Algorithms for Event-Driven Application Brownout

David Desmeurs

April 3, 2015
Master’s Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Johan Tordsson, Co-Supervisor: Cristian Klein
Examiner: Frank Drewes

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

Existing problems in cloud data centers include hardware failures, unexpected peaks of incoming requests, or waste of energy due to low utilization and lack of energy proportionality, which all lead to resource shortages and as a result, application problems such as delays or crashes. A paradigm called Brownout has been designed to counteract these problems by automatically activating or deactivating optional computations in cloud applications. When optional computations are deactivated, the capacity requirement is decreased, which enables low enough response times to obtain responsive applications. Brownout has shown to successfully avoid overloads, however response times are often unstable and they sometimes present spikes due to sudden changes in the workload. This master thesis project is a contribution to the existing Brownout paradigm, to improve it. The goal is to find a way to stabilize response time around a certain set-point by taking the number of pending requests into consideration.

We designed and implemented new algorithms to improve Brownout and we produced experimental results based on the popular web application benchmark RUBiS. The RUBiS application was modified and deployed on a virtual machine in a Xen environment, and it received requests from emulated clients through a proxy. On this proxy, we first implemented a controller to set a threshold determining if optional computations shall be activated or not in the RUBiS application. Then we investigated machine learning algorithms using an offline training method to be able to set correct thresholds. As an offline training method is not desirable in real environments, we combined the controller and machine learning algorithms, such as using the outputs of the latter as controller feedforward, to obtain satisfying results.

Experimental results showed that the new Brownout algorithms can improve the initial Brownout by a factor up to 6. We determined this improvement by taking the 95th percentile response times into account, and comparing how far they are on average from a selected set-point. According to this improvement, determining if optional computations shall be activated or not based on queue-length of pending requests is a good approach to keep the response time stable around a set-point.
# Contents

1 Introduction ................................. 1
   1.1 Brownout Paradigm .......................... 1
   1.2 Thesis Outline ............................. 2

2 Problem Description ......................... 3
   2.1 Background and Related Work ............... 3
   2.2 Problem Statement ........................... 5
   2.3 Goals ................................... 5
   2.4 Methods .................................. 6
   2.5 Definitions and Notation ................... 7

3 Control Theory .............................. 9
   3.1 PID/PI Controller ........................... 9
   3.2 Fast Reaction or Stable Response Time ........ 11
   3.3 Windup and Anti-Windup ..................... 13
   3.4 Optimizing Utilization or Response Time ........ 15
   3.5 Controller Output Filter ................. 16
   3.6 Aspect to Consider in Real Environment ... 16
   3.7 Discussion ................................ 17

4 Machine Learning ............................. 19
   4.1 Training with Favorable States ............ 19
   4.2 Perceptron ................................. 20
   4.3 Setting Thresholds with Linear Regression .... 20
   4.4 Mapping Arrival Rate to Threshold .......... 22
   4.5 Online versus Offline Training ............ 22
   4.6 Discussion ................................ 23

5 Controller with Machine Learning .......... 25
   5.1 Controller or Machine Learning ............ 25
   5.2 Controller and Machine Learning .......... 26
   5.3 Dynamic Equation ........................... 28
5.4 Discussion .................................................. 29

6 Evaluation ........................................ 31
   6.1 Measuring Performance ................................. 31
   6.2 Algorithms Performance .................................. 32
      6.2.1 Controller ........................................ 32
      6.2.2 Machine Learning .................................. 33
      6.2.3 Controller with Machine Learning ...................... 34
      6.2.4 Initial Brownout .................................. 35
   6.3 Advantages and Drawbacks ............................ 35
   6.4 Most Suitable Algorithm .............................. 36

7 Conclusion ........................................ 39
   7.1 Limitations ........................................ 39
   7.2 Lessons Learned ..................................... 39
   7.3 Future Work ........................................ 40

8 Acknowledgements ................................ 41

References ......................................... 43

A Source Code ...................................... 47

B Source Code Integration ....................... 59
List of Figures

2.1 Initial Brownout architecture. ............................................... 3
2.2 No Brownout, concurrencies and response time. .......................... 4
2.3 Initial Brownout activated, concurrencies and response time. .......... 4
2.4 Initial Brownout, response time and dimmer. ............................. 5
2.5 Initial Brownout, CPU utilization. ....................................... 5
2.6 New Brownout architecture. ................................................. 6
2.7 Experimental results production process. .................................. 7

3.1 PID controller. ................................................................. 10
3.2 PI controller, low $K_p$ and $K_i$ values (respectively 5 and 1), response time and threshold. .................................................. 11
3.3 PI controller, low $K_p$ and $K_i$ values (respectively 5 and 1), optional contents and utilization. ............................................. 11
3.4 PI controller, high $K_p$ and $K_i$ values (respectively 20 and 5), response time and threshold. .................................................. 12
3.5 PI controller, high $K_p$ and $K_i$ values (respectively 20 and 5), optional contents and utilization. ............................................. 12
3.6 Windup, response time and threshold. .................................... 13
3.7 Windup, optional contents and utilization. ................................ 13
3.8 Anti-Windup, response time and threshold. ............................... 14
3.9 Anti-Windup, optional contents and utilization. ........................ 14
3.10 Optimizing CPU utilization. ................................................ 15
3.11 Optimizing response time. ................................................ 15
3.12 PI controller without filter, response time and threshold. .............. 17
3.13 PI controller with filter, response time and threshold. ................ 17

4.1 Linear regression, response time and threshold. .......................... 21
4.2 Linear regression, utilization and optional contents. ..................... 21
4.3 Mapping AR to threshold, response time and threshold. ................. 23
4.4 Mapping AR to threshold, utilization and optional contents. .......... 23
5.1 Controller or machine learning, response time and threshold. .......... 26
5.2 Controller or machine learning, utilization and optional contents. .... 26
5.3 PID controller with feedforward. ......................................... 26
5.4 Controller and machine learning, response time and threshold. ........ 28
5.5 Controller and machine learning, utilization and optional contents. ... 28
5.6 Dynamic equation, response time and threshold. .......................... 29
5.7 Dynamic equation, utilization and optional contents. .................... 29
5.8 Dynamic equation, response time and a value. ............................ 30
5.9 Dynamic equation, response time and b value. ............................ 30
6.1 Workload for the common evaluation experiments. ....................... 32
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Evaluation of controller algorithms.</td>
<td>33</td>
</tr>
<tr>
<td>6.2</td>
<td>Evaluation of controller algorithms, concurrency 400 for 1000 seconds.</td>
<td>33</td>
</tr>
<tr>
<td>6.3</td>
<td>Evaluation of MLAs based on offline learning.</td>
<td>33</td>
</tr>
<tr>
<td>6.4</td>
<td>Evaluation of controller with MLAs.</td>
<td>34</td>
</tr>
<tr>
<td>6.5</td>
<td>Evaluation of advantages and drawbacks of all algorithms.</td>
<td>36</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

More than 1.3% of the world’s electricity is consumed in data centers, and the power consumption in this sector quickly increases [27]. In cloud data centers, unexpected events are common. For instance, peaks in the amount of incoming requests can happen, increasing the number of requests up to 5 times [15]. Failure of hardware components is particularly frequent as many servers (more than 100,000 for certain data centers) are running to provide diverse services [34]. Moreover, it has been observed that a high proportion of the utilized energy is wasted [14], often due to the lack of energy proportionality and low utilization of hardware components [14]. The lack of energy proportionality means that the number of watts needed to run services is not linear with the hardware utilization. The reason is that some components (such as memory, disk, or CPU) are consuming energy even when they are not utilized. Low utilization is due to headroom, that is, in order for an application to be responsive, the server where the application operates should not be used at 100%, as predicted by queueing theory [29]. However it has been observed that servers are mostly used between 10% and 50% [14], which is too low and a waste of energy.

1.1 Brownout Paradigm

The Brownout paradigm is a method to make cloud data centers more energy efficient. Brownout enables hardware utilization to be increased while keeping applications responsive to avoid overloads. The process is autonomic, which means that no manual interactions are necessary for Brownout to work [24]. Brownout is a self adaptive system with a control loop model based on a feedback process. The process includes four key activities: collecting data (such as the response time), analyze the data, deciding (algorithm outputs), and acting (to avoid overloads) [17].

The idea behind the Brownout paradigm is as follows. To avoid overloads, optional computations might sometimes be deactivated so the response time is low enough to obtain responsive applications. Removing optional computations for certain requests makes them leaner, hence Brownout is a special type of per request admission control. Indeed some requests are fully admissible, while others may only be admissible without optional computations. An example of application can be an e-commerce website, where product descriptions are shown along with related products suggested to the end users. The related products can be marked as optional content as they are not strictly necessary for the website to work. Indeed, even if the related products section improves the user experience, it is often preferable to sacrificing them to obtain a more responsive website. The problem is to know
when optional computations – or optional contents – should be served or not.

1.2 Thesis Outline

This thesis is separated into eight chapters. After this first introduction chapter, the second chapter describes the problem including the goals as well as methods used to run experiments. The third chapter describes the design, implementation, and experimental results of a feedback controller for Brownout. The fourth chapter consists in a presentation of what can be achieved by implementing machine learning algorithms (MLAs), with results based on an offline training method. The fifth chapter describes the combination of control algorithms and MLAs, including relevant results. The sixth chapter evaluates and compares the three proposed methods of the previous chapters. Finally, the seventh chapter is a conclusion of the work, including possible future works with Brownout, followed by acknowledgments, a list of references, and appendices illustrating parts of the Brownout algorithms source code.
Chapter 2

Problem Description

The initial Brownout shows good results to avoid overloads, which increases hardware utilization. However the response time is not always stable and sometimes presents spikes. Therefore we need to investigate a new approach to improve Brownout.

2.1 Background and Related Work

The initial Brownout is designed and implemented with a controller taking into account the response time alone [24]. The controller outputs a dimmer value such as $0 \leq \text{dimmer} \leq 1$, which is a probability used to serve or not optional contents. The dimmer is periodically updated based on the error computed with a set-point and the response time. The response time taken into account is the 95th percentile of the measured response times during the last time period. For example, with a set-point of 0.5 second, and a 95th percentile response time of 0.7 second during the last time period, the error is equal to $0.5 - 0.7 = -0.2$. The control algorithm takes the error into account to produce a new dimmer value. Then, for each following incoming request, the probability that optional contents are served depends on the dimmer value (as it is a value comprised between 0 and 1). Figure 2.1 represents the initial Brownout architecture.

![Initial Brownout Architecture](image)

Figure 2.1: Initial Brownout architecture.
We ran an experiment to show the effect of the initial Brownout compared to an experiment using the same settings but without Brownout. Figure 2.2, created from the results of the experiment without Brownout, shows the 95th percentile response time (dotted) and the concurrency (straight lines) that is the concurrent amount of emulated users sending requests. As we can see, we first set a low concurrency of 100 users during the first 100 seconds, and we observe that the 95th percentile response time is low as well. Then, when we increase the concurrency to 500 and then to 600, the 95th percentile response time increases a lot and reaches very high values. At this point, the virtual machine where the requests were sent is overloaded. Figure 2.3 shows the effect of the initial Brownout with the same set of concurrencies. As we can see the 95th percentile response time is much lower. This is due to Brownout being activated (i.e., some optional contents are dropped) to avoid the virtual machine to be overloaded.

Other works related to Brownout exist. For instance, Brownout has been tested with load balancing algorithms to improve cloud service resilience [25]. The goal in this work was to implement brownout-aware load balancing algorithms and see if cloud services could remain responsive, even in case of cascading failures. Brownout has also been included within an overbooking system, where the purpose was to develop and improve reactive methods in case of overload situations [32].

Brownout is related to the more global problem of performance and admission control. A common way to guarantee web server performance is to drop and/or reorder certain requests when overloads are detected by using diverse techniques related to request scheduling and admission control [30, 20, 10, 12]. For instance, in [10], 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 [12], the admission control is based on user-sessions that can be admitted, rejected, or deferred. Another related work is the implementation of an autonomic risk-aware overbooking architecture that is able to increase resource utilization in cloud data centers by accepting more virtual machines than
2.2 Problem Statement

The initial Brownout shows unstable response times that sometimes present spikes. This is due to the periodic nature of its controller, and because the decision to serve or not optional contents is based on response time alone.

![Initial Brownout, response time](image1)

![Initial Brownout, CPU utilization](image2)

Figure 2.4: Initial Brownout, response time and dimmer.

Figure 2.5: Initial Brownout, CPU utilization.

