Assessing the Performance Overhead of a Self-Adaptive System

Vitor Silva, João M. Franco, Francisco Correia, Raul Barbosa, and Mário Zenha-Rela

CISUC - Centre for Informatics and Systems
University of Coimbra, Portugal
vhsilva@student.dei.uc.pt
{jmfranco,fcorreia,rbarbosa,mzrela}@dei.uc.pt

Abstract. Self-adaptive systems continuously monitor runtime properties and analyze, plan and execute adaptation strategies to adjust their behavior in order to meet satisfactory quality levels. These systems have been applied to a wide variety of scenarios ranging from managing clusters to self-driving cars. However, such achievements require significant computational power, and there is no solid knowledge on which adaptation phases experience the higher consumption of resources. This is unfortunate, as only such insight can support a systematic optimization by improving tools, algorithms or technologies.
In this paper we assess the performance of a reference model architecture for self-adaptive systems, the MAPE-K loop, identify bottlenecks, then suggest and implement some improvements. Finally, we present a performance comparison between different approaches including the one with our improvements and show the actual financial costs of deploying such systems in a mainstream cloud computing service (AWS). We show that our approach leads to a reduction of up to 45 % on annual computational costs, without degradation on the quality of service provided.

1 Introduction

Self-adaptive systems are able to adjust their behavior in response to changes on the environment or the system itself. They operate with minimal human intervention, or even without any intervention, as they are designed to meet desired quality goals, such as performance, availability and security. Traditional self-adaptive systems, beyond the basic functionalities, require a set of complementary independent tasks to monitor, analyze, decide and act in order to respond to changes while at the same time maintaining the desired quality levels. However, a system that is constantly monitoring, planning decisions and evaluating possible adaptation strategies incurs an increased computational cost when compared to non-adaptive solutions. This overhead can have a detrimental affect on baseline performance specially under a high system load. To the best of our knowledge, the research community has not addressed the impact of the self-adaptive mecahnisms on the baseline computational resources, neither is clearly known which adaptation phase consumes more resources.

In this paper we analyze the typical self-adaptation loop [1] based on the MAPE-K model proposed by IBM in 2006 [2] in terms of performance and cost of deploying and maintaining such solutions. In particular, we carry out a comparison between non-adaptive and adaptive systems taking into consideration their performance and the actual achievement of the desired quality goals. Finally, we identify the components that are top consumers of computational resources and implemented several strategies to reduce that overhead.

Our paper contributes to the research community by assessing the computational overhead of a self-adaptive system in both the controller and the incurred overhead in the target system. Moreover, this study allowed to identify the overhead of each phase of the adaptation loop (monitoring, analyzing, planning and execution) and also the amount of resources spent by having a self-managing, healing and optimizing system driving adaptations to meet desired quality goals.

In the evaluation procedure, we adopted Rainbow [3–5] as the self-adaptive solution and applied a case-study of a news infrastructure system with a workload based on a real Internet phenomenon representing a flood of visitors traffic (the *slashdot effect*). In each experiment, we gathered information about performance, the usage of CPU and memory, which allowed us to identify implementation issues and possible improvements.

This paper is organized as follows. Section 2 presents the related work and Section 3 details the method adopted in this study. In Section 4 we introduce the case-study used for evaluating the approach, whose results are presented and discussed in Section 5. Our contributions are highlighted in Section 6 and the limitations of this study are identified in Section 7. Section 8 closes the paper.

2 Related Work

Self-adaptive systems aim to automate processes in order to discard manual activities which are time-consuming, error-prone and costly. To address this IBM proposed the **MAPE-K** loop [2] which became the reference model for the self-adaptive community [1, 6] and encompasses the following phases:

– **M**onitor - Collects runtime data about system properties, such as response time or failed requests;
– **A**nalysis - Reasons about the data received from the Monitoring phase and decides whether an adaptation is required taking into consideration the service quality goals;
– **P**lan - Decides which adaptation strategy or the course of action is more appropriate to achieve the goals;
– **E**xecute - This phase is responsible to apply the actions determined in the Planning phase;
– **K**nowledge - A base of knowledge shared between the different phases of the MAPE-K loop that includes information such as the model of the system, collected runtime properties and results of analysis.

