However, even though we observed that the perceptron could produce decent results (i.e., RT_{95} not being too far away from the set-point), it does not seem the most appropriate MLA in this context. Indeed only one feature seems necessary – the arrival rate – and many favorable states are needed to train the perceptron. However, during these experiments we noted a proportional relation between arrival rate and threshold during high utilization (i.e., when Brownout is active, with optional contents sometimes being dropped). Hence we used the least square method to obtain a linear model, i.e., an equation of the form $threshold = a * arrivalRate + b$, where a and b are parameters determined by the least square method (using favorable states). However, thresholds obtained using this method are sensitive to the equation parameters, and they are incorrect when the VM is lowly utilized or overloaded as the values found for a and b are only suitable during high utilization. This is not the case when the VM is lowly utilized (the threshold being much higher than the queue-length) or overloaded (the threshold being 0 so the queue-length is always above the threshold implying no optional content being served). As these initial machine learning approaches have drawbacks, we instead opt for a simple algorithm that learns from past arrival rates and thresholds.

3) Mapping Arrival Rate to Threshold: A simpler way to obtain thresholds given the arrival rate is to map arrival rates to thresholds in case of favorable states, i.e., a new threshold is inserted in a map only when a favorable state is detected. With an offline training method, the map is eventually populated during an initial and long experiment where the workload slowly increases. This covers most possible arrival rates (until a certain upper limit) and thus enough favorable states should be produced. The map M is defined as

$$M = \{(ar_1 : t_1), (ar_2 : t_2), ..., (ar_n : t_n)\} \quad (2)$$

where n is the total number of keys/values in the map, ar_i is an arrival rate indexed by i, and t_i is the threshold corresponding to ar_i.

Algorithm 2 Mapping arrival rate to threshold

```
1: Map $M$ is populated with arrival rates and thresholds
   previously produced during offline training
2: $threshold \leftarrow 0$
3: $previousThreshold \leftarrow 0$
4: while server is running do
5:    $AR \leftarrow arrivalRate$
6:    $threshold \leftarrow$ median($\{M[AR + i]$  |
7:                              $i = 0, \pm1, \pm2, ..., \pm K\}$)
8:    if $threshold = null$ then
9:       $threshold \leftarrow previousThreshold$
10:   end if
11:   $previousThreshold \leftarrow threshold$
12:   sleep for $controlPeriod$
13: end while
```

As shown in Algorithm 2, the median of a set of thresholds close to t_i is used to obtain a final threshold given the arrival rate ar_i. Therefore a threshold can be obtained even if t_i does not exist in the map, and possible inaccuracies can be avoided. If there is still no output from the MLA ($threshold = null$ in Algorithm 2), then the previously determined threshold is used. Although, with a sufficient offline training, this should not happen. The number of closest thresholds is determined by a K value, which is inspired by the K Nearest Neighbors (KNN) algorithm [28]. We select the median instead of the average as the latter is sensitive to outliers. This type of outliers can occur when the measured arrival rates are in transit from low utilization with very high thresholds, to high utilization with proportional thresholds, or from high utilization to overload with thresholds set to 0 (so optional contents are never served). The parameter K should be carefully chosen. If K is too low, inaccuracies are possible as not enough thresholds in the map are taken into account. But K should not be too high to avoid to take irrelevant thresholds into account, that is, thresholds corresponding to arrival rates far away from the current measured one.

4) Undesirable Offline Training: Offline training may not be feasible for two reasons. First, in a real environment (e.g., a web developer wants to use Brownout with a web application) a benchmark of the web application is needed for the MLA to be trained, which is tedious. Another reason is that the context is likely to change. For instance, the application can run in a VM consisting of two cores, and later the VM is reconfigured to use eight cores. At this moment all previously learned data based on two cores would be useless, and another offline training would be necessary. Instead, to counteract these possible context changes, an online training approach is necessary, but the threshold must be set somehow to be trained with favorable states. For that, a controller can be combined with an MLA that is progressively trained online with detected favorable states.

C. Controller with Machine Learning

Combining a control algorithm with an MLA could achieve the best of the two approaches. Indeed the MLA cannot be executed alone as, at some point, it has to be trained with favorable states. Conversely using the controller only implies a tradeoff between RT_{95} being close to the set-point and slow controller reactions to significant arrival rate changes, versus a RT_{95} further away from the set-point but faster controller reactions to sudden changes in arrival rate.

1) PI Controller with MLA Feedforward: A possible approach is to use MLA output as feedforward signal in the control algorithm. The final controller output is equal to the feedback added to the feedforward. The MLA is used to obtain the feedforward component, with the mapping of arrival rate to threshold, by taking the median of a set of thresholds given the current arrival rate. The feedforward part is next handled by a filter, as shown in lines 8 to 12 in Algorithm 3.

2) Filtering the Feedforward Signal: As the MLA is trained online, inaccurate feedforward signals may occur, as well as non-existing ones (when the median function in Algorithm 3 does not contain any threshold). In order to keep the feedforward signal relevant, a filter is added. Subsequently the feedforward part does not change suddenly, and the feedback can quickly compensate to obtain appropriate thresholds.