Figure 2.4 shows the 95th percentile response time (dotted) and the dimmer value. Figure 2.5 shows, for the same experiment as figure 2.4, the CPU utilization of the virtual machine (VM) used during the experiment. As we can see, the 95th percentile response time is not stable and sometimes presents spikes. Spikes especially appear when the concurrency changes, which can be seen at the top of the graphs (conc.: 500, 850, 600, 700, 50). The concurrency is the number of emulated users sending requests. For concurrencies 500, 850, 600, and 700, the dimmer is always between 0 and 1, which implies that optional contents have \((\text{dimmer} \times 100)\)% chance to be served. In these cases Brownout is used to avoid overloads, in contrast to the last 100 seconds where the concurrency is low (50) and optional contents are always served (as \(\text{dimmer} = 1\)) without affecting the application responsiveness (the 95th percentile response time being always low).

2.3 Goals

Initial experiments showed that basing decisions (serving or not optional contents) on the queue-length, that is, the number of pending requests in a web server (a proxy in our case), leads to more stable response times compared to the initial Brownout method. Therefore the primary goal is to implement algorithms that determine whether to serve optional contents based on the queue-length in order to keep stable response times. In addition, the CPU
utilization as well as the number of times optional contents are served should be maximized. Indeed, deactivating optional contents deteriorates the user experience, which is why it should be applied only when necessary. For example, a study found that recommendations (a type of optional content) can increase by 50% the number of song sales [21], which makes optional contents desirable in applications. The CPU utilization should be maximized as well (although keeping a small headroom is adequate) to make an efficient use of energy.

2.4 Methods

To be able to run experiments to test algorithms and produce results, the benchmark web application RUBiS [8] has been deployed in the university cloud testbed. The RUBiS application has been modified to have a URL pointing towards a page with or without optional contents, and it is installed on a virtual machine (VM). The underlying architecture where the VM operates is based on Xen. Xen permits to have multiple commodity operating systems deployed on VMs sharing a conventional hardware [13]. A proxy named lighttpd [6] has been deployed in the domain-0 of Xen (the domain-0 being separated from the VM(s)). Figure 2.6 represents a schema of the architecture. Lighttpd acts as a proxy to send requests from emulated users to the VM. The RUBiS application running with an Apache web server in the VM is waiting for requests from the lighttpd proxy. A tool named Httpmon [5] is used to send diverse amount of requests (hence the term emulated users) to the lighttpd proxy. Httpmon applies the Poisson distribution [36], a reasonably realistic model for emulating real website users sending requests, and it is possible to use Httpmon with an open or closed model. In a closed model, a new request is only sent by the completion of a previous request followed by a think-time (Httpmon allows to choose the think-time). In an open model, a new request is sent independently of other request completions [31].

![Figure 2.6: New Brownout architecture.](image)

The VM may run with one to eight CPU cores. We decided to run experiments either with one or eight cores to see the two extremes. However we observed that keeping the response time stable with one core was more difficult to achieve than with eight cores, therefore most of the results presented in this thesis have been produced from experiments using eight cores. Indeed, when it is easier to keep the response time stable, good and bad results are more obvious, that is, we can easily see how far the response time is compared to the set-point. Most of our experiments were based on a closed model, and graphs presented in this thesis have been made with a closed model and a think-time of 3 seconds. The main
reason to choose a closed model is that the first results produced for the initial Brownout were based on a closed model, which permits to easily compare the two approaches. However, in the evaluation chapter, we also include textual results based on an open model, to see the difference.

We implemented the Brownout algorithms within the lighttpd source code (lighttpd is written in C). Measurements are produced each second (i.e., the time window – or time period – is 1 second) and a threshold value is updated. The decision for serving optional contents or not is made for each request. If the current queue-length is below the threshold, then optional contents are served, otherwise they are not. In addition to the measurements produced within the lighttpd proxy (e.g., the arrival rate, the number of times optional contents are served, and so on), the CPU utilization is measured using `virt-top`, a tool to measure CPU utilization in virtual environments, deployed in the domain-0 of Xen.

We produce experimental results with the following process. First we implement or modify an algorithm within the lighttpd source code. Then we choose the settings, such as the different number of concurrent users and how much time they send requests (in general, 100 seconds for each concurrency). Then we start an experiment. Once the experiment is finished, we use scripts to plot the results. These scripts enable to fully automate the experimentation process to avoid experimental bias due to human intervention. Given the results we either are satisfied and investigate new possibilities, or we modify the algorithm(s) and/or change the settings, and then run a new experiment. Figure 2.7 represents this cyclic process.

![Experimental results production process](image)

Figure 2.7: Experimental results production process.

### 2.5 Definitions and Notation

Algorithms and experimental results are based on measured data, which are defined as follows:

- **Arrival rate (AR).** The arrival rate (AR) is the number of new incoming requests during the last second (as measurements are made periodically each second).

- **Queue-length.** The queue-length is the number of pending requests in the proxy, in other words, the number of requests waiting to be treated.
- **Response time.** The response time is the total amount of time from the moment a request is sent by the client to the moment the response (produced by the server) is fully received by the client. It is important to notice that, during our experiments, there were no network delays as requests were sent directly within the Xen environment. Therefore all the developed algorithms do not take possible network delays into account.

- **95th percentile response time** ($RT_{95}$). Response times are measured during each time period of 1 second. The 95th percentile of the measured response times is taken into account by the algorithms.

- **Exponential moving average.** The exponential moving average is the weighted average of measured data, with more weight being given to a certain amount the latest data. We chose this amount to be 10, which applies an 18.18% weighting to the most recent measured data. This choice can be justified with the exponential moving average following closely the measured data, as it can be seen in [7] (graph under the section *The Lag Factor*).

- **Utilization or CPU utilization.** The utilization is the percentage of time that the CPU is actively executing instructions on behalf of the VM where the RUBiS application is deployed.

Some of important terms used in this thesis are defined as follows:

- **Threshold.** Whenever a new request is to be treated, the number of pending requests in the queue, the queue-length, is known. The threshold is used so when the queue-length is lower than the threshold, then optional contents are served, otherwise they are not.

- **Low utilization.** When a VM is lowly utilized, the 95th percentile response time ($RT_{95}$) is considered low, such as between 0 and 0.3 second, even if optional contents are always served.

- **High utilization.** When a VM is highly utilized, optional contents are not always served in order to achieve application responsiveness. In this case $RT_{95}$ is around a set-point fixed in the Brownout algorithm (e.g., a set-point of 0.5 second), in a more or less stable manner.

- **Overload.** When a VM is overloaded, application responsiveness is not achieved implying very high $RT_{95}$, despite no optional contents being served.

- **Favorable state.** A favorable state occurs when a certain threshold value leads to $RT_{95}$ being close to the set-point. With a set-point of 0.5 second, and in case of high utilization, $0.45 < RT_{95} < 0.55$ is considered favorable.

- **Significant arrival rate change.** The concurrency, or concurrent number of users sending requests, can change over time. The arrival rate is the measured number of incoming requests each time period, which is monotonically influenced by the concurrency. A significant arrival rate change occurs within a few seconds, when the concurrency is low and then suddenly high, or vice versa.
Chapter 3

Control Theory

To be able to set the threshold in order to decide if optional contents should be served or not, we employed techniques related to control theory to obtain satisfying results. Proportional-Integral-Derivative (PID) controllers are widely used in practice [26], and control theory is a useful approach for self-managing systems [19]. Therefore we designed and implemented a PID controller that determines appropriate thresholds to stabilize the response time around a set-point.

3.1 PID/PI Controller

A PID controller takes into account an error that is the subtraction of a set-point and a process variable. In our case, the process variable is the 95th percentile response time ($RT_{95}$) measured in seconds. The set-point is a predetermined value so $RT_{95}$ should oscillate around the set-point. We chose a set-point of 0.5 second because, in general, users dislike requests taking too long and may give up [1]. As we take into account the 95th percentile response time, it means that we tolerate 5% of requests measured each time period being above the set-point. Choosing the 95th percentile response time instead of the average allows to produce consistent response times [23], and, overall, more timely responses for the users, hence improving their satisfaction [18]. The best case scenario would be to have $RT_{95} = setpoint = 0.5$, but in reality such accuracy cannot be achieved, and thus the best case is $RT_{95}$ oscillating as close as possible around the set-point. Therefore, given the tolerance we accept, a set-point of 0.5 second is justified as it globally avoids most requests taking $setpoint + error$ second(s), where $error$ represents how far $RT_{95}$ is, on average, compared to the set-point. However the set-point is a configuration parameter, and could have been set to another value. For instance, a higher set-point could be more suitable for certain types of applications, such as applications whose users do not mind waiting a little bit more to receive responses. Or a lower set-point for applications where really slow response times are excepted.

Given the fixed set-point $setpoint$, the error measured for each time period is $setpoint - RT_{95} = 0.5 - RT_{95}$. With this error, the PID controller outputs a feedback equal to $K_p \times error + K_i \times integral + K_d \times derivative$. The integral term $integral$ accumulates the last errors. For that purpose, the integral value is initialized to 0 and, for each control, $integral = integral + error \times dt$ where $dt$ is the time period of 1 second ($dt = 1$). The integral is a simple addition of accumulated errors as it is a discrete model (measurements being produced each second, which is not continuous). The derivative term $derivative$ predicts
future errors by analyzing error changes. For that purpose, \( \text{derivative} = \frac{\text{error} - \text{error}_{\text{previous}}}{\text{dt}} \), where \( \text{error}_{\text{previous}} \) is the error calculated prior to the current error \( \text{error} \), and \( \text{dt} \) is the time period equal to 1 second. The values \( K_p \), \( K_i \), and \( K_d \) are tuning parameters that are pre-determined (constant values). To illustrate how the controller feedback – the threshold – is calculated, the pseudo-code algorithm is as follows:

```cpp
/*
 * Initialization:
 */
set_point = 0.5
integral = 0
error_previous = 0

/*
 * For each control, each dt second(s):
 */
error = set_point - RT_95
integral = integral + error * dt
derivative = (error - error_previous) / dt
error_previous = error
threshold = Kp * error + Ki * integral + Kd * derivative
```

A PI controller is the same as a PID controller without the derivative part. Therefore, when \( K_d = 0 \), one can talk about PI control. The reason for not using the derivative part in most of our experiments is that, in noisy environments, the derivative can lead to an unstable process variable [3]. Because of the fluctuating impact the derivate part can have on real environments, it is estimated that around only 20\% of the deployed controllers use the derivative part [11]. The \texttt{Httpmon} tool employed to emulate requests uses the Poisson distribution with a pre-defined think-time, that we chose to be 3 seconds. This leads to a noisy environment as the number of requests sent by emulated concurrent users fluctuates differently over time due to the Poisson distribution, even if the concurrent number of users remains the same.

![PID controller diagram](image)

**Figure 3.1: PID controller.**

Figure 3.1 represents a PID controller diagram. In our experiments \( K_d \) is set to 0,
making the controller a PI controller. With the measured $RT_{95}$ subtracted to the set-point, an error is produced. With the error a feedback is determined by adding the $P$, $I$ and $D$ parts together, and the threshold is directly set with the feedback. During the process (of 1 second) the threshold determines if optional contents shall be served or not, and $RT_{95}$ is measured at the end of the process.

### 3.2 Fast Reaction or Stable Response Time

Diverse methods for tuning the $K_p$ and $K_i$ parameters exist, such as the Ziegler-Nichols method [11]. 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 stable with too high $K_p$ and $K_i$ values. We obtained the best results with a low value for $K_i$, around 1 or 2, and a higher value for $K_p$, around 5 to 8. However, if low values permit to have the most stable response time, the controller reacts slowly to adjust $RT_{95}$ in case of significant AR changes.

We produced experimental results with diverse number of concurrent emulated users. A significant AR change happens when a certain number of concurrent users – or concurrency – is set for a given period of time, and then a much higher of lower number of concurrent users is set for another given period of time. We chose to change the numbers of concurrent users each 100 seconds, and the concurrency is specified at the top of the graphs presented in this thesis when necessary.