Cheng *et al.* [4] studied the effectiveness of Rainbow self-adaptive system by assessing the maintainability of quality attributes in the face of changes, runtime overheads of adaptation and the engineering effort required to deploy such system. They also estimated the overhead of computational resources needed to run such systems and suggested that this particular self-adaptive solution would consume less than 2 % of the CPU in the 'Delegate' process located in target system. Since these estimations have been performed by the development team of the self-adaptive system under study, we decided to validate their results and extend their work by determining the resource overhead in the controller and also, in each one of the MAPE-K phases.

Oreizy *et al.* [7] proposed an infrastructure with two independent processes encompassing an evolution and an adaptation phases. During his study, authors identified large computational overheads in specific phases of the MAPE-K loop, specially in the Monitoring phase. The study concludes by suggesting the development of new technologies and algorithms to collect metrics at runtime in order to decrease the consumption of computational resources.

Villegas *et al.* [8] proposed a framework to evaluate quality-driven self-adaptive systems by assessing certain properties that are observable from the controller. In this set of observable properties, the small-overshoot measured could be regarded as the adaptation overhead in the context of our work. However, the authors define this small-overshoot as the utilization of computational resources in the execution of an adaptation in order to understand if it performs well under given conditions [8–10], while in our work we assess the computational resources used by the self-adaptive algorithm itself distinguishing between each one of the MAPE-K phases. Thus, we do not assess a particular adaptation, but the whole system identifying the excess of computational resources required to deploy a self-adaptive system and also, to compare with the resources used in a non-adaptive approach.

3 Methodology

Assessing the performance of a self-adaptive system in order to identify issues and distinguish between different phases of the adaptation loop, requires distinct methods and techniques to be applied. In this section, we discuss which of these techniques and methods have been applied to our study and how the performance analysis was implemented.

3.1 Jmeter

We adopted Apache Jmeter as our stress measurement tool. Jmeter sends HTTP requests to a Load Balancer server that redirects the requests among the available web-servers. Then, these servers reply with the appropriate response to the Jmeter which records the response successfulness and the end-to-end time spent in this operation.

Furthermore, Jmeter provides OS Process Samplers [11] that can be used to execute commands in the local machines. We used this service to execute *.sh* bash shell scripts to collect CPU and memory measures on the required servers.

3.2 Scripts

The bash scripts were built to retrieve CPU and memory information from the relevant processes like httpd, mysql and Self-Adaptive application processes. These scripts invoke the Unix *top* command to gather relevant data of all alive running processes, such as:

– PID: the process ID of each process;
– %CPU and %MEM: percentage of CPU and memory usage of each process;
– Command: name of the command that points to the PID of each process.

The first step is to collect the PIDs, through the command name, and save them. It is essential to store the PIDs because there may be more than one process running with the same PID and the same name (Parent and Children), namely Httpd processes. Then, this information is parsed using the PIDs that were stored before, in order to update the sum of %CPU and %MEM. The second step is to append the information to a Comma-Separated Values (CSV) file. This procedure occurs with a granularity of one second during the system runtime.

3.3 NetBeans Profiler

In this study we performed a dynamic analysis in the system to identify which method, class or package was consuming most resources. To this end, we used the Netbeans Profiler [12] which maps the running bytecode instructions to the written code.

When measuring CPU time, two types of profiling must be considered:

– Exclusive time: reports the time spent, in percent, of all methods or classes in runtime;
– Inclusive time: reports the time spent, in percent, of the methods that were called by a specific method in runtime.