3) Other Possible Approaches: In addition to the controller with MLA outputs as feedforward signal, we investigated two other approaches. The first approach is to either use the controller or the MLA. When the MLA has no output, the controller is used to set the threshold, and when the MLA has been trained enough to output threshold values, these values are directly used to set the threshold. The second approach is a dynamic equation technique where the threshold is set to $a * arrival\text{-}rate + b$ where a is a value set by the controller feedback, and b a value set thanks the least square method using threshold values in the map (of arrival rate to threshold). The reasons for this implied dynamic linear equation are the observed proportionality between arrival rate and threshold during high utilization, and the ability of the controller feedback (a value) to quickly adapt to arrival rate changes while still being able to handle low utilization and overload. However, even though these two approaches can produce decent results, they proved not as good as the above described controller with MLA outputs as feedforward.

Algorithm 3 PI controller with feedforward

```
1: $threshold \leftarrow 0$
2: $previousError \leftarrow 0$
3: $feedback \leftarrow 0$
4: $previousFeedforward \leftarrow 0$
5: $y \leftarrow 0$ // used to filter the process variable ($RT_{95}$)
6: while server is running do
7:     $AR \leftarrow arrivalRate$
8:     $feedforward \leftarrow$ median($\{M[AR + i]$ |
9:                                 $i = 0, \pm 1, \pm 2, ..., \pm K\})$
10:    $feedforward \leftarrow pf * previousFeedforward +$
11:        $(1 - pf) * feedforward$ // pf as filter parameter
12:    if filter is activated then
13:        $y \leftarrow p * y + (1 - p) * RT_{95}$ // p as filter parameter
14:        $error \leftarrow setPoint - y$
15:    else
16:        $error \leftarrow setPoint - RT_{95}$
17:    end if
18:    $feedback \leftarrow feedback + K_p * (error -$
19:        $previousError) + K_i * error * controlPeriod$
20:    $feedback \leftarrow$ min(max($feedback$, $-feedforward$),
21:                            $AR\_EWMA - feedforward$)
22:    $threshold \leftarrow feedback + feedforward$
23:    $previousError \leftarrow error$
24:    $previousFeedforward \leftarrow feedforward$
25:    sleep for $controlPeriod$
26: end while
```

Consequently, we mainly describe and evaluate the controller with MLA outputs as feedforward signals in this paper.

V. Evaluation

In this section we evaluate the algorithms for event-driven Brownout: the feedback controller, the MLA based on offline training, and the combination of controller and MLA based on online training. We take into account their performance, compare them, and discuss the most suitable one in a web server scenario. In addition we evaluate the initial periodic Brownout in the same conditions to determine the improvement of our event-driven algorithms.

The main focus of the presented performance evaluation is on how close the response time, i.e., RT_{95}, is to the set-point. However, in order to give an idea of the used energy, we show also CPU measurements.

A. Experiment setup

To be able to run experiments to test algorithms and produce results, the benchmark web application RUBiS[3] has been deployed in our cloud testbed. The RUBiS application has been modified in a previous work [4] to have a URL pointing towards a page including optional contents. The modification of RUBiS has an important impact: requests with optional contents need much more resources and therefore results in slower response times than requests without optional contents. The application is running inside a VM hosted by the Xen hypervisor [29]. Xen is deployed in a server consisting of a total of 32 CPU cores (AMD Opteron^{TM} 6272 at up to 2.1 GHz) and 56 GB of memory. The *lighttpd*[4] web server is installed in the domain-0 of Xen (the domain-0 being separated from the VM(s)). *Lighttpd* acts as a proxy to forward requests from emulated users to the VM, hence event-driven Brownout algorithms are implemented within the *lighttpd* source code. The RUBiS application running with an Apache web server in the VM is waiting for requests from the *lighttpd* proxy. To emulate users, we use the *Httpmon*[5] tool, which can be used in either open or closed model. In a closed model, a new request is only sent upon the completion of a previous request followed by a configurable think-time. In an open model, a new request is sent independently of previous requests' completion [30]. The intensity of the workload can be configured at run-time through the concurrency parameter, which roughly corresponds to the number of users accessing the website. For that purpose, *Httpmon* applies a Poisson distribution, a reasonably realistic model for emulating real website users sending requests [31].

We configured certain parameters for all evaluation experiments, as follows:

- The filter parameter p for filtering the process variable RT_{95}, and the filter parameter pf for filtering the feedforward signal (see algorithms in Section IV-A1 and Section IV-C1), are set to 0.6 and 0.95, respectively. These values have been selected after testing filters parameters in {0.5, 0.6, 0.7, 0.8, 0.9, 0.95} with repeated experiments to see which values gave to the best results.
- The exponentially weighted moving average smoothing the arrival rate, which is used as upper boundary in case of saturation (Section IV-A2), takes into account the 10 past measured arrival rates, which implies a 18.18% smoothing.
- The set-point is set to 0.5s. The reason is that, in general, users dislike requests taking too long and may give up [32]. As the 95th percentile response time implies a 5% tolerance of requests being above the set-point, a set-point of 0.5s is justified as it globally avoids most requests taking *set-point* + $\overline{error}$ second(s), where $\overline{error}$ represents the average deviation of RT_{95} from the set-point. Although, for applications where users would not mind waiting longer to receive responses, a higher set-point can be used.
- The control time period is 1 second, and measurements (such as the 95th percentile response times (RT_{95}), threshold values, percentages of optional contents served, and CPU utilization) are made each second and taken into account by the event-driven Brownout algorithms.
- The *Httpmon* tool is configured with a think-time of 3 seconds to emulate users sending requests.