![Figure 3.2: PI controller, low $K_p$ and $K_i$ values (respectively 5 and 1), response time and threshold.](image1)

![Figure 3.3: PI controller, low $K_p$ and $K_i$ values (respectively 5 and 1), optional contents and utilization.](image2)

Figure 3.2 shows the results from an experiment where the threshold is directly set to the feedback output of the controller, which is $K_p \cdot \text{error} + K_i \cdot \text{integral}$. The parameters $K_p$ and $K_i$ are respectively set to 5 and 1, which are low values permitting a stable $RT_{95}$ around
the set-point when the AR is stable as well. As we can see, when the concurrent number of
users changes, $RT_{95}$ is far away from the set-point (0.5). The set-point is represented with
the horizontal green line. $RT_{95}$ needs some time to be around the set-point again after the
concurrency changes during the first part of the experiment, which is a slow reaction. This is
also the case at the right beginning of the experiment, as the integral is initially set to 0. In
this case the integral needs time to increase and, along with the whole controller algorithm,
produces appropriate threshold values. The two horizontal blue lines represent tolerance
values (0.4 and 0.6), that is, when $RT_{95}$ is between these two lines it is acceptable (given
the context of an eight cores CPU, with a closed model). Although the tolerance values are
only shown to have a visual idea of how close $RT_{95}$ is from the set-point, they do not have any
other usage. We can also see that, when the concurrency does not significantly change (from
500 to 550 and then to 600), the controller has no need to react fast, which is why $RT_{95}$ does
not go too far away from the set-point. Figure 3.3 shows the percentage of optional contents
served and the CPU utilization of the VM for the same experiment. As expected, when
the concurrency is high, the percentage of optional contents served is low, and vice versa.
Since, for the whole experiment, the VM is highly utilized (i.e., the percentage of optional
contents is always between 0% and 100%), the CPU utilization should be maximized with
a headroom, which is around 2% to 20% in this experiment. The headroom is higher when
the concurrency is low, most likely because fewer requests are treated in parallel.

![Figure 3.4: PI controller, high $K_p$ and $K_i$ values (respectively 20 and 5), response time and threshold.](image1)

![Figure 3.5: PI controller, high $K_p$ and $K_i$ values (respectively 20 and 5), optional contents and utilization.](image2)

Figure 3.4 shows the results from an experiment made in the same context as the one in
Figure 3.2 and Figure 3.3, except that $K_p = 20$ and $K_i = 5$. With higher values for these
tuning parameters, we can see that the controller reacts faster to concurrency changes, but
$RT_{95}$ is less stable. When comparing Figure 3.3 and Figure 3.5, we can see that the controller
slow or fast reaction does not significantly affect CPU utilization and percentage of optional
contents begin served. The CPU utilization and optional content percentage in Figure 3.3
are respectively 89.20 and 36.13 on average, compared to 90.63 and 38.78 in Figure 3.5.
3.3 Windup and Anti-Windup

An existing problem with the integral part of the controller is called Windup [2]. This problem occurs when the integral accumulates errors, and the process variable, the threshold, is outside certain boundaries. The threshold can go beyond the boundaries when the VM is lowly utilized (i.e., optional contents are always served). The error is thus always positive (as $RT_{95}$ is always lower than the set-point in the experiments) and the integral term keeps increasing. Afterwards, if the VM is more utilized, optional contents may sometimes have to be deactivated to keep $RT_{95}$ around the set-point. However, as the integral is very high due to the windup, it will take too long to decrease, which implies too high thresholds produced by the controller feedback. Windup also happens when the VM is overloaded, that is, even if optional contents are never served, $RT_{95}$ is still too high. In this case the error is always negative, the integral keeps decreasing, and it will take too long to increase again when necessary.

Figures 3.6 and 3.7 show two cases of integral windup. First the concurrency is 50, which implies that the VM is lowly utilized. Therefore the error is always positive as $error = setpoint - RT_{95}$ with $RT_{95}$ being always lower than the set-point. Consequently the integral keeps increasing and the controller outputs higher and higher feedbacks used as threshold. When the concurrency changes to 400, then the threshold must decrease to obtain a stable $RT_{95}$. But since the integral increased so much, it takes a lot of time (around 50 seconds in this experiment) for $RT_{95}$ to be around the set-point. Eventually $RT_{95}$ is around the set-point and, when the time is 200 seconds, the concurrency changes to 1200, which implies an overloaded VM. Therefore $RT_{95}$ is very high, and the error ($error = setpoint - RT_{95}$ with $RT_{95} > setpoint$) is always negative. Consequently the integral keeps decreasing, and therefore the threshold is set to values far below 0. When the time is 300 seconds, the concurrency is set back to 400, but since the integral decreased so much, it takes too long to go back to a suitable value. Therefore the threshold keeps being below 0.

![Diagram](image.png)

**Figure 3.6:** Windup, response time and threshold.

**Figure 3.7:** Windup, optional contents and utilization.
instead of increasing above 0 for $RT_{95}$ to be around the set-point again.

Diverse methods exist to avoid the Windup problem. One of them is to determine boundaries for the final output of the controller (the threshold) and use these boundaries to detect an integral windup. The obvious lower boundary is 0 as there is no need for the threshold to be lower than 0 (the queue-length being always positive). The upper boundary is less obvious. Indeed it is important to always serve optional contents whenever possible so the user experience is not deteriorated. Therefore the threshold should be high enough during low utilization so the queue-length is always below the threshold, resulting in the optional contents being always served. We observed that taking the AR as upper boundary produces a satisfying anti-windup solution. Indeed it is unlikely that the queue-length for the next second will ever be higher than the previous measured AR. Although, for accuracy, the AR exponential moving average is taken as upper boundary. That is because the AR fluctuates a lot, therefore the exponential moving average enables to smooth AR fluctuations while still being accurate in case of significant AR changes.

![Figure 3.8: Anti-Windup, response time and threshold.](image1)

![Figure 3.9: Anti-Windup, optional contents and utilization.](image2)

Figures 3.8 and 3.9 show two cases of integral anti-windup. The concurrencies are the same as in the previous experiment, but boundaries have been set to detect windup. When the VM is lowly utilized (concurrency set to 50) the integral does not increase when the threshold goes beyond the AR exponential moving average. Therefore there is no problem for quickly reaching relevant thresholds when the concurrency changes from 50 to 400. When the VM is overloaded (concurrency set to 1200) the integral does not decrease when the threshold goes below 0. Therefore, when the concurrency changes to 400 again, the integral just has to increase a little bit until appropriate thresholds can be reached. How quickly the integral increases or decreases after a low utilization period or an overload period also depends on the $K_p$ and $K_i$ values (as the higher these values are, the faster the controller reacts to significant AR changes).
3.4 Optimizing Utilization or Response Time

We shortly investigated an approach in order to optimize utilization or response time by looking at produced errors. As $error = setpoint - RT_{95}$, the error can either be positive with $RT_{95}$ being lower than the set-point, or negative with $RT_{95}$ being higher than the set-point. When $error > 0$, the error can be multiplied by a pre-defined parameter $K$ (with $K > 0$) so the threshold, set by $K_p * error + K_i * integral$, increases faster than it decreases. In this case the number of optional contents served is statistically higher, and therefore $RT_{95}$ is higher as well. But utilization is more optimized as it is closer to 100%, with a small headroom. If, on the contrary, the error is multiplied by $K$ only when $error < 0$, then the threshold decreases faster than it increases. In this case the number of optional contents served is statistically lower, and therefore $RT_{95}$ is lower as well. Thus $RT_{95}$ is optimized as it is statistically more often lower than the set-point. However utilization is slightly less optimized. The results obtained also depend on the parameter $K$. $K$ must be high enough to observe an optimization, but not too high otherwise we observed CPU utilization spikes.

\begin{figure}[h]
\centering
\includegraphics[width=0.45\textwidth]{cpu_utilization.png}
\caption{Optimizing CPU utilization.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.45\textwidth]{response_time.png}
\caption{Optimizing response time.}
\end{figure}

Figures 3.10 shows results where CPU utilization is optimized. As we can see, $RT_{95}$ is more often higher than the set-point. $K$ is set to 5 and the concurrency is set to 400 for the whole experiment. The average CPU utilization is 92.8% and the median is 93.4%. The average $RT_{95}$ is 0.562 second and the median is 0.552 second.

Figures 3.11 shows results where the response time is optimized. As we can see, $RT_{95}$ is more often lower than the set-point. $K$ is set to 5 and the concurrency is set to 400 for the whole experiment. The average CPU utilization is 90.2% and the median is 90.7%. The average $RT_{95}$ is 0.457 second and the median is 0.454 second.

Overall, there is a small CPU utilization difference, around 2% according to our measurements, but a neat different with $RT_{95}$, which is around 0.1 second.
3.5 Controller Output Filter

When the hardware configuration allows a high number of requests to be treated at the same time, we observe that the queue-length is high on average, as well as the threshold when the VM is highly utilized. In contrast, when a low number of requests is treated at the same time, the queue-length and therefore the threshold are much lower. The hardware configuration can be the number of CPU cores (and their frequency), as we observe a significant decline in the average queue-length with one core CPU compared to an eight cores CPU (and assuming the same other experiment settings, such as the set of concurrencies used). The lower the threshold is, the more its updates are important, which are due to fluctuations in $RT_{95}$ measurements, as $RT_{95}$ is known to be volatile. Increasing or decreasing the threshold by, for example, 3, has thus more impact when the threshold is equal to 4 on average, than when it is equal to 20 on average. If this impact is too important, $RT_{95}$ is less stable around the set-point, which is unwanted even though the controller can react fast to significant AR changes. To counteract this problem a filter on the controller output – the threshold – can be added. A controller output filter limits significant controller output updates regardless of the underlying cause. Subsequently, controller output filters can reduce persistent controller output fluctuations that may degrade the controller performance [4]. However a filter also adds a delay for sensing the true threshold values, which can have negative consequences, such as slow reactions to significant AR changes. To avoid that, we use the exponential moving average with a period of 10, that is, the 10 previous threshold values are taken into account so an 18.18% weighting is applied to the last of these threshold values. As a result of doing this, we obtain fast enough reactions to significant AR changes while preventing the threshold to increase or decreasing too quickly when the AR does not significantly change.

![Figure 3.12: PI controller without filter, response time and threshold.](image1)

![Figure 3.13: PI controller with filter, response time and threshold.](image2)

To produce Figure 3.12 and Figure 3.13, we ran a specific experiment with a one core CPU and a constant concurrency leading to serving optional contents around 30% of the time. No filter is used in the experiment for Figure 3.12, and a controller output filter is
used with the experiment for Figure 3.13. In both experiments, $K_p = 8$ and $K_i = 2$, which imply an important impact when updating the threshold, as a one core CPU is used (which is why the threshold is around 4 on average). As we can see, the filter enables a more stable $RT_{95}$ around the set-point.

Overall, it is worth mentioning that the number of past data taken into could be lower or higher than 10. With a value lower than 10, the threshold would be closer to the true values, hence increasing or decreasing more quickly. In contrast, with a value higher than 10, the threshold fluctuations would be even more reduced, but with the risk of slower reactions to significant AR changes. Finally, other methods than the moving average exist for filtering, such as a first or second order filter.

### 3.6 Aspect to Consider in Real Environment

Right after the proxy is started, the integral is set to 0 and the initial threshold (before any control) is also set to 0. Therefore optional contents are not served during the first seconds no matter what is the workload. That is unwanted in case of low utilization. To avoid this problem, Brownout can be deactivated (but the control algorithm setting a threshold is always executed) until $RT_{95}$ is greater than the set-point. As soon as $RT_{95} > \text{setpoint}$, Brownout is activated and optional contents may or not be served according to the threshold set by the controller. Once Brownout is activated, it is never deactivated as the threshold has reached appropriate values. This technique can also be applied when the controller is associated with a machine learning algorithm (Chapter 5). The reason is that machine learning algorithms may not have outputs at the beginning if they have no prior knowledge, implying that during that time, only the controller is used to update the threshold. Although, in this paper, we present results produced with Brownout being always activated. The reason is that the workload is always known, as we specify the concurrency setting, and the first seconds do not matter for the results as we are focused on what the algorithms are capable of when Brownout is activated (given diverse workloads, specified with the concurrency setting).

### 3.7 Discussion

As the feedback ($K_p * error + K_i * integral$) is used to set the threshold, the integral must first accumulate enough errors in order to obtain appropriate thresholds. When the AR significantly changes (such as, in the experiments, setting the concurrency from 200 to 800), the integral must as well change to obtain appropriate thresholds. Indeed, if the integral equals 0, or is close to 0, only the proportional part of the controller ($K_p * error$) would determine the threshold. This is not feasible unless $K_p$ is high enough, combined with a positive error, which can produce appropriate thresholds. It is of course unrealistic as it would be rare and therefore $RT_{95}$ would be far away from the set-point most of the time. Instead, what happens is that $K_i * integral$ is a close value to the threshold (if $K_i = 1$ then the integral itself is actually more or less equal to the threshold) and $K_p * error$ is used as a small reaction to errors in order to adjust the threshold. The integral is therefore the most important component of the controller to obtain appropriate thresholds when only the feedback is used to determine the threshold.