When measuring memory usage, we have also two profiling options:

– Memory option: reports the amount of bytes allocated by a specific object that was referenced by a specific method; this a dynamic perspective of the memory usage in runtime.
– Garbage Collector option: calls the Garbage Collector and update the results of memory allocation. This is a like a 'photo', of memory usage at a specific instant in time.

3.4 Analysis of Performance Metrics

The development team of the self-adaptive solution under study (described in Section 4.1) kindly provided us the source code of their implementation. With this information, we were able to analyze the code considering the results of the performance metrics from the scripts and the trace information retrieved from the Netbeans Profiler. Thus, we identified blocks of code that could be targets for improvement. We therefore built a customized version of the self-adaptive solution with the improvements identified, referred as *improved adaptation* from now on.

4 Case-study

In this section we detail the case-study applied in our experimentation procedure. More specifically, we specify the adopted self-adaptive solution, target system and also the intrinsics regarding adaptation, such as goals, metrics and tactics that trigger adaptations. The workload, based on a realistic Internet traffic pattern (embedding a *slashdot effect*), was applied to the variants under test.

4.1 Self-Adaptive System

Rainbow, the platform under test, is an architecture-based self-adaptive system developed at Carnegie Mellon University whose framework is based on the MAPE-K approach and depicted in Figure 1.

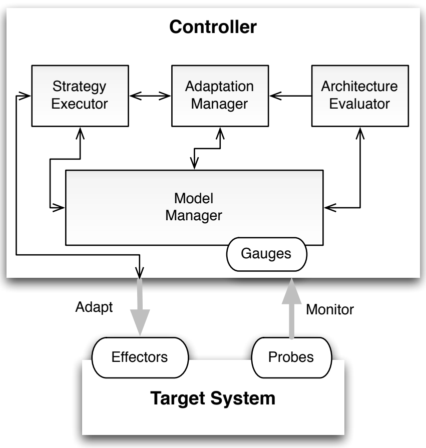


Fig. 1. The *Rainbow* framework

The Controller unit (also known as Oracle) is responsible for managing the adaptation process and is in charge of the following activities:

– Monitor - gathers runtime data from the target system through Probes and Gauges which update properties in the architectural model managed by the Model Manager;
– Analysis - the Architecture Evaluator is responsible for determining from the architectural model whether an adaptation should be triggered to achieve the desired quality levels;
– Plan - the Adaptation Manager selects an adaptation strategy to cope with the deviation from the correct behavior;
– Execute - the Strategy Executor is responsible for applying the sequence of actions defined by the chosen adaptation strategy on the target system through Effectors;
– Knowledge - data is shared between the different adaptation phases. In the Rainbow framework this knowledge-base is handled by the Model Manager which holds an architectural model of the runtime system.

The target system is defined as the resource that is being monitored and adapted to meet the self-adaptation goals.

4.2 Target System

The target system of choice is Znn.com, a typical infrastructure for a news website whose diagram can be depicted in Figure 2. The adaptation control unit

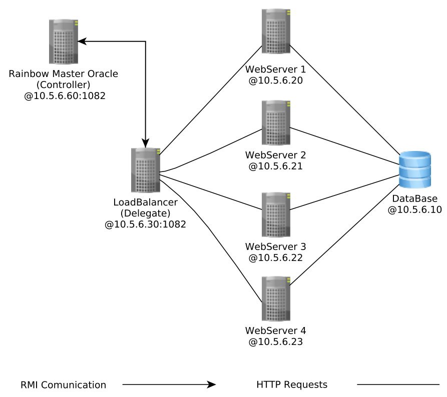


Fig. 2. Znn.com diagram

(the Rainbow *Controller*), was setup in a separate Virtual Machine (VM) to ensure the collection of resource usage values not affected by other processes or tasks. Rainbow requires the setup of a small component (Delegate) to implement Probes and Effectors in the target system. The case-study is composed by a tiered architecture with a set of web-servers that serve content, both textual and graphical, from back-end databases to clients. In addition, it uses a load-balancer to reroute the requests from the clients to a pool of servers. The number of active servers will depend on the selected adaptations to fulfill the system goals.