Finally, the K_p and K_i parameters must be tuned for the

[3]http://rubis.ow2.org

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

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

control algorithms. Diverse methods for controller tuning exist [24]. However we employed a simple exhaustive approach, as satisfying results could be produced with this approach. For that purpose, we ran many experiments with sets of values for K_p and K_i to try all possibilities within certain limits. Indeed it was not useful to try too high values as we observed that RT_{95} becomes less and less close to the set-point with high K_p and K_i. We obtained the best results with a low value for K_p, around 1, and a higher value for K_p, around 6 (not shown with figures for briefness). In addition, a process variable filter can be activated when thresholds are low on average, as previously explained in Section IV-A3. We observed that low values for both K_p and K_i lead to slow controller reactions to significant arrival rate changes, but RT_{95} is closer to the set-point when no such changes occur. The reverse effect is observed with high values for both K_p and K_i. In our experiments, a significant arrival rate change occurs within seconds, when the concurrency is low and then suddenly high, or vice versa.

B. No Brownout vs. Initial Brownout

We produce experimental results with diverse number of concurrent emulated users sending requests. The number of concurrent users (concurrency) is changed every 100 seconds, which is specified at the top of the figures, and always the same (conc.: 200, 1200, 200, 700, 600, 500). The *Httpmon* tool is configured with a closed model, and the VM is configured with an 8 cores CPU.

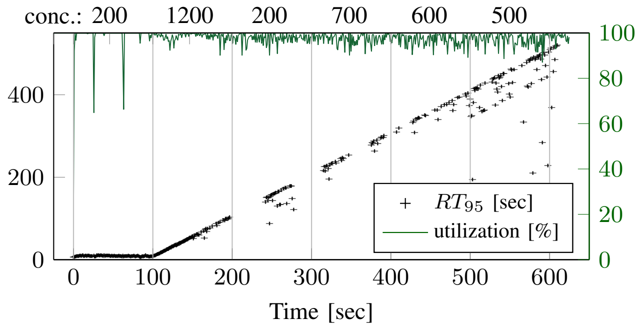

Fig. 2. No Brownout, RT_{95} in seconds (dotted) and CPU utilization [%].

Figure 2 shows results from an experiment where Brownout is not used, with RT_{95} (dotted) and CPU utilization (green line). As we can see, the VM is overloaded, with really high RT_{95}. When the largest concurrency (1200 concurrent users) is applied, RT_{95} starts to increase rapidly and keeps growing.

Figure 3 shows the effect of the initial periodic Brownout on the same workload, i.e., the same concurrency pattern. As we can see, RT_{95} is much lower than without Brownout. This is due to Brownout being activated (i.e., some optional contents are dropped) to avoid the VM to be overloaded. In order to do that, the *dimmer* value (red line), here represented as a percentage (right axis), is used as a probability implying that optional contents have $dimmer\%$ chance to be served. However, as previously mentioned, RT_{95} is not close to

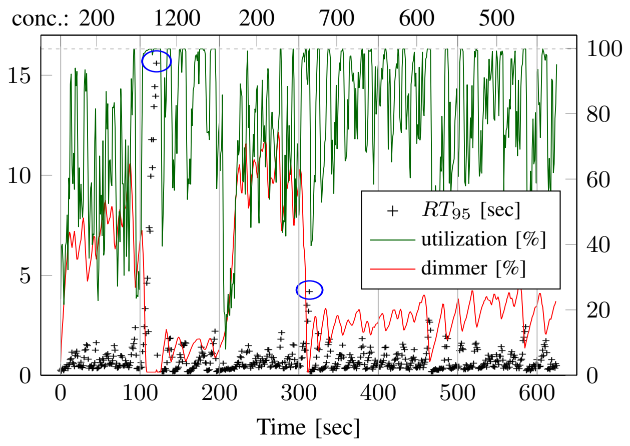

Fig. 3. Initial periodic Brownout activated with $pole = 0.9$ as recommended in [33], RT_{95} in seconds (dotted), dimmer and CPU utilization as percentages.

the set-point and sometimes presents spikes. Spikes appear prominently when the concurrency changes rapidly, such as from 250 to 1500 after 100 seconds.

C. Controller, Machine Learning, and Combined Approach

Here we present results obtained with the feedback controller, the MLA based on offline learning, and the combination of controller and MLA based on online training. The experiment settings are the same as the ones in Figure 2 and Figure 3 (i.e., same concurrencies, closed model, and 8 cores CPU) to easily compare the effects of the event-driven Brownout algorithms.

Figure 4 shows results from an experiment where the threshold is set with the controller feedback. The parameters K_p and K_i are set to 1, which is a low value in order to avoid threshold oscillations due to noise in RT_{95}. As we can see, when the concurrency suddenly changes, especially during the first half of the experiment, RT_{95} is far away from the set-point of 0.5s that is represented with the horizontal green line. As a result, RT_{95} needs some time to be reach the set-point again after significant concurrency changes, which is a slow controller reaction. This is also the case at the very beginning of the experiment. As the previous feedback (see Algorithm 1) is initially set to 0, it needs time to increase and, along with the controller algorithm, leads to appropriate thresholds. The two horizontal lines above and under the line for the set-point represent tolerance values (0.4s and 0.6s), therefore we consider it acceptable when RT_{95} is between these two lines (given the context of an 8 cores CPU, with a closed model). We can also see that, when the concurrency does not significantly change during the last 300 seconds of the experiment, the controller has no need to react fast, which is why RT_{95} is close to the set-point. The graph on the top shows the percentage of optional contents served and the CPU utilization in the VM. As expected, when the concurrency is high, the percentage of optional contents served is low, and reversely, when the concurrency is low, more optional content is served. As the VM is highly utilized for the whole experiment (i.e., the percentage of optional contents is always between 0% and

100%), the CPU utilization should be maximized (potentially with a small headroom), which is the case except for the low concurrency 200 where the headroom can be quite large.