Consequently, the tuning parameters $K_p$ and especially $K_i$, are the ones that determine how fast the controller reacts to significant AR changes. The higher $K_i$ is, the less the integral must increase or decrease when there is a significant AR change. However, with high
$K_p$ and $K_i$, we observed that $RT_{95}$ is less stable when the AR does not significantly change. Therefore, if only the PI controller is used to determine threshold values, a compromise must be made between a stable $RT_{95}$ but slow reactions to significant AR changes, or a less stable $RT_{95}$ but faster reactions to significant AR changes. An investigation of these parameters showed satisfying results with $K_p$ around 6 to 10, and $K_i$ around 2 or 3, as a compromise. In addition, a controller output filter can be added to avoid too large threshold updates that may render $RT_{95}$ less stable around the set-point when the threshold is low on average, as described in Section 3.5. It is also important to notice that tuning $K_p$ and $K_i$ depends on the set-point. All values determined for $K_p$ and $K_i$ in this thesis are based on a set-point equal to 0.5. We observed that with higher set-points, errors are larger, and thus lower $K_p$ and $K_i$ values should be used. According to our observations, there is a proportionality between the set-point and the average error, when the set-point is greater than 0.5. For example, if the set-point is twice as much larger (i.e., 1), then errors are as well twice as much larger on average, hence $K_p$ and $K_i$ should be twice as much lower.
Chapter 4

Machine Learning

Techniques imported from statistical machine learning have shown to be effective for autonomic control in cloud data centers [16]. As the control algorithms we conceived showed satisfying results in some situations (such as $RT_{95}$ being close to the set-point when the AR does not significantly change), learning from these results can lead to improvements, i.e., avoiding slow controller reactions to significant AR changes, or unstable $RT_{95}$. Therefore we investigate another approach to find appropriate thresholds by designing and implementing machine learning algorithms (MLAs). The MLAs described in this chapter rely on offline data including favorable states, which are used to train the MLAs.

4.1 Training with Favorable States

A favorable state is defined as follows. A set of measured data determines a state. A state $s$ can be represented as a tuple: $s := \{RT_{95}, AR, \text{threshold}, \text{queue-length}, \%\text{-optional-content}\}$. For instance, a state $A$ could be that $RT_{95}$ is 0.65 second, the threshold is 15, the AR is 100, and so on. The next state $B$ could be that $RT_{95}$ is 0.52 second, the threshold is 13, the AR is 102, and so on. Given state $A$, the threshold is set to 13, which produces state $B$. As $RT_{95}$ is almost equal to the set-point (0.5) in state $B$, $B$ can be considered a favorable state. We chose to select favorable states when $RT_{95}$ is higher than 0.45 second and lower than 0.55 second, which is a tolerance of 0.05. Of course we could have chosen a smaller tolerance, but then favorable states would have been less frequent. What is learned in this example is that a threshold value 13 is appropriate for a state similar to state $B$, as $B$ is a favorable state.

To be able to train MLAs with appropriate thresholds, the PI controller is run during a special experiment to produce some favorable states, which are stored for offline training. The experiment includes many distinct concurrencies to produce as many favorable states as possible. Then, during an offline training phase, the MLA is trained, hence it can later reproduce favorable states during new experiments. The PI controller is thus only a way to produce favorable states, it is not used for any other purpose in this chapter. Different training methods are used with favorable states found within offline data. For instance, the perceptron tries to classify favorable states to associate them with distinct threshold values.
4.2 Perceptron

The perceptron is a classification algorithm based on an artificial neuron network. A perceptron takes as input a set of features and then outputs a value by the use of weight vectors. For a multi-output perceptron, such as the one implemented in this work, there is a weight vector for each output, and in our case each output is threshold value. The features are measured data by the lighttpd proxy in this work. The weight vectors are initialized with default values (such as zeros), and they are updated while the perceptron is trained with favorable states. As we employ an offline training method, the weight vectors are calculated offline. Then, once the perceptron algorithm is executed online, i.e., during a new experiment, thresholds are determined with the weight vectors given new situations as inputs.

To summarize, there are three necessary phases to be able to run experiments based on the perceptron algorithm:

- The first phase is a pre-training experiment that consists in producing favorable states. For that the PI controller is used and only favorable states are stored to be used offline for the next phase.
- During the second phase the perceptron is trained offline with all the stored favorable states, and its weight vectors are eventually determined.
- Finally the third phase is a new experiment during which the perceptron applies its knowledge, given new situations. For each of these new situations, the perceptron can output a threshold value using its weight vectors.

Selecting relevant features to produce the weight vectors and handle new situations is important. We tried many possibilities with different sets of features. For example, the AR, the average queue-length, and the average number of optional content served, and/or a combination of these. We observed that only the AR really has an impact to determine appropriate thresholds.

However, even though the perceptron could produce decent results (i.e., $RT_{95}$ not being too far away from the set-point on average) for some of our experiments, it does not seem the most appropriate MLA in this context. Indeed only one feature seems necessary – the AR – and quite a lot of favorable states are required to obtain appropriate weight vectors during the training phase. To achieve a good accuracy, there must be many weight vectors in order to obtain the required diversity in threshold values. For example, the weight vectors can correspond to the thresholds 0, 1, 2, 3 up to a maximal measured threshold during the training phase. The higher the maximal measured threshold is, the longer it takes to correctly determine all weight vectors. Moreover, thanks to the training experiments we made, we observed that there is a relation between AR and threshold when the VM is highly utilized, which could make the perceptron – or this type of classification algorithm – less necessary.

4.3 Setting Thresholds with Linear Regression

As the AR seems to be the only parameter of importance for learning appropriate threshold values, we investigated the relation between AR and threshold more closely. Therefore we examined the possibility of a linear regression model.

A simple linear regression model assumes a relationship between two variables. Here the variables are the AR and appropriate threshold associated. We made an experiment
with a PI controller and a slowly increasing concurrency to cover many possible ARs, and we obtained results showing the relationship between AR and threshold. When the AR increases, the threshold increases in a proportional way. However this is only the case when the VM is highly utilized, that is, optional contents are sometimes served and sometimes deactivated. The reason is that, when the VM is lowly utilized, the threshold must be high enough (so optional contents are always served) and when the VM is overloaded the threshold should be 0 (so optional contents are never served). Therefore, in these two cases, the observed thresholds are significantly distinct with the ones corresponding to high utilization.

By taking all the ARs corresponding to appropriate threshold values during high utilization, we used the least square method \cite{35} to obtain a linear model, i.e., an equation of the form \( y = ax + b \). Here, \( y \) is the MLA output (the threshold in this case), \( x \) is the AR, and \( a, b \) are parameters determined by the least square method. To be able to use this method, enough data need to be produced by an experiment using a PI controller to obtain favorable states. Therefore, as previously described with the perceptron, three phases are used to first produce favorable states, then learn from them, and then run a new experiment with the equation. In the new experiment, the thresholds are set with the equation \( \text{threshold} = a \ast AR + b \) where \( a \) and \( b \) are pre-determined thanks to the offline training approach.

\[
y = ax + b
\]

\( a = 0.0432 \) and \( b = 10.1924 \) have been determined with the least square method based on offline training. The results are satisfying and the threshold is quickly updated when the concurrency changes. The VM is always highly utilized in this experiment.

The relationship between AR and threshold values shows satisfying results. However we observed that the value \( a \) determined by linear regression is very sensitive to produce threshold values. If \( a \) is not correctly determined, the threshold may be always slightly

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure41}
\caption{Linear regression, response time and threshold.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure42}
\caption{Linear regression, utilization and optional contents.}
\end{figure}

Figures 4.1 and 4.2 show results produced with the equation \( \text{threshold} = 0.0432 \ast AR + 10.1924 \). The parameters \( a = 0.0432 \) and \( b = 10.1924 \) have been determined with the least square method based on offline training. The results are satisfying and the threshold is quickly updated when the concurrency changes. The VM is always highly utilized in this experiment.

The relationship between AR and threshold values shows satisfying results. However we observed that the value \( a \) determined by linear regression is very sensitive to produce threshold values. If \( a \) is not correctly determined, the threshold may be always slightly
higher or lower than necessary resulting in $RT_{50}$ being always slightly higher or lower than the set-point. Inaccuracies are due to the noisy environment, as there is not always the same number of requests within a time period even with a constant concurrency, because of the Poisson distribution used to generate client request rates. Moreover, in case of low utilization or overload, the equation cannot set appropriate threshold values as previously explained.

### 4.4 Mapping Arrival Rate to Threshold

A simpler way to obtain thresholds given the AR is to map AR to threshold. The key in the map is a measured AR, and the value is the corresponding measured threshold. At first, the map is empty, and it is populated whenever a favorable state is met. With an offline training method, the map is eventually populated with all possible ARs (until a certain upper limit) assuming enough favorable states are produced during the first pre-training phase. During the last phase, that is, a new experiment to test the MLA, the thresholds are determined thanks to the map. The map $M$ is defined as

$$M = [(ar_1, t_1), (ar_2, t_2), ..., (ar_n, t_n)]$$ (4.1)

where $n$ is the total amount of keys/values in the map. $ar_i$ is an AR indexed by $i$ and $t_i$ is the threshold corresponding to $ar_i$. Therefore $M[ar_i] = t_i$. However, due to possible inaccuracies, to obtain a threshold given the AR $ar_i$, the output is not just $t_i$ but

$$\text{median}(\{M[ar_i-K], ..., M[ar_i], ..., M[ar_i+K]\})$$ (4.2)

where $K$ is a parameter determining how many thresholds in the map should be taken into account on each side of $M[ar_i]$. We can see $K$ as a number of neighbors in the map, which is inspired by the K Nearest Neighbors (KNN) algorithm [9]. We select the median of the thresholds taken into account in order to avoid possible inaccuracies. For example, with $K = 2$, the $K*2+1 = 2*2+1 = 5$ thresholds in the map could be $\{10.6, 10.5, 27.4, 11.1, 11.9\}$, with 11.1 the median (rounded to 11). The median is used instead of an average as averages are sensitive to outliers, such as 27.4 in the above example. This type of outliers can occur when the measured ARs are within one of the two transits. The first transit is between low utilization, with very high thresholds, and high utilization, with proportional threshold values with the AR. The second transit is between high utilization, with proportional threshold values with the AR, and overload, with thresholds set to 0 (so optional contents are never served).

Figures 4.3 and 4.4 show the results produced with the mapping of AR to threshold values. The results are satisfying and the threshold is quickly updated when the concurrency changes. The VM is always in highly utilized in the experiment. $K$ was set to 3, and thus a total of $K*2+1 = 3*2+1 = 7$ thresholds – or map entries – were taken into account in the map.

### 4.5 Online versus Offline Training

Offline training is not desirable for two reasons. First, in a real environment (e.g., a web developer wants to use Brownout with a web application) an emulation or simulation should be run onto the web application for the MLA to be trained, which is tedious. Another reason is that the context can change after the offline training, for instance the application can run
in a VM consisting of two (CPU) cores, and later the VM is improved to have eight cores. At this point all previously learned data based on two cores would be useless, and another offline training phase would be necessary.

Online training is therefore necessary, but to learn from favorable states the threshold must be set somehow. For that, a controller can be added along with an MLA until the latter has been properly trained, the controller can eventually produce appropriate threshold values to be learned by the MLA.

4.6 Discussion

As an online training approach is more desirable, MLA and controller should be combined. Before that, we need to select the most appropriate MLA for this purpose. In what follows we discuss the drawbacks of the perceptron and linear regression algorithms, and then we explain why the mapping of AR to threshold seems to be the most appropriate MLA.

The perceptron (Section 4.2), even though it produces decent threshold values with optimal weight vectors, is not considered to be the most suitable approach. The possible high number of weight vectors needed can imply a long training for the weight vectors to be accurate. Moreover we observed that only the AR seems necessary to be taken into account. The linear regression (Section 4.3) showed satisfying results, indicating a proportional relation between AR and threshold when the VM is highly utilized. However threshold values obtained using the linear regression technique are incorrect when the VM is lowly utilized or overloaded. Indeed, as threshold = AR*a + b where a and b are determined by the least square method, the values found for a and b are only suitable when the threshold increases proportionally with the AR. 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 is served). The mapping of AR to threshold can answer this problem.
With the mapping of AR to threshold, the MLA simply sets thresholds corresponding to measured ARs as map entries, in case of favorable states, i.e., when $RT_{95}$ is close to the set-point. Therefore next time a measured AR exists in the map (or is close to an existing one, depending on the parameter $K$), the MLA produces an output. 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 ARs far away from the measured one. According to our observations, $K = 3$ leads to satisfying results. With $K$ being high enough, as soon as an incoming AR is near an existing AR in the map, an output can be obtained. Moreover, when enough entries exist in the map, the linear regression described in Section 4.3 method is still applicable. On the long term, the MLA is trained enough and it outputs appropriate thresholds for any AR, even ARs measured during low utilization or overloads. When the VM is lowly utilized, the threshold is high enough for optional contents to always be served. Conversely, when the VM is overloaded the threshold is low enough (most likely 0) for optional contents to never be served. We observed that the results are roughly the same when the threshold is set by an equation with the linear regression method or by the mapping of AR to threshold. As the mapping of AR to threshold can be used during low/high utilization and overloads, it is suitable to be combined with a control algorithm.
Chapter 5

Controller with Machine Learning

Combining a control algorithm with an MLA can use the best of the two approaches. Indeed the MLA cannot work alone as, at some point, it has to learn from favorable states, and conversely using the controller only implies a compromise between stable $RT_{95}$ and slow reactions to significant AR changes, versus a less stable $RT_{95}$ but faster reactions to significant AR changes. This chapter describes different ways to combine controller and MLA to use the advantages of both approaches to keep $RT_{95}$ stable around the set-point as much as possible. As specified in Section 4.6, the mapping of AR to threshold is the most appropriate MLA as it can be quickly trained and handles all low/high utilization and overload cases. Hence this MLA is combined with the controller, and diverse ways of combining these algorithms are described in this chapter.