Each virtual machine of this configuration has the following hardware specifications:

- CPU - Intel Xeon E2620 @2.40GHz, 1 Core;
- Memory - 512 MB;
- Virtual Disk - SATA Controller 16 GB;
- OS - Debian (64 bits).

4.3 Adaptation Goals

As in a typical news provider, ZNN.com focuses on serving news content reliably to its customers while keeping the operating costs at the minimum. In short, the adopted case-study aims to achieve the following quality objectives:

- Availability - The probability that the system is operating properly when it is requested for use, *i.e.*, the probability that the system is not failed or undergoing a repair when it is invoked for use. In the ZNN.com case-study we consider that a failure has occurred when a request is responded with an incorrect HTTP status code, is lost, or takes more than 2 seconds to be responded.

- Operational Cost - Measures the number of Virtual Machines (VMs) required in each scenario under test. We shall use actual cost figures form a commercial cloud provider.

- Usage - The current percentual system load, measured as the amount of work received by the system over the maximum load that is supported by all the available servers.

Rainbow requires the definition of the trade-offs (weights) in multiple quality goal scenarios. Table 1 defines the relative importance between quality dimensions. It can be noticed that availability has been considered twice as important as the current usage of computational resources (cost) or the current load of the system (usage).

4.4 Adaptation Tactics

The Controller may select one of the following tactics to perform adaptation:

- **Enlist server** - From the server pool, if there is a spare server ready to be activated, this strategy enables it. Regarding quality goals, this tactic increases both availability and the used computational resources, since it enables one more server, but decreases the utilization of each one;

Table 1. *Rainbow* Utility preferences

	Weight
Availability	50 %
Cost	25 %
Usage	25 %
Total	100 %

- **Discharge the slowest server** - If there are at least two active servers, our approach will discharge the slowest one (*i.e.*, the one with the highest mean response time). Since we are disabling a server, this tactic decreases cost and may increase system usage;
- **Discharge the least reliable server** - If there is at least two active servers, we will discharge the less reliable. In this case, we also decrease cost and increase system utilization as the previous tactic, but it might increase availability by disabling a failing server.

4.5 Workload

We planned to test our system through a workload that could trigger different adaptations and at the same time to be realistic. Hence, we adopted a workload that is based on an Internet phenomenon, known as *Slashdot effect* or *flash crowd*. This phenomenon is characterized by a low-traffic web-site which suddenly is flooded by visitors for a period of time due to, for example, a breaking news or redirected from a highly-visited website.

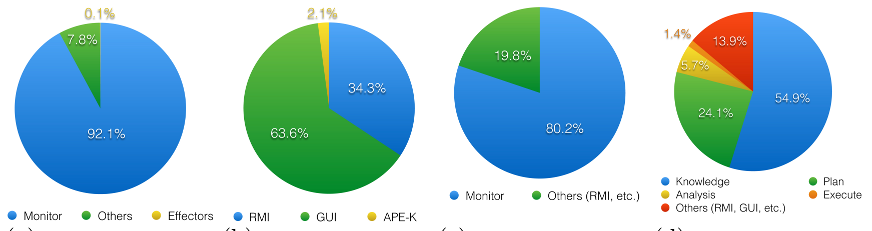
Our workload was designed based on the sample collected from a 2Princeton website [13] and adapted from Cheng *et al.* [4]. The collected sample has been scaled down from twenty-four to one hour, keeping a similarity on visit traffic pattern:

Low Activity 1 minute of low activity requests ($\simeq$ 39 requests/sec);
Sharp Rise 5 minutes of sharp rise in incoming traffic ($\simeq$ 152 requests/sec);
High-Peak 18 minutes of high peak of requests ($\simeq$ 313 requests/sec);
Ramp-Down 36 minutes of linear decrease on requests ($\simeq$ 168 requests/sec)

5 Evaluation

We gathered data about the system performance, using the tools presented in Section 3 to assess the overhead of computational resources in the Self-Adaptive solution under study. Figure 3 depicts the results of that assessment, allowing one to conclude that the Monitor phase consumed a large amount of CPU usage: the Delegate process spent in average near to 58.7 % of which 92.1 % has been consumed by the that particular adaptation phase.