Figure 5 shows results from an experiment made in the same context as the one in Figure 4, except that $K_p = 15$ and $K_i = 15$, which is an aggressive tuning. With higher values for the tuning parameters, we can see fast controller reactions to concurrency changes, but RT_{95} is further away from the set-point during the last 300 seconds of the experiment. The reason is that, with high K_p and K_i values, the threshold is quickly updated when errors are large due to significant arrival rate changes, but the threshold is also more sensitive to the noise observed in RT_{95}, which may degrade performances.

Overall, Figure 4 and Figure 5 respectively show low tuning and aggressive tuning. By using the moderate tuning described at the end of Section V-A (with $K_p = 1$ and $K_i = 6$), we can achieve the best performance, that is, RT_{95} begin close enough to the set-point and still have fast enough reactions to significant arrival rate changes.

Figure 6 shows results produced with the mapping of arrival rate to threshold as MLA, based on significant offline training. The results are satisfying and the threshold is quickly updated when the concurrency changes. The parameter K is set to 4 implying that up to 9 map entries are taken into account to produce thresholds.

Figure 7 shows results produced with the controller with MLA outputs as feedforward signals, based on online training. The controller is tuned with $K_p = 1$ and $K_i = 6$, as a tradeoff between RT_{95} being close to the set-point and fast reactions to significant arrival rate changes. The parameter K was set to 4 implying that up to 9 map entries are taken into account to obtain feedforward values. As we can see, RT_{95} follows the set-point closer and closer over time. This is thanks to the incremental training of the MLA where favorable states are collected during runtime, which gives better feedforward signals as the experiment progresses.

D. Performance Comparison and Discussion

We compare the performance of all evaluated algorithms as follows. First, we determine the total amount of error $error_{total}$. For that, with each measured RT_{95} during experiments (every second), the error $|setpoint - RT_{95}|$ is added to the total error, i.e., $error_{total} \leftarrow error_{total} + |setpoint - RT_{95}|$. All evaluation experiments are conducted within high utilization, implying that Brownout is always active with optional contents being served between 0% and 100%, and that RT_{95} oscillates around the set-point. Next we divide the total error with the duration of the experiment (in seconds) to obtain the mean absolute error. The mean absolute error represents how far RT_{95} is from the set-point.

1) Evaluation with Challenging Concurrencies: In order to evaluate all algorithms, we used the set of concurrencies shown in Figure 8. Each concurrency is used 100 seconds, implying a total time of 1600 seconds for each experiment, and all experiments are repeated 10 times for statistical significance, with mean absolute errors and their deviations being described

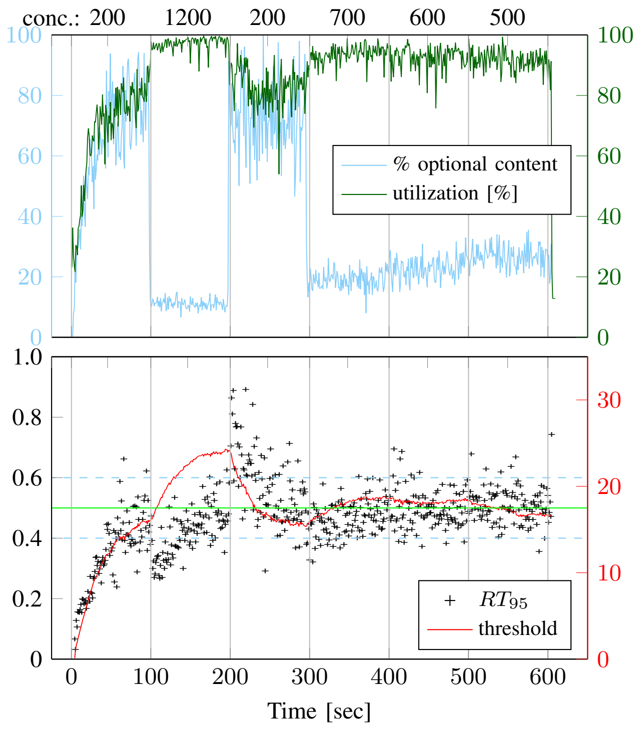

Fig. 4. Feedback controller, tuned with $K_p = 1$ and $K_i = 1$.

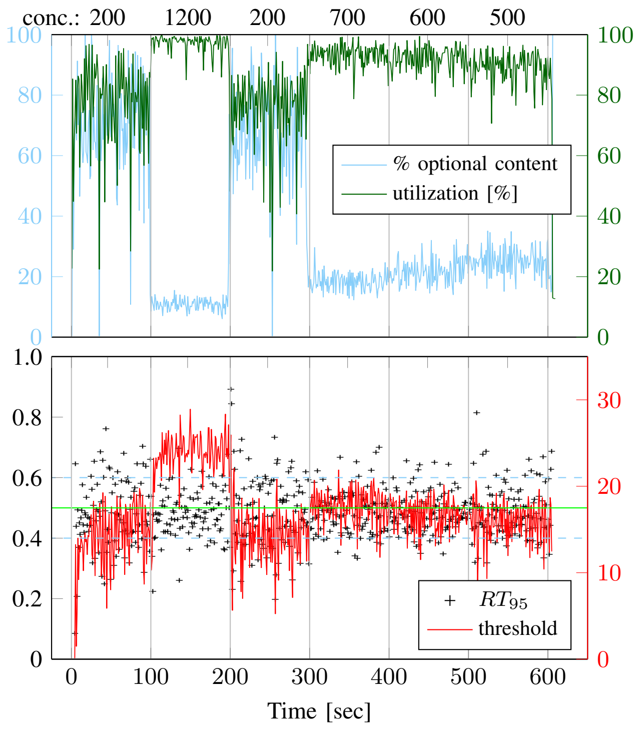

Fig. 5. Feedback controller, tuned with $K_p = 15$ and $K_i = 15$.