5.1 Controller or Machine Learning

A simple approach is to combine controller and MLA but use these in isolation, i.e., the controller outputs are used only when the MLA has no output. In this scenario, at first, the controller sets threshold values so the MLA can learn, and eventually with enough knowledge the MLA is able to handle most, if not all, situations. Both the controller and MLA are implemented and run in parallel. When there is an output from the MLA, the threshold is set to this output. Otherwise it is set to the controller output. This can be viewed as exclusive OR (XOR): either the controller algorithm outputs thresholds without the MLA, or thresholds are obtained only with the MLA.

Figures 5.1 and 5.2 show results produced with the mapping of the AR to threshold values as MLA. The controller was tuned with $K_p = 8$ and $K_i = 2$ to react relatively fast to significant AR changes. The parameter $K$ was set to 3 in order to take at most $3 \times 2 + 1 = 7$ map entries (of the ones that exist), which are used to obtain MLA outputs. Another experiment showed less satisfying results with $K_p = 5$ and $K_i = 1$, values implying slower reactions to significant AR changes. We observe that, for example, around time equals 300 seconds, the threshold significantly changes from one second to another. This is due to the controller setting a certain threshold and the MLA a significantly distinct one right after, or vice versa.
Figure 5.1: Controller or machine learning, response time and threshold.

Figure 5.2: Controller or machine learning, utilization and optional contents.

5.2 Controller and Machine Learning

Another approach is to use MLA outputs as feedforward ($FF$) in the control algorithm. In this case the MLA is part of the PI controller. The final controller output is equal to the feedback ($FB$, with $FB = K_p \times \text{error} + K_i \times \text{integral}$) added to $FF$ (the MLA output).

![PID controller with feedforward diagram]

Figure 5.3: PID controller with feedforward.

Figure 5.3 represents a diagram of a PID controller with feedforward. In the experiments, $K_d$ is set to 0, making the controller a PI controller with feedforward. With the measured $RT_{95}$ subtracted from the set-point an error is calculated. With the error a feedback is determined by adding the $P$, $I$ and $D$ parts (or just $P$ and $I$ parts with $K_d = 0$), and the
feedback is added with the feedforward (the MLA output) so the result of this operation is the threshold. During the process of 1 second, the threshold determines if optional contents shall be served or not, and $RT_{95}$ is measured at the end of the process.

A problem occurs when the feedforward $FF$ is first unknown, i.e., when the MLA has not been trained enough yet, and then is determined for the first time. In this case only $FB$ is used to compute the threshold. To illustrate the problem, let us assume that the optimal threshold should be 20. At first the MLA has no output, therefore $FB$ eventually reaches approximatively 20 and thus the threshold is more or less set to 20. Then the MLA eventually learns, and outputs more or less 20 as well. In this case the threshold equals $FB + FF$, which is approximatively $20 + 20 = 40$. At this point the threshold is way to high and thus $RT_{95}$ is far away from the set-point. To counteract this problem, the previous feedforward value, $FF_{previous}$, is saved, and when $FF_{previous}$ is 0 and $FF > 0$, the integral is reset to 0. With the previous example, at first $FB$ eventually reaches approximatively 20, so $FB = 20$ and $FF = 0$. Then, after another second, $FF = 20$, so the integral is reset as $FF_{previous} = 0$. At this point $FB$ becomes much lower as it is set to only $K_p * error$ because $K_i * integral = K_i * 0 = 0$. Therefore $FB + FF = K_p * error + 20$, which is an appropriate threshold value.

Another problem can appear afterwards, when the MLA does not have an output again. In this case $FF = 0$, and $FB + FF$ would be a way too low value. To counteract this other problem, if the MLA has no output (implying $FF = 0$), then $FF$ is set to $FF_{previous}$. This might not be always accurate as $FF_{previous}$ may have been set for a different AR compared to the current one, but it is better than setting $FF$ to 0.

By setting the threshold to $FB + FF$, the feedforward eventually becomes the principal value determining the threshold. If the threshold is around 20, then $FF$ is approximatively 20, while $FB$ oscillates around 0 – or a low value – to maintain appropriate threshold values (unless the MLA has not learnt yet, as previously described). Consequently $FB$ is used differently, and new problems related to integral windup can appear. When the VM is lowly utilized, the integral keeps increasing, and $FB$ increases as well. This is not a problem when the VM is lowly utilized as the threshold, set to $FB + FF$, is high enough so optional contents are always served. But when the VM is highly utilized, then $FB$ and $FF$ are both high (assuming the MLA has been trained before and thus $FF > 0$) resulting in a too high threshold. In this scenario, to reach appropriate thresholds, the integral must decrease a lot, and as a result $RT_{95}$ becomes far away from the set-point. To counteract this problem, the integral can simply be reset when the VM goes from low utilization to a higher utilization. The same problem can be observed when the VM is overloaded. At this point the integral decreases a lot and thus $FB$ is much lower than 0. When the VM is not overloaded anymore, the integral must increase a lot to obtain appropriate $FB$ values (around 0). To counteract this problem, the integral is reset when the VM goes from overload to high or low utilization.

Figures 5.4 and 5.5 show the results produced for the combination of controller and machine learning, where the MLA is the mapping of AR to threshold. The controller was tuned with $K_p = 5$ and $K_i = 1$. The parameter $K$ was set to 3, implying to take at most $3 * 2 + 1 = 7$ map entries (of the ones that exist) to obtain feedforward values. We can see in the figures that $RT_{95}$ becomes more and more stable around the set-point over time. This is due to the MLA that is trained online and then applies its knowledge to produce appropriate thresholds.
5.3 Dynamic Equation

The linear regression method (Section 4.3) showed satisfying results by setting the threshold to $a \times AR + b$. Although the $a$ value was sensitive and could easily produce inappropriate thresholds due to possible inaccuracies in learned data. The $b$ value was less sensitive. Using the mapping of AR to threshold not only enables a threshold to be obtained given an AR, but it also enables to apply the linear regression method to be used, assuming enough entries in the map. We observed that setting the threshold to $FB \times AR$, where $FB$ is the feedback of the controller, could produce decent results with very low values for $K_p$ and $K_i$, around 0.01 to 0.05. The idea is to use the least square method to find the $b$ value so the threshold is set to $FB \times AR + b$. The equation to set the threshold is dynamic as $FB$ is the controller output and $b$ is set thanks to the least square method once enough entries exist in the map.

We implemented the dynamic equation algorithm as follows. First the feedback ($FB$) of the controller is calculated and used as the $a$ value of the linear equation $\text{threshold} = a \times AR + b$. $b$ is initially set to 0 as the MLA has not been trained yet. Once values enough are in the map, then $b$ is set with the least square method. The more values are in the map, the better the accuracy is to find $b$. Later, the least square method might produce either a slightly different $b$ value, or a significantly different $b$ value. If the difference is significant, then $b$ should be updated. Otherwise, we chose not to update $b$ if the difference is lower than 1. The reason is that $b$ should be fixed while $a$ can oscillate as little as possible to keep appropriate threshold values.

Figures 5.6 and 5.7 show the results produced with the dynamic equation approach. The controller was tuned with $K_p = 0.03$ and $K_i = 0.01$. These values were obtained with an exhaustive approach, that is, trying diverse sets of possible $K_p$ and $K_i$ values and taking the most appropriate. The $b$ value was first set after 10 thresholds were inserted in the map. This is not much, but it enables to show the effect of inaccuracies when $b$ is set too early (the purpose of these results is to show what happens, not to produce the best case
scenario, otherwise $b$ would have been set later for the first time to be more accurate).

Figures 5.8 and 5.9 show the results from the exact same experiment as in Figure 5.6 and Figure 5.7. In Figure 5.8, the $a$ value of the dynamic equation is shown, with $a$ being the feedback of the PI controller. In Figure 5.9, the $b$ value of the dynamic equation is shown, with $b$ being determined by the least square method. At first $b = 0$, and then at time 102 seconds, $b$ is first initialized to 18.882, resulting in a huge spike for $RT_{95}$. This is due the the result of the equation being far away from appropriate thresholds. Eventually $b$ reaches an appropriate value, 11.798, at time 282 seconds, and $a$ almost does not change (around 0.4 second) even though the concurrency changes. This implies that the controller, whose feedback is $a$, does not need to react fast to significant AR changes as the $a$ value remains approximatively the same no matter what is the AR.

## 5.4 Discussion

First we investigated the possibility of using a controller in parallel with an MLA (Section 5.1). When the MLA has an output, then the threshold is set to the MLA output, otherwise it is set to the controller output. The results are not the best we could achieve. That is because the controller and MLA might have significant differences in their output values. When the controller sets a threshold $T_c$, and just after the MLA sets a threshold $T_{mla}$, if $T_c$ and $T_{mla}$ are significantly different, then the next measurement of $RT_{95}$ might be far away from the set-point. The reason could be that, according to how is tuned the controller, its output is not appropriate because the controller either reacts too slowly, or does not keep $RT_{95}$ stable enough. This behavior is unwanted, however the MLA is eventually trained enough so its outputs are used all the time. Therefore, on the long term, using either the controller or MLA can still be a decent approach.

A PID controller, or PI controller in our case, can be extended with a feedforward. We investigated the possibility of using the MLA outputs as feedforward (Section 5.2). The
threshold is set to the feedback added to the feedforward. At first, the MLA has no output and therefore the feedback is the main value to set the threshold. Then, once the MLA has been trained enough, it outputs values set as feedforward, which is added to the feedback. In this case, the main component to determine the threshold is the feedforward, and the feedback compensates possible inaccuracies made by the MLA. On the long term, the MLA is eventually able to always output values, which makes this combination of controller and MLA better than the option with either the controller or MLA. This is thanks to the advantage to have the controller feedback being able to compensate possible inaccuracies produced by the MLA.

Finally we designed a dynamic equation method (Section 5.3) to set the threshold with the equation \( \text{threshold} = a \times AR + b \). The \( a \) value is set by the feedback of the controller. The goal is to obtain a fix value for \( b \) by using the least square method and, meanwhile, permitting \( a \) to oscillate as little as possible with the controller feedback. This process eventually obtains the same results as the linear regression with the MLA alone, but can also be able to set correct thresholds when the VM is lowly utilized or overloaded. When the VM is highly utilized, and assuming the equation is accurate (i.e., \( a \) and \( b \) are well determined), then it does not matter what is the AR, as \( a \) and \( b \) value should remain almost the same. This is the idea behind the dynamic equation approach: the combination of algorithms is robust to significant AR changes while still being able to keep \( RT_{95} \) stable around the set-point.

The algorithms described in Chapter 3, Chapter 4, and this chapter, now need to be evaluated and compared. For that purpose we run a longer experiment including diverse workloads in order to fairly compare the algorithms or combination of algorithms.
Chapter 6

Evaluation

In this chapter we evaluate each algorithm, or combination of algorithms (described in Chapter 5), designed and implemented to improve Brownout. We take into account their performances, compare them between each other and with the initial Brownout, and discuss which ones can be the most suitable.

6.1 Measuring Performance

In order to evaluate each algorithm, we measured their performance as follows. First we measure a total amount of error $error_{total}$. For each measured $RT_{95}$, the error $|setpoint - RT_{95}|$, with $setpoint = 0.5$, is added to the total amount of errors, i.e., $error_{total} = error_{total} + |setpoint - RT_{95}|$. However, when the VM is lowly utilized or overloaded, the error is ignored. The reason is that in case of low utilization, $RT_{95}$ is normally well below the set-point, and it cannot be higher as optional contents are always served. Therefore it is a normal behavior and not an error. In case of overload, $RT_{95}$ is well above the set-point, but this is not an error as optional contents are never served, hence it is normal that $RT_{95}$ is high. The number of times $N$ errors are taken into account is also measured, and the algorithm performance is obtained by dividing the total amount of errors with $N$, which represents the average of errors, i.e., $\frac{error_{total}}{N}$. The average error represents how far $RT_{95}$ is from the set-point during high utilization. For example, an average error of 0.1 means that, on average, $RT_{95}$ is 0.1 second higher or lower than the set-point. Therefore the closer the average error is to 0, the better.