Fig. 3. Graph results for Profiler procedure



(a) CPU usage by Delegate process: 58.7 % (b) CPU usage by Oracle process: 9.9 % (c) Memory usage by Delegate process: 147 MB (d) Memory usage by Oracle process: 320 MB

On the other hand, there was a huge bulk memory consumption in the Oracle process. Of the 512 MB available in the virtual machine, the Oracle process consumed in average 320 MB. In this case, the Knowledge and Plan phases of MAPE-K loop spent in average close to 176 MB (54.9 % of 320 MB) and 77 MB (24.1 % of 320 MB), respectively.

This section details the experiments in terms of quality attributes and computational resources consumption considering three types of adaptive approaches:

Non-adaptive The non-adaptive approach with four permanently active servers ready to serve client requests.

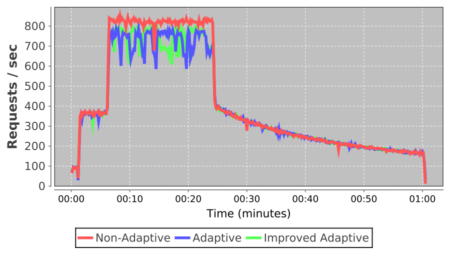
Adaptation Original adaptive approach aiming to achieve the quality goals presented in Section 4.3 through the implemented tactics described in Section 4.4. This is the original adaptive approach based on the baseline implementation which did not address performance concerns.

Improved Adaptation We profiled the original adaptation and identified a large amount of CPU and memory usage by certain adaptation phases, architectural components or blocks of code, as presented in Figure 3. Aiming to identify sources of overhead, we examined the Monitoring and Knowledge phases in detail as these had greater resource consumption. In the Monitoring phase, we found that parsing the Apache logs to collect runtime data could be significantly improved. Furthermore, we eliminated explicit calls to the garbage collector (these aimed to reduce the memory footprint, but increased the CPU usage disproportionally). Regarding the Knowledge phase, we identified and corrected a memory consumption problem in a method that copied the runtime architectural model. The *improved adaptation* implements these changes in the Monitoring and Knowledge phases, and results in a low performance overhead without modifying the self-adaptive algorithm.

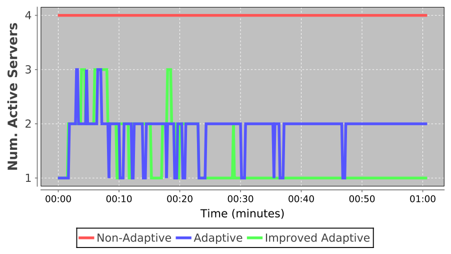
5.1 Analysis of Quality Attributes

Self-adaptive solutions perform changes to achieve and maintain reasonable quality goals. In this research work, we implemented a customized improvement of a traditional self-adaptive solution, the *improved adaptation*. One of our requirements is to present similar quality goals as traditional approaches and so, in this section we discuss the quality outcome of the approaches under study.

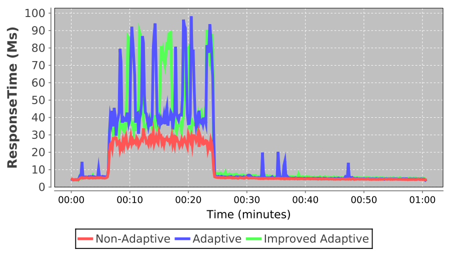
Fig. 4. Graph results for Quality Attributes