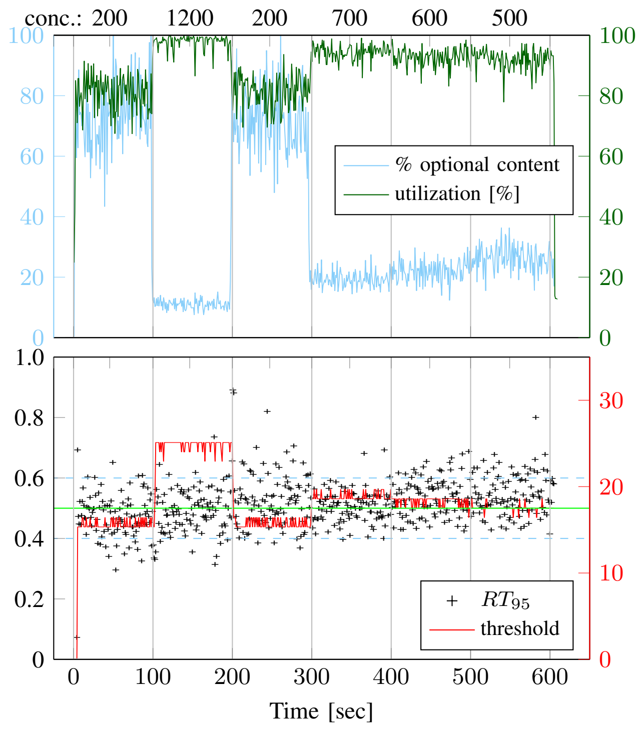

Fig. 6. Mapping of arrival rate to threshold, $K = 4$ (implying 9 map entries).

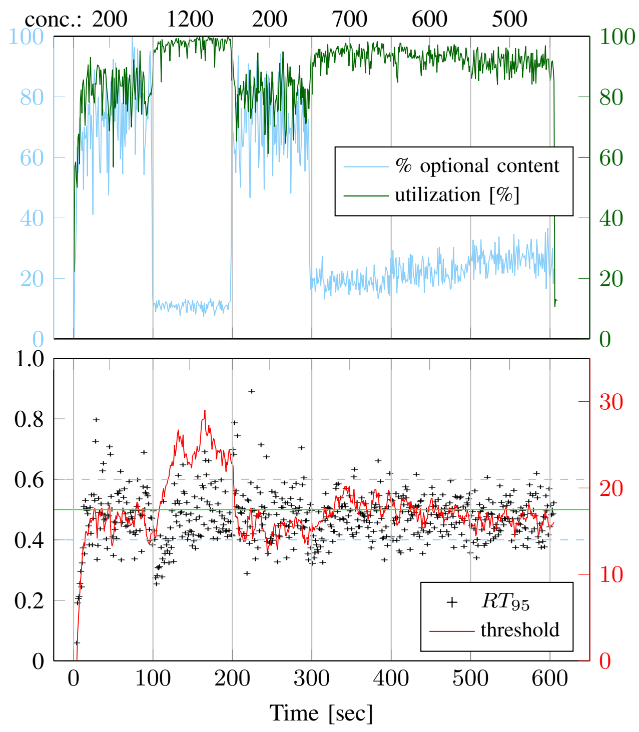

Fig. 7. MLA output as controller feedforward, $K_p = 1$, $K_i = 6$, $K = 4$.

in Table I. The first set of 8 concurrencies (for the first 800 seconds) is repeated with small differences (during the last 800 seconds) to see if the MLA can produce appropriate outputs even though it was not trained online during the first half of the experiment with the exact same concurrencies.

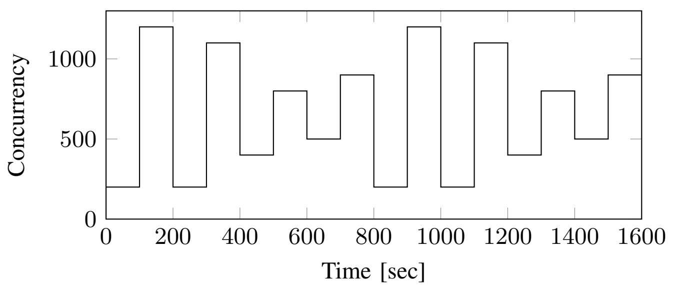

Fig. 8. Main evaluation experiment: set of 16 concurrencies.

In Table I, the second, fourth and fifth rows show the comparison between feedback controllers with low, high, and tradeoff value of K_p and K_i values. The experiment includes significant arrival rate changes (implied by the significant concurrency changes), which explain why the controller with low K_p and K_i values produces a high mean absolute error, as this controller reacts slowly to the significant arrival rate changes.