In this thesis, we decided to show relevant calculated performances (determined by the average errors). For that purpose, we used the same settings and ran an experiment for each algorithm or combination of algorithms. The concurrency setting is as follows: 250, 800, 250, 700, 300, 500, 400, 600, 240, 790, 260, 710, 290, 510, 390, 610. The time period for each of these concurrencies is 100 seconds, making a total time of 1600 seconds for each experiment. Figure 6.1 shows this workload with different concurrencies on the y-axis and the time on the x-axis. The first set of 8 concurrencies, from 150 to 600, is then repeated with small differences (from 240 to 610) to see if the MLAs can produce decent outputs even though they could not be trained from the exact same concurrencies earlier in the experiment. All used concurrencies imply highly utilized VMs (no overloads or low utilization). The reason is that, as previously specified, it is not relevant to calculate errors in case of low utilization or overload. Moreover, we are mainly interested in results produced when Brownout is active, that is, when optional contents are served with a probability $p$ such as $0\% < p < 100\%$. 

This choice of concurrencies also allows us to observe a relevant average errors for the linear regression approach with the MLA alone, as this approach does not handle low utilization and overload cases.

6.2 Algorithms Performance

In this section we present the performance of each algorithm, or combination of algorithms, designed and implemented in this work. Then we compare these performances with the one of the initial Brownout, also calculated with our common evaluation experiment.

6.2.1 Controller

As described in Chapter 3, the controller uses its feedback ($K_p \cdot \text{error} + K_i \cdot \text{integral}$) to set the threshold. Low $K_p$ and $K_i$ values assure $RT_{95}$ to be stable around the set-point when there are no significant AR changes, whereas high $K_p$ and $K_i$ values assure fast reactions to significant AR changes.

Table 6.1 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. The average errors are calculated with the common experiment using the settings described in the above section Measuring Performances. The experiment includes significant AR changes, which explain why the controller with high $K_p$ and $K_i$ values produces a smaller average error, as the controller can react fast to these significant AR changes with high $K_p$ and $K_i$ values. For both choices of $K_p$ and $K_i$ values, an open model produce slightly more errors.
### 6.2. Algorithms Performance

#### Table 6.1: Evaluation of controller algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Avg. error, closed model</th>
<th>Avg. error, open model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller, low $K_p$ and $K_i$ values</td>
<td>$K_p = 5, K_i = 1$</td>
<td>0.069</td>
<td>0.071</td>
</tr>
<tr>
<td>Controller, high $K_p$ and $K_i$ values</td>
<td>$K_p = 20, K_i = 5$</td>
<td>0.062</td>
<td>0.064</td>
</tr>
</tbody>
</table>

#### Table 6.2: Evaluation of controller algorithms, concurrency 400 for 1000 seconds.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Avg. error, closed model</th>
<th>Avg. error, open model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller, low $K_p$ and $K_i$ values</td>
<td>$K_p = 5, K_i = 1$</td>
<td>0.048</td>
<td>0.050</td>
</tr>
<tr>
<td>Controller, high $K_p$ and $K_i$ values</td>
<td>$K_p = 20, K_i = 5$</td>
<td>0.058</td>
<td>0.059</td>
</tr>
</tbody>
</table>

Table 6.2 shows another 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. This time, the average errors are calculated with a separate workload from the one in Table 6.1, here the concurrency is always 400 (implying a highly utilized VM). The goal of this experiment is to investigate $RT_{95}$ oscillations and ensure that low $K_p$ and $K_i$ give a more stable $RT_{95}$ than high $K_p$ and $K_i$ values when there are no significant AR changes. We ignored the first 100 seconds, as at the beginning of the experiments the integral must accumulate enough errors to produce appropriate thresholds, especially with low $K_p$ and $K_i$ values. As we can see the average error for low $K_p$ and $K_i$ values is lower than the average error for high $K_p$ and $K_i$ values. This confirms that low $K_p$ and $K_i$ values imply a more stable $RT_{95}$ than with high $K_p$ and $K_i$ values, when the AR does not significantly change.

#### 6.2.2 Machine Learning

As described in Chapter 4, we investigated three different MLAs. The perceptron was not considered to be a suitable approach, the linear regression was not appropriate in case of lowly utilized or overloaded VM, and the mapping of AR to threshold showed satisfying results overall.

#### Table 6.3: Evaluation of MLAs based on offline learning.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Avg. error, closed model</th>
<th>Avg. error, open model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear regression</td>
<td>threshold = $0.0432 \times AR + 10.1924$</td>
<td>0.059</td>
<td>0.056</td>
</tr>
<tr>
<td>Mapping of AR to threshold</td>
<td>$K = 3$, implying to take 7 map entries</td>
<td>0.060</td>
<td>0.062</td>
</tr>
</tbody>
</table>

Table 6.3 shows the comparison between the linear regression and the mapping of AR to threshold. The MLAs are based on offline data produced during distinct experiments. The average errors are calculated with the common experiment using the settings described in
Section 6.1. As we can see, both algorithms lead to similar average errors, which confirm that the mapping of AR to threshold is the most appropriate algorithm as it handles the cases of low utilization and overload. These results are better than the ones for the controller, however they are based on an offline training which enables the MLA to output appropriate threshold most of the time, unlike the controller that has to adapt to AR changes.

6.2.3 Controller with Machine Learning

As described in Chapter 5, we investigated three different ways to combine the controller with the mapping of AR to threshold as MLA. First we used either the controller or MLA to set the threshold, then we used MLA outputs as feedforward for the controller, and finally we employed a dynamic equation approach.

Table 6.4: Evaluation of controller with MLAs.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Description</th>
<th>Avg. error, closed model</th>
<th>Avg. error, open model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Either controller or MLA</td>
<td>Controller: $K_p = 8$ and $K_i = 2$; MLA: $K = 3$, implying to take at most 7 map entries in the map to compute the threshold</td>
<td>0.081</td>
<td>0.080</td>
</tr>
<tr>
<td>Controller with feedforward as MLA output</td>
<td>Controller: $K_p = 8$ and $K_i = 2$; MLA: $K = 3$, implying to take at most 7 map entries to compute the feedforward</td>
<td>0.064</td>
<td>0.064</td>
</tr>
<tr>
<td>Dynamic equation</td>
<td>Controller: $K_p = 0.03$ and $K_i = 0.01$; MLA: waits 20 map entries to set $b$ from the map</td>
<td>0.070</td>
<td>0.076</td>
</tr>
</tbody>
</table>

Table 6.4 shows a comparison between either the controller or MLA, the controller with feedforward as MLA output, and the dynamic equation. The average errors are calculated with the common experiment using the settings described in Section 6.1. The choice of either the controller or MLA gives the worst results, making this approach less suitable than the others. The average error for the controller with feedforward as MLA output and the dynamic equation are more satisfying. With the dynamic equation, when the $b$ value is initially set, $RT_{95}$ can be far away from the set-point as the controller needs to adjust its feedback, the $a$ value. Better results can be achieved with the controller with feedforward as MLA output in this evaluation experiment.

Overall, it is fair to notice that MLAs start with a handicap: they have no knowledge to begin with, and must be trained during the experiment. Therefore the results are satisfying, even though they produce higher average errors than the controller algorithms alone previously described. One possibility to counteract this problem is to use MLA outputs only once the map is populated with a minimal number of entries. It it already the case with the dynamic equation approach that waits for 20 entries, as the least square method, which outputs the $b$ value, requires data – here ARs corresponding to thresholds – to output relevant $b$ values. The two other algorithms – either the controller or MLA and the controller with feedforward as MLA output – could also wait a certain number of entries before having
an impact to determine thresholds. We chose not to do that in these common evaluation experiments as it would not be fair. Indeed, if the MLA outputs start to be used in the middle of the experiment, then, as the next set of concurrency is approximatively the same with the first part, the average errors would most likely be optimal. In contrast, if the MLA outputs start to be used before or after the middle of the experiment, then average errors would not be optimal, and they could be worst than the one described in Table 6.4. As a result of doing this, the average errors are approximations, that is, not statistically significant as repeating the same experiment leads to significantly different average errors (we observed up to a 12% difference by repeating several times the same experiment). However what should be retained is not the exact performance of these combination of algorithms, but the fact that they can produce satisfying results even if the MLAs start to be used with minimal training.

6.2.4 Initial Brownout

As described in Section 2.1, the initial Brownout consists in a controller taking into account \( RT_{95} \) alone. Response times are often unstable and they sometimes present spikes due to sudden changes in the workload, which implies that \( RT_{95} \) is often far away from the set-point. We use the same common evaluation experiment with the initial Brownout to compare it with the new algorithms described in this thesis. By running the common experiment with the initial Brownout, we obtain an average error of 0.394 second for the close model, and an average error of 0.416 second for the open model. Even by not taking into account MLAs without controller performances (MLAs alone being not realistically usable in real environments, as described in Section 4.5), some of the best performances achieved with the new Brownout algorithms of this work lead to at least six times lower average errors, which is a six-fold improvement over the initial Brownout performance. 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 common evaluation settings; 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, the new Brownout algorithms may perform differently, therefore we can talk about an improvement by a factor up to 6, given our experimental results.

6.3 Advantages and Drawbacks

All algorithms we designed and implemented have certain advantages and drawbacks. With controller algorithms alone, the \( RT_{95} \) stability mainly depends on how the parameters \( K_p \) and \( K_i \) are tuned. With MLAs alone, the offline training method is what determines how optimal the threshold values will be. With the controller and/or MLA, the MLA uses an online training method that has an important impact over time to obtain appropriate thresholds. Therefore critical moments occur when the MLAs have not been trained enough, or when they produce inaccuracies.

Table 6.5 shows the advantages and drawbacks of all algorithms (or combination of algorithms) presented in this thesis. First the controller algorithms alone are described, then the MLAs alone based on offline training, and then the controller with MLAs. It is not specified in the table, but of course the MLAs alone have the main drawback of not being realistically usable as offline training is not desirable in real environments.
### Table 6.5: Evaluation of advantages and drawbacks of all algorithms.

<table>
<thead>
<tr>
<th>Algorithms ; Average errors for closed / open models</th>
<th>Advantages</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller, low $K_p$ and $K_i$ values ; 0.069 / 0.071</td>
<td>Keeps $RT_{95}$ stable when there are no significant AR changes</td>
<td>Does not react fast in case of significant AR changes</td>
</tr>
<tr>
<td>Controller, high $K_p$ and $K_i$ values ; 0.062 / 0.064</td>
<td>Reacts fast to significant AR changes</td>
<td>Does not keep $RT_{95}$ stable enough when there are no significant AR changes</td>
</tr>
<tr>
<td>Linear regression (offline training) ; 0.059 / 0.056</td>
<td>In case of high utilization, $RT_{95}$ is stable with a well determined equation</td>
<td>Does not support the cases of low utilization and overload</td>
</tr>
<tr>
<td>Mapping AR to threshold (offline training) ; 0.060 / 0.062</td>
<td>Handles low/high utilization and overload cases, and, overall, $RT_{95}$ is stable enough</td>
<td>Possible inaccuracies in the map (but not often significant)</td>
</tr>
<tr>
<td>Either controller or MLA ; 0.081 / 0.080</td>
<td>The controller helps keeping decent threshold values while the MLA is learning</td>
<td>May often produce bad results when the MLA has not been trained enough (and is used in this case)</td>
</tr>
<tr>
<td>Controller with feedforward as MLA output ; 0.064 / 0.064</td>
<td>Once the MLA has been trained enough, the controller feedback helps to prevent inaccuracies from the MLA in order to obtain appropriate thresholds</td>
<td>May produce some bad results when the MLA has not been trained enough (and is used in this case)</td>
</tr>
<tr>
<td>Dynamic equation ; 0.070 / 0.076</td>
<td>Once the equation is accurate, $RT_{95}$ is stable even in case of significant AR changes</td>
<td>Need to wait enough entries in the map to obtain decent $b$ values with the MLA</td>
</tr>
</tbody>
</table>

### 6.4 Most Suitable Algorithm

Given all the designed and implemented algorithms, if we had to choose one to be deployed in a real environment, the choice would most likely depend on the application type. Assuming an application where unexpected peaks of requests can frequently occur, a combination of a controller with an MLA would probably be the most suitable method. Otherwise the controller feedback alone would react too slowly to unexpected peaks, unless it is tuned with high $K_p$ and $K_i$ values, but in this case $RT_{95}$ would not be stable enough when there are no significant AR changes. Therefore combining an MLA with a controller would enable fast reactions to these peaks while keeping $RT_{95}$ stable enough. 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 rare, a simple feedback controller would suffice. Tuned with reasonably low $K_p$ and $K_i$ values, the controller would keep $RT_{95}$ stable enough, and it would not need to react
fast to significant AR changes as unexpected peaks would not be frequent. Even if the AR keeps increasing or decreasing over long periods of time, the controller would be able to keep satisfying $RT_{95}$ as the AR changes would not occur within a few seconds. In addition, a controller output filter can be added to prevent the threshold from fluctuating too much, which is particularly useful when the threshold is low on average, as described in Section 3.5.
Chapter 7

Conclusion

To improve the Brownout paradigm, we investigated a new approach based on the queue-length of pending requests. We designed and implemented algorithms to output thresholds deciding whether or not optional contents should be served. First we developed a PI controller to update the threshold each time period of 1 second. The PI controller showed satisfying results, but improvements were needed as the controller, in some cases, did not react fast enough to significant AR changes, and in other cases $RT_{95}$ was not stable enough. Then, based on data that the controller can produce – namely favorable states, we developed MLAs. We investigated three main algorithms: perceptron, linear regression, and mapping AR to threshold. We concluded that mapping AR to threshold was the most suitable to handle every cases. As MLAs must somehow learn from existing and relevant data, we implemented algorithms using both a controller and MLAs with data learned online. Finally we evaluated all the algorithms, or combination of algorithms, and we compared them, and found that the new Brownout algorithms can improve the initial Brownout by a factor up to 6. The PI controller alone is suitable for real environments, but when frequent unexpected peaks of requests occur, it seems preferable to combine a controller with an MLA to quickly react to any situation, assuming the MLA has been trained enough.

7.1 Limitations

All experiments were based on a single web page of the RUBiS web application. On this web page, related products (the optional content) either are shown, or not. In a real environment, different web pages (assuming a web application, but it does not have to be) can contain optional contents. For instance a page $A$ may serve optional contents taking $X$ second(s) to be computed with $N$ concurrent users frequently sending requests, and a page $B$ may serve optional contents taking $Y$ second(s) to be computed with the same concurrency $N$. If $X$ and $Y$ differ significantly, then results regarding how far $RT_{95}$ is from the set-point may differ and lead to performance degradations. Indeed the current version of Brownout does not take into account the response time for specific requests but for all requests.

7.2 Lessons Learned

We observed that $RT_{95}$ can be less stable with a one core CPU than an eight cores CPU (although adding a controller output filter can reduce this lack of stability). We noticed
that, with one core, the queue-length is, on average, lower than with eight cores. This might explain the reason for $RT_{95}$ to be less stable with only one core. Another reason could be that, with only one core – or just a few cores, fewer requests can be handled in parallel, which makes $RT_{95}$ less stable as well.

Machine learning algorithms are trained with previously produced data. These data are produced within a certain context. A context is, for example, the number of CPU cores and other hardware configurations. In cloud data centers, the context can change, for instance a VM with two cores might be upgraded with eight cores. When these context changes happen, all previously learned data with two cores may no longer be useful, and worst, they could lead to $RT_{95}$ being far from the set-point or unstable. To avoid this problem, a possible approach would be to detect hardware modifications and delete, or store somewhere, all previously learned data to retrain the MLA. Meanwhile, another algorithm, in this case a controller, would keep appropriate threshold values. However this approach may not be enough, as software modifications might as well change the context. For instance, an SQL query might take on average 0.6 second with a certain amount of concurrent users, and be later modified (by a developer) so it would take only 0.2 second on average with the same amount of concurrent users. The consequences of this significant SQL query modification could be that the MLA no longer has correct outputs, and thus must be retrained.

### 7.3 Future Work

The initial brownout has been tested with load balancing algorithms [25]. The new version of Brownout could also be tested to see if the new algorithms based on the queue-length of pending requests do not degrade performances with load balancing algorithms. Both Brownout versions avoid overloads, but the new version described in this thesis also permits to obtain more stable response times. This effect could alter results obtained with the load balancing algorithms, therefore it is worth investigating the new Brownout action with the load balancing algorithms.

In cloud data centers, elasticity enables operators to provide or withdraw resources autonomically, to match demand to the available resources as close as possible [22]. For this purpose, autoscaling methods are applied to start new VMs to avoid overloads commonly by monitoring server utilization or response times. If Brownout is used within a cloud computing system including an autoscaling algorithm, then the autoscaling algorithm might produce unwanted behaviors. The reason is that Brownout enables to avoid overloads until a certain point, but it degrades the user experience, which is unwanted as users should both be pleased by responsive applications and by everything the applications can offer such as, for example, product recommendations (the optional content). 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. This would imply that new VMs are started to avoid overloads and optional contents would always be served unless there is really no more available resources, or during the time new VMs are booting, which can take several minutes. Overall, diverse autoscaling algorithms exist with techniques based on control theory, reinforcement learning, queuing theory, time-series analysis, or simple static threshold-based rules [28].
Chapter 8

Acknowledgements

I would like to particularly thank Johan Tordsson and Cristian Klein for their precious help and guidance. I really enjoyed our discussions, and their advices and suggestions were always useful to keep me on the right path. I would also like to thank Karl-Erik Årzen for his advice about control theory, as well as everyone who has worked on the initial version of Brownout and related works. Finally, I want to thank my friends and family for their support.
References


Appendix A

Source Code

The source code below, written in C, permits to use Brownout with a proxy, such as lighttpd in our case. A proxy implies that there are one or several replicas (e.g., virtual machines) where incoming requests are sent. In our case we used one replica, but the source code is conceived to enable several replicas, all of them using Brownout algorithms. To avoid to present non interesting source code, only the header (.h) files are shown, as well as the MLAs and controller algorithms respectively in files brownout_ml.c and brownout_algorithms.c.