The mean absolute error obtained by using the mapping of arrival rate to threshold based on offline training is the lowest (first row of Table I), as the MLA already knows the threshold to output for any arrival rate, at any time in the experiment. For the controller with MLA outputs as feedforward signal (third row of Table I), it is fair to notice that its MLA starts with a handicap: it has no knowledge to begin with, and must be trained during the experiment, hence the obtained result is satisfying. One possibility to achieve a better performance would be to use MLA outputs only once the map is populated with a minimal number of entries. Hence the MLA would wait to be trained enough before having an impact to determine thresholds. However we chose not to do that in this evaluation experiment as it would not be fair. Indeed, if the MLA outputs are used in the middle of the experiment, then, as the next set of concurrency is approximately the same with the first part, the mean absolute errors would be optimal. In contrast, if the MLA outputs are used before or after the middle of the experiment, then mean absolute errors would not be optimal.

2) Ensuring RT_{95} Close to the Set-Point with Low Tuning: We claim that low K_p and K_i values enable RT_{95} to be close to the set-point when the arrival rate does not significantly change. In order to confirm this, we run another evaluation experiment with a constant concurrency of 600 emulated users (implying a highly utilized VM) to avoid significant arrival rate changes. The experiment is not repeated but is run for 2000 seconds in order to achieve stable results.

Table II shows the comparison between a PI controller with low K_p and K_i values and a PI controller with higher K_p and K_i values. For a fair comparison, we ignored the first seconds of the experiments as the feedback must increase

TABLE I
MAIN EVALUATION OF BROWNOUT ALGORITHMS. VM CONFIGURED WITH AN 8 CORES CPU. CONCURRENCIES DESCRIBED IN FIGURE 8.

Algorithm	**Mean error closed model**	**Mean error open model**	**Advantage**	**Drawback**
Mapping of arrival rate to threshold (offline training), $K = 4$	0.055 ± 0.002	0.056 ± 0.003	Overall, RT_{95} is close to the set-point enough, thanks to the offline training	Possible inaccuracies in the map (but not significant), and may require re-training
Feedback controller, tuned with $K_p = 1$ and $K_i = 6$	0.059 ± 0.001	0.059 ± 0.002	Tradeoff between fast reactions to significant arrival rate changes and RT_{95} being close to the set-point	
Controller with MLA outputs as feedforward signals, $K_p = 1$ and $K_i = 6$, and $K = 4$	0.063 ± 0.001	0.062 ± 0.001	Once the MLA has been sufficient trained, the controller feedback avoids MLA inaccuracies	May produce suboptimal results when the MLA has not been trained enough
Feedback controller, tuned with $K_p = 15$ and $K_i = 15$	0.068 ± 0.001	0.068 ± 0.001	Reacts fast to significant arrival rate changes	Does not keep RT_{95} close enough to the set-point if no significant changes in arrival rate
Feedback controller, tuned with $K_p = 1$ and $K_i = 1$	0.079 ± 0.002	0.081 ± 0.002	Keeps RT_{95} close to the set-point when there is no significant arrival rate changes	Reacts slow in case of significant arrival rate changes
Initial periodic Brownout, with $pole = 0.8$ as recommended in [4]	0.644 ± 0.057	0.836 ± 0.349	The pole set to 0.8 enables slightly faster controller reactions than with $pole = 0.9$	RT_{95} is still not close enough to the set-point, with possible spikes
Initial periodic Brownout, with $pole = 0.9$ as recommend in [33]	0.962 ± 0.191	1.732 ± 0.558	The pole set to 0.9 is the best configuration, according to [33]	RT_{95} is not close enough to the set-point, with possible spikes

TABLE II
EVALUATION OF FEEDBACK CONTROLLER ALGORITHMS. CONCURRENCY: 600. VM CONFIGURED WITH AN 8 CORES CPU.

Algorithm	**Mean error closed model**	**Mean error open model**
Feedback controller, tuned with $K_p = 1$ and $K_i = 1$	0.045	0.045
Feedback controller, tuned with $K_p = 15$ and $K_i = 15$	0.059	0.061

TABLE III
FEEDBACK CONTROLLER WITH FILTER VS. INITIAL PERIODIC BROWNOUT. VM CONFIGURED WITH A 1 CORE CPU. CONCURRENCIES: 40, 100, 160, 100, 40, 80, 90, 100, 110, 120

Algorithm	**Mean error closed model**	**Mean error open model**
Feedback controller + filter, tuned with $K_p = 1$ and $K_i = 6$	0.119 ± 0.011	0.122 ± 0.018
Initial periodic Brownout, with $pole = 0.9$ as recommended in [33]	0.438 ± 0.034	0.517 ± 0.054

initially to appropriate thresholds, in particular with low K_p and K_i values. As we can see, the lowest mean absolute error is obtained with low K_p and K_i values. This confirms that low K_p and K_i values imply RT_{95} to be more close to the set-point than with high K_p and K_i values, when the arrival rate does not significantly change.

3) Less Challenging Concurrencies: We observed that our event-driven algorithms are less satisfying when the queue-length and thus thresholds are low on average, which is the case with a VM configured with a 1 core CPU. On the other hand, the initial periodic Brownout performs better when the arrival rate does not significantly change, as RT_{95} spikes can be avoided. Therefore we run another evaluation experiment with a VM configured with a 1 core CPU, and a less challenging set of concurrencies (avoiding significant differences), which is: 40, 100, 160, 100, 40, 80, 90, 100, 110, 120. Each concurrency is used 100 seconds, implying a total time of 1000 seconds for each experiment, and all experiments are repeated 10 times for statistical significance, with mean absolute errors and their deviations being described in Table III.

Table III shows the comparison between the feedback controller with filter and the initial periodic Brownout. A filter is always activated for the controller as it is known that thresholds are low on average with a VM configured with a 1 core CPU. As we can see, the initial periodic Brownout preforms better than in the experiments summarized in Table I, whereas the event-driven Brownout performs worse than in Table I. However, even in this setting the event-driven Brownout outperforms the initial periodic version.

E. Discussion

Let us now discuss the event-driven Brownout algorithms and their improvement over the initial periodic Brownout.

1) Comparison with the Initial Periodic Brownout: As described in Section II, the initial periodic Brownout consists of a controller taking into account RT_{95} alone. Response times are not close to the set-point and occasionally present spikes due to sudden changes in the workload. By running the evaluation with challenging concurrencies with the initial periodic Brownout (Table I), we obtain the best results with $pole = 0.8$ and a closed model where the mean absolute error is 0.644 ± 0.057s. By taking into account the performance of the feedback controller tuned with $K_p = 1$ and $K_i = 6$, which has the lowest error average (excluding the MLA based on offline training), the event-driven Brownout achieves a eleven-fold improvement compared to the original Brownout. However it is fair to notice that this improvement has been achieved within a certain context including: the hardware configuration, such as the choice of an eight cores CPU; the set of concurrencies; the RUBiS application benchmark with the way optional contents are computed, i.e., the PHP code and MySQL queries; the chosen think-time of 3 seconds with the Poisson distribution to emulate requests. In different contexts, such as the one described in Table III where the improvement factor is around 4 (taking into account results for both closed and open models), the event-driven Brownout algorithms may perform differently. Based on the experiments, we conclude that the improvement is *up to* 11 times better.

2) Advantages and Drawbacks: The designed and implemented algorithms have certain advantages and drawbacks as described in Table I. With the feedback controller, RT_{95} being close to the set-point mainly depends on how the parameters K_p and K_i are tuned. With a MLA alone, the offline training method is what determines how optimal the threshold values will be, that is, the accuracies and amount of threshold matching arrival rates. However offline training may not be feasible in real environments. With the controller with MLA outputs as feedforward signal, the MLA is based on progressive online training. Therefore critical moments can occur when the MLA has not been trained enough, or when it produces inaccuracies.

3) Event-Driven Brownout in a Web Server Scenario: If one of the algorithms had to be selected to be deployed in a web server scenario, the selection would most likely depend on the web application type. Assuming an application where unexpected peaks of requests can frequently occur, the controller with MLA outputs as feedforward signals would probably be the most suitable method. Indeed, if a feedback controller is used instead, it might react too slow to unexpected peaks, unless it is tuned with high K_p and K_i values. However, in that case RT_{95} would not always be close enough to the set-point, in particular when the arrival rate does not change significantly. Therefore, the controller with MLA outputs as feedforward signals would enable fast reactions to these peaks while keeping RT_{95} close enough to the set-point. However this approach is not completely robust as the MLA may not have been trained enough, or it may present inaccuracies.

On the other hand, for an application where unexpected peaks of requests are less frequent, a simple feedback controller would suffice. Tuned with moderate K_p and K_i values, such as 1 for K_p and 6 for K_i, the controller would keep RT_{95} close enough to the set-point, and the controller would not need to react fast to significant arrival rate changes as unexpected peaks would not be frequent. Even if the arrival rate keeps increasing or decreasing over long periods of time, the controller would be able to keep satisfying RT_{95} as the arrival rate changes would not be sudden. A filter should also be activated when the required threshold values are low on average in order to limit large threshold fluctuations.

VI. Conclusion and Future Work

To improve the initial Brownout paradigm, we investigate an event-based approach based on the queue-length of pending requests to make the decision of whether serving optional contents. We designed and implemented event-driven algorithms that output a threshold used to make the decision. First we developed a PI controller that updates the threshold each control period. The PI controller showed decent results, but we aimed to obtain improvements with machine learning approaches. Hence, based on favorable states that the controller can produce, we investigated MLAs and selected a mapping of arrival rate to threshold. As a MLA must somehow learn from existing data and offline training is not always feasible, we implemented a combination of the PI controller and the MLA based that is trained online. Finally we evaluated all approaches, compared them, and found that our event-driven Brownout algorithms show an improvement in maintaining response-time close to the set-point of up to 11 times without lowering utilization.

As the initial periodic brownout has been tested with load balancing algorithms [5], the event-based version of Brownout could also be tested with these algorithms as part of a future work. Both Brownout versions avoid overloads, but the event-based version described in this paper also obtains response times closer to the desired set-point. This effect could alter results obtained with the load balancing algorithms, therefore it is worth to investigate the event-driven Brownout action with such algorithms. In addition, the initial periodic Brownout has been included within an overbooking system, where the purpose was to develop and improve reactive methods in case of overload situations [34]. Again it would be worth investigating the effect of the event-driven Brownout on this work, to see whether the results differ.