——— FILE brownout.h ————

```c
#ifndef _BROWNOUT_H_
#define _BROWNOUT_H_

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "dynamic_array.h"
#include "brownout_ml.h"
#include "brownout_statistics.h"
#include "brownout_algorithms.h"

/*
 * Just for code clarity, to avoid long lines of code
 */
#define BR brownoutReplicas[indexReplica]

/*
 * Variables used for machine learning algorithms, for each replica
 */
typedef struct BrownoutMachineLearning {
    DynamicArray * map;
    int numInsertedValue;
    int minARForHighUtil;
    int maxARForHighUtil;
    int numNeighbors;
    double b;
} BrownoutMachineLearning;

/*/
* Variables used for statistics, for each replica
*/
typedef struct BrownoutStatistics {
    double rt95;
    double prevRt95;
    int ar;
    double arEMA;
    double filter;
    int filterCounter;
    double tmpArEMA;
    int numPeriodEMA;
    int indexEMA;
    DynamicArray * responseTime;
    DynamicArray * queueLength;
    DynamicArray * optionalContent;
    int minQueueLength;
    int maxQueueLength;
    double rtAVG;
    double qlAVG;
    double ocAVG;
    double prevRtAVG;
    double prevQlAVG;
    double prevOcAVG;
    double ocEMA;
    double tmpOcEMA;
} BrownoutStatistics;

/*
* Variables used for brownout algorithms, for each replica
*/
typedef struct BrownoutReplica {
    double threshold;
    double previousThreshold;
    double integral;
    double previousError;
    double previousFeedforward;
    int state;
    BrownoutStatistics * stats;
    BrownoutMachineLearning * ml;
    double feedback;
    double feedforward;
    int controlIndex;
    short isBrownoutActivated;
} BrownoutReplica;

/*
* Replicas as global variable, defined in brownout.c
*/
extern BrownoutReplica * brownoutReplicas;

/*
* Total number of brownout replicas as global variable, defined in brownout.c
*/
extern int brownoutTotalReplica;

/*
* Initialize struct variables for one brownout replica
*/
void brownoutInitializeReplica(int indexReplica);

/
* Initialize struct variables for all brownout replicas */
void brownoutInitializeReplicas(int fromIndexReplica);

/*
* Check if all or some replicas have not be initialized
*/
void brownoutCheckBrownoutReplicas(int indexReplica);

/*
* ***** Must be called by the web server *****
* Return a string which should be set as web server header to know if
* optional contents shall be served or not
*/
char * brownoutDecision(int indexReplica, int queueLength);

/*
* ***** Must be called by the web server *****
* Control and update threshold of one replica, therefore it must be called
* by all replicas
*/
void brownoutControl(int indexReplica, double timePeriod);

#endif

——— FILE brownout_algorithms.h ————

#ifndef _BROWNOUT_ALGORITHMS_H_
define _BROWNOUT_ALGORITHMS_H_
#include "brownout.h"

/*
* Brownout Algorithm
*/
define BROWNOUT_ALGORITHM_NONE 0
#define BROWNOUT_ALGORITHM_FEEDBACK 1
#define BROWNOUT_ALGORITHM_FEEDBACK_PLUS_FEEDFORWARD 2
#define BROWNOUT_ALGORITHM_DYNAMIC_EQUATION 3
#define BROWNOUT_ALGORITHM_CONTROLLER_XOR_ML 4
#define BROWNOUT_ALGORITHM_OFFLINE_TRAINING 5
#define BROWNOUT_ALGORITHM /* one of the above: */
                  BROWNOUT_ALGORITHM_FEEDBACK

/*
* Activate (1) or not (0) the filter on controller output (threshold)
*/
define BROWNOUT_FILTER_ACTIVATED 0

/*
* Optimize CPU or response time
*/
define BROWNOUT_OPTIMIZE_NONE 0
#define BROWNOUT_OPTIMIZE_CPU_UTILIZATION 1
#define BROWNOUT_OPTIMIZE_RT95 2
#define BROWNOUT_OPTIMIZE /* one of the above: */ BROWNOUT_OPTIMIZE_NONE
#define BROWNOUT_OPTIMIZE_K 5 // K as the error multiplicator
#include "brownout_algorithms.h"

double brownoutTreatError(double error) {
  if (BROWNOUT_OPTIMIZE == BROWNOUT_OPTIMIZE_CPU_UTILIZATION && error > 0) {
    error *= BROWNOUT_OPTIMIZE_K;
  } else if (BROWNOUT_OPTIMIZE == BROWNOUT_OPTIMIZE_RT95 && error < 0) {
    error *= BROWNOUT_OPTIMIZE_K;
  }
  return error;
}

double brownoutFilter(int indexReplica) {
  double thresholdEMA;
  if (BR.stats->filterCounter < BR.stats->numPeriodEMA) { // simple moving average
    BR.stats->filter += BR.threshold;
    BR.stats->filterCounter++;
    thresholdEMA = BR.stats->filter / BR.stats->filterCounter;
    if (BR.stats->filterCounter == BR.stats->numPeriodEMA) {
      BR.stats->filter = thresholdEMA;
    }
  } else { // EMA
    BR.stats->filter = (BR.threshold - BR.stats->filter) * 
      (2.0 / (BR.stats->numPeriodEMA + 1.0)) + 
      BR.stats->filter;
    thresholdEMA = BR.stats->filter;
  }
  return thresholdEMA;
}

void brownoutHandleFeedforward(int indexReplica) {
BR.feedforward = brownoutMachineLearningMapping(indexReplica, (double)
BR.stats->ar);

if (BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_FEEDBACK_PLUS_FEEDFORWARD) {
    if (BR.previousFeedforward >= 0 && BR.feedforward < 0) {
        BR.feedforward = BR.previousFeedforward;
    } else if (BR.previousFeedforward < 0 && BR.feedforward >= 0) {
        BR.integral = 0;
    }
    if (BR.state < 0) {
        BR.integral = 0;
        BR.state = 0;
    }
    BR.previousFeedforward = BR.feedforward;
}

void brownoutAlgorithm(int indexReplica, double timePeriod) {

    // set - point
    static const double SET_POINT = 0.5; // 1.0;

    // run the algorithm only when Brownout is activated
    if ( ! BR.isBrownoutActivated) {
        if (BR.stats->rt95 > SET_POINT)
            BR.isBrownoutActivated = 1;
        else
            return;
    }

    // tuning parameters (K_D = 0.0 means that a PI controller is used)
    static const double K_P = (BROWNOUT_ALGORITHM ==
        BROWNOUT_ALGORITHM_DYNAMIC_EQUATION) ? 0.03 : 8.0;
    static const double K_I = (BROWNOUT_ALGORITHM ==
        BROWNOUT_ALGORITHM_DYNAMIC_EQUATION) ? 0.01 : 2.0;
    static const double K_D = (BROWNOUT_ALGORITHM ==
        BROWNOUT_ALGORITHM_DYNAMIC_EQUATION) ? 0.0 : 0.0;

    // determine the error
    double error = SET_POINT - BR.stats->rt95;
    error = brownoutTreatError(error);

    // set feedforward
    if (BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_CONTROLLER_XOR_ML ||
        BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_FEEDBACK_PLUS_FEEDFORWARD)
        brownoutHandleFeedforward(indexReplica);

    // integral part
    double newIntegral = BR.integral + error * timePeriod;

    // derivative part
    double derivative = (error - BR.previousError) / timePeriod;
    BR.previousError = error;

    // set feedback
    BR.feedback = K_P * error + K_I * newIntegral + K_D * derivative;

    // set final output
double finalOutput;
if (BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_FEEDBACK) {
    finalOutput = BR.feedback;
}
else if (BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_OFFLINE_TRAINING) {
    static double prevFinalOutput = 0;
    finalOutput = brownoutMachineLearningMapping(indexReplica,
        BR.stats->ar);
    if (finalOutput <= 0) {
        finalOutput = prevFinalOutput;
    }
    prevFinalOutput = finalOutput;
}
else if (BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_CONTROLLER_XOR_ML) {
    finalOutput = BR.feedforward >= 0 ?
        BR.feedforward : BR.feedback;
}
else if (BROWNOUT_ALGORITHM ==
    BROWNOUT_ALGORITHM_PLUS_FEEDFORWARD) {
    finalOutput = BR.feedback + (BR.feedforward < 0 ? 0 :
        BR.feedforward);
}
else if (BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_DYNAMIC_EQUATION) {
    if (BR.ml->numInsertedValue > 20) {
        double bValue =
            brownoutMachineLearningLeastSquareMethod(indexReplica, (int)
                BR.stats->arEMA, 2);
        if (abs(bValue - BR.ml->b) > 1) {
            BR.ml->b = bValue;
        }
    }
    finalOutput = BR.feedback * BR.stats->arEMA + BR.ml->b;
} else {
    // no algorithm, i.e., brownout not active
}

// set the threshold in case of negative saturation
if (finalOutput < 0) {
    BR.threshold = 0;
}
// set the threshold in case of positive saturation, with 5 being the
// minimal value for the maximal boundary
else if (finalOutput > (BR.stats->arEMA < 5 ? 5 : BR.stats->arEMA)) {
    BR.threshold = BR.stats->arEMA;
}
// set the threshold when there is no saturation
else {
    BR.threshold = finalOutput;
    // update integral
    BR.integral = newIntegral;
}

if (BROWNOUT_FILTER_ACTIVATED) {
    BR.threshold = brownoutFilter(indexReplica);
}
#ifndef _BROWNOUT_ML_H_
#define _BROWNOUT_ML_H_

#include "brownout.h"

/*
 * Activate machine learning debug or not (boolean, 0 or 1)
 */
#define BROWNOUT_ML_DEBUG 1

/*
 * Where to write the debug, e.g., stdout or stderr
 */
#define BROWNOUT_WHERE_WRITE_ML_DEBUG stderr

/*
 * Initialize the BrownoutMachineLearning for a brownout replica
 */
void brownoutMachineLearningInitializeReplica(int indexReplica);

/*
 * Apply the least square method. Assuming the linear equation y = ax + b,
 * if returnValue = 1 then a is returned, if returnValue = 2 then b is
 * returned, otherwise it is y
 */
double brownoutMachineLearningLeastSquareMethod(int indexReplica, int ar,
                                                int returnValue);

/*
 * Given an arrival rate ar, return a computed threshold with the map
 */
double brownoutMachineLearningMapping(int indexReplica, int ar);

/*
 * The machine learning algorithm learns given an arrival rate and the
 * threshold found
 * by the brownout replica (indexReplica)
 */
void brownoutMachineLearns(int indexReplica, int ar);
#endif

#include "brownout_ml.h"

void brownoutMachineLearningInitializeReplica(int indexReplica) {
  BR.ml->map = (DynamicArray *) malloc(sizeof(DynamicArray));
  DynamicArray_init(BR.ml->map, 2048, -1);
  BR.ml->numInsertedValue = 0; // total and unique number of insertions in
  the map
  BR.ml->minARForHighUtil = INT_MAX; // for linear regression (only
  possible during "high utilization")
  BR.ml->maxARForHighUtil = 0; // for linear regression (only possible
  during "high utilization")
}
BR.ml->numNeighbors = 3; // 3 neighbors, "K = 3", implying at most K * 2 + 1 = 7 map entries taken into account
BR.ml->b = 0; // b value for the dynamic equation

if (BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_OFFLINE_TRAINING) {
    /* populate the map with offline training data here (not shown as it takes too much space) */
}

double brownoutMachineLearningLeastSquareMethod(int indexReplica, int ar, int returnValue)
{
    double x1 = 0, x12 = 0, y1_ = 0, x1y1 = 0, a, b, e, n = 0; // n is number of values taken into account, init at 0

    for (int i = BR.ml->minARForHighUtil; i <= BR.ml->maxARForHighUtil; i++)
    {
        if (DynamicArray_get(BR.ml->map, i) >= 0) {
            x1 += ((double) i);
            y1_ += DynamicArray_get(BR.ml->map, i);
            x1y1 += (((double) i) * DynamicArray_get(BR.ml->map, i));
            x12 += ((double) (i * i));
            n++;
        }
    }

    a = (n * x1y1 - x1 * y1_) / (n * x12 - x1 * x1);
    b = y1_ / n - a * x1 / n;

    e = DynamicArray_get(BR.ml->map, BR.ml->minARForHighUtil) - b - a * BR.ml->minARForHighUtil;
    b += e;

    if (returnValue == 1) {
        return a;
    } else if (returnValue == 2) {
        return b;
    }
    return ar * a + b;
}

double brownoutMachineLearningMapping(int indexReplica, int ar) {
    if (ar >= 0 && ar < BR.ml->map->totalSize) {
        double valuesToTakeIntoAccount[BR.ml->numNeighbors * 2 + 1];
        int len = 0;
        for (int i = ar - BR.ml->numNeighbors; i <= ar + BR.ml->numNeighbors; i++) {
            if (DynamicArray_get(BR.ml->map, i) >= 0) {
                valuesToTakeIntoAccount[len++] = DynamicArray_get(BR.ml->map, i);
            }
        }
        if (len == 0)
            return -1;
        // sort
for (int i = 0; i < len; i++) {
    for (int j = i + 1; j < len; j++) {
        if (valuesToTakeIntoAccount[i] > valuesToTakeIntoAccount[j]) {
            double tmp = valuesToTakeIntoAccount[i];
            valuesToTakeIntoAccount[i] = valuesToTakeIntoAccount[j];
            valuesToTakeIntoAccount[j] = tmp;
        }
    }
}

// return median
if (len % 2 == 0) {
    return round((valuesToTakeIntoAccount[len / 2 - 1] +
                  valuesToTakeIntoAccount[len / 2]) / 2.0);
} return round(valuesToTakeIntoAccount[len / 2]);
return -1;

void brownoutMachineLearns(int indexReplica, int ar) {

    // no need to learn if already offline training
    if (BROWNOUT_ALGORITHM == BROWNOUT_ALGORITHM_OFFLINE_TRAINING) {
        return;
    }

    // extend the map if necessary
    while (ar >= BR.ml->map->totalSize) {
        DynamicArray_extend(BR.ml->map);
    }

    // online learning
    if (true) {
        // high utilization
        (0.45 < BR.stats->rt95 && BR.stats->rt95 < 0.55 &&
          BR.stats->ocAVG < 1 && BR.stats->ocAVG > 0 &&
          BR.stats->prevOcAVG < 1 && BR.stats->prevOcAVG > 0) ||
        // low utilization
        (0.3 < BR.stats->rt95 && BR.stats->ocAVG == 1 &&
          BR.stats->prevOcAVG == 1) ||
        // overload
        (BR.stats->rt95 > 1.5 && BR.stats->ocAVG == 0 &&
          BR.stats->prevOcAVG == 0)) {
            // ---- first update min/max if necessary ---- //
            // low util
            if (BR.stats->ocAVG == 1) {
                if (BR.ml->minARForHighUtil < ar)
                    BR.ml->minARForHighUtil = ar;
            }
            // overload
            else if (BR.stats->ocAVG == 0) {
                if (BR.ml->maxARForHighUtil > ar)
                    BR.ml->maxARForHighUtil = ar;
            }
            // high util
            else {
                if (BR.ml->minARForHighUtil > ar)
                    BR.ml->minARForHighUtil = ar;
                if (BR.ml->maxARForHighUtil < ar)
                    BR.ml->maxARForHighUtil = ar;
            }
        }
    }
}
// ---- then update the map ---- //

// if new value
if (DynamicArray_get(BR.ml->map, ar) < 0) {
    BR.ml->numInsertedValue++;
}
// update the map
DynamicArray_set(BR.ml->map, ar, BR.threshold);
// debug
if (BROWNOUT_ML_DEBUG) {
    fprintf(BROWNOUT_WHERE_WRITE_ML_DEBUG, "[B]-new-map-entry-:
        map[%d] = %.3lf;
", ar, BR.threshold);
}
}

———– FILE brownout_statistics.h ———–

#ifndef _BROWNOUT_STATISTICS_H_
define _BROWNOUT_STATISTICS_H_
#include "brownout.h"

/* Where to write the statistics, e.g., stdout or stderr */
define BROWNOUT_WHERE_WRITE_STATISTICS stderr

/* Initialization the BrownoutStatistics struct of a replica */
void brownoutStatisticsInitializeReplica(int indexReplica);

/* Reset statistic variable for a replica, should be used at the end of each control */
void brownoutResetReplicaCounters(int indexReplica);

/* ***** Must be called by the web server *****
* Measure the response time */
void brownoutMeasurementResponseTime(int indexReplica, double responseTime);

/* ***** Must be called by the web server *****
* Measure the arrival rate */
void brownoutMeasurementArrivalRate(int indexReplica);

/* Measurement of the variable determining if optional contents shall be served or not */
void brownoutMeasurementOptionalContent(int indexReplica, double withOptional);
/*
 * Measurement the current queue length
 */
void brownoutMeasurementQueueLength(int indexReplica, int queueLength);

/*
 * Start statistic calculations, should be used before the brownout
 * algorithm execution for each control.
 * Returns a boolean, if false then no control is necessary (case of 0
 * request during the last time period)
 */
short brownoutGoStatistics(int indexReplica);

/*
 * Write statistics somewhere, such as stdout or stderr
 */
void brownoutWriteStatistics(int indexReplica);
#endif

———– FILE dynamic_array.h ———–

#ifndef _DYNAMIC_ARRAY_H_
define _DYNAMIC_ARRAY_H_
#include <stdlib.h>

typedef struct DynamicArray {
    double * array;
    int currentSize;
    int totalSize;
    int emptyValue;
} DynamicArray;

void DynamicArray_init(DynamicArray *da, int initialTotalSize, double emptyValue);
void DynamicArray_extend(DynamicArray *da);
void DynamicArray_append(DynamicArray *da, double value);
double DynamicArray_get(DynamicArray *da, int i);
void DynamicArray_set(DynamicArray *da, int i, double value);
void DynamicArray_sort(DynamicArray *da);
double DynamicArray_sum(DynamicArray *da);
void DynamicArray_empty(DynamicArray *da);
void DynamicArray_resetSize(DynamicArray *da);
void DynamicArray_free(DynamicArray *da);
#endif
Appendix B

Source Code Integration

The source code, written in C, can be included within a web server such as lighttpd (used as proxy in this work), which must be able to calculate response times of treated requests. To integrate the source code, the process is described in a readme.txt file, as follows.

```c
// 1// Include brownout:
#include "brownout.h"

// 2// For each replica indexed by indexReplica, measure the response time
// (for each treated request):
brownoutMeasurementResponseTime(indexReplica, /* response time value here */);

// 3// For each replica indexed by indexReplica, measure the arrival rate
// (for each incoming request):
brownoutMeasurementArrivalRate(indexReplica);

// 4// For each replica indexed by indexReplica, take a decision before a
// request is to be treated:
brownoutDecision(indexReplica, /* queue length value here */);

// brownoutDecision returns a string (char *), which should be used as
// header.
// For example with lighttpd web server:
proxy_set_header(con, "X-With-Optional",
brownoutDecision(indexReplica, /* queue length value here */));

// 5// For each replica indexed by indexReplica, execute the control function:
brownoutControl(indexReplica, /* time period value in seconds here */);

// The time period depends when each control is executed (e.g., every second)
```