In cloud data centers, elasticity enables operators to provide or withdraw resources autonomically, to match demand to the available resources as close as possible [35]. For this purpose, autoscaling methods are applied to, for instance, start new VMs to avoid overloads commonly detected by monitoring server utilization or response times. Overall, diverse autoscaling algorithms exist with techniques based on control theory, reinforcement learning, queuing theory, time-series analysis, or simple static threshold-based rules [36]. However, for Brownout-enabled applications, neither the response time nor CPU utilization are good metrics for overload detection (and thus not for autoscaling). If a Brownout-enabled application is autoscaled, then the used elasticity algorithm is likely to produce unwanted behaviors. Brownout avoids overloads until a certain point, but it degrades the user experience, which is undesirable. Therefore a possible future direction is to create Brownout aware autoscaling algorithms. Intuitively these autoscaling algorithms should consider VMs overloaded as soon as Brownout is active (i.e., optional contents are sometimes dropped), even though, in term of CPU utilization, these VMs are not overloaded thanks to Brownout. Taking this into consideration would imply that new VMs are started to avoid overloads and optional contents would still always be served unless there are no more available resources, or during the time new VMs are booting, which can take several minutes.

Acknowledgments

This work was partially supported by the Swedish Research Council (VR) for the project "Cloud Control", and through the LCCC Linnaeus and ELLIIT Excellence Centers. The authors would also like to thank Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez who have worked on the initial version of Brownout.

References

[1] J. Koomey, "Worldwide electricity used in data centers," *Environmental Research Letters*, vol. 3, 2008.

[2] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE computer*, vol. 40, no. 12, pp. 33–37, 2007.

[3] C. Millsap, "Thinking clearly about performance," *Queue*, vol. 8, no. 9, pp. 10:10–10:20, 2010.

[4] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 700–711.

[5] C. Klein, A. V. Papadopoulos, M. Dellkrantz, J. Dürango, M. Maggio, K.-E. Årzén, F. Hernández-Rodriguez, and E. Elmroth, "Improving cloud service resilience using brownout-aware load-balancing," in *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*, 2014, pp. 31–40.

[6] A. Robertsson, B. Wittenmark, M. Kihl, and M. Andersson, "Design and evaluation of load control in web server systems," in *American Control Conference, 2004. Proceedings of the 2004*, vol. 3, 2004, pp. 1980–1985 vol.3.

[7] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in e-commerce web sites," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. ACM, 2004, pp. 276–286.

[8] M. Andersson, J. Cao, M. Kihl, and C. Nyberg, "Admission control with service level agreements for a web server," in *IASTED International Conference on Internet and Multimedia Systems and Applications*. ACTA Press, 2005, pp. 275–280.

[9] A. Ashraf, B. Byholm, and I. Porres, "A session-based adaptive admission control approach for virtualized application servers," in *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, 2012, pp. 65–72.

[10] L. Tomás and J. Tordsson, "An autonomic approach to risk-aware data center overbooking," *Cloud Computing, IEEE Transactions on*, vol. 2, no. 3, pp. 292–305, 2014.

[11] Y. He, S. Elnikety, and H. Sun, "Tians scheduling: Using partial processing in best-effort applications," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, June 2011, pp. 434–445.

[12] Y. He, S. Elnikety, J. Larus, and C. Yan, "Zeta: Scheduling interactive services with partial execution," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: ACM, 2012, pp. 12:1–12:14.

[13] J. Kelley, C. Stewart, N. Morris, D. Tiwari, Y. He, and S. Elnikety, "Measuring and managing answer quality for online data-intensive services," *CoRR*, vol. abs/1506.05172, 2015.

[14] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 383–397, Mar. 2015.

[15] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: Queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 29–42.

[16] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi, "Distributed and interactive cube exploration," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, March 2014, pp. 472–483.

[17] D. Fleder, K. Hosanagar, and A. Buja, "Recommender systems and their effects on consumers: The fragmentation debate," NET Institute, Working Papers 08-44, 2010.

[18] K. J. Åström and R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton, NJ, USA: Princeton University Press, 2008.

[19] A. Leva, M. Maggio, A. V. Papadopoulos, and F. Terraneo, *Control-Based Operating System Design*. Institution of Engineering and Technology, 2013.

[20] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel, "Software engineering meets control theory," in *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS 15, 2015.

[21] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[22] B. King, *Performance Assurance for IT Systems*. CRC Press, 2004.

[23] 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," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[24] K. Åström and T. Hägglund, *Advanced PID Control*. ISA-The Instrumentation, Systems, and Automation Society, 2006.

[25] K. H. Ang, G. Chong, and Y. Li, "Pid control system analysis, design, and technology," *IEEE Transactions on Control Systems Technology*, vol. 13, no. 4, pp. 559–576, 2005.

[26] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, ser. HotCloud'09. USENIX Association, 2009.

[27] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva, "Comparison of decision-making strategies for self-optimization in autonomic computing systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 4, pp. 36:1–36:32, Dec. 2012.

[28] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[29] 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," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, 2003.

[30] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," in *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. USENIX Association, 2006, pp. 18–18.

[31] M. Arlitt and C. Williamson, "Internet web servers: workload characterization and performance implications," *Networking, IEEE/ACM Transactions on*, vol. 5, no. 5, pp. 631–645, Oct 1997.

[32] "Amazon found every 100ms of latency cost them 1% in sales," http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, accessed: 2015-02-01.

[33] M. Maggio, C. Klein, and Årzén, "Control strategies for predictable brownouts in cloud computing," in *Proceeding of the 19th IFAC World Congress*, vol. 19, 2014, pp. 689–694.

[34] L. Tomás, C. Klein, J. Tordsson, and F. Hernández-Rodriguez, "The straw that broke the camel's back: safe cloud overbooking with application brownout," *International Conference on Cloud and Autonomic Computing (CAC)*, pp. 151–160, 2014.

[35] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. USENIX, 2013, pp. 23–27.

[36] T. Lorido-Botran, J. Miguel-Alonso, and J. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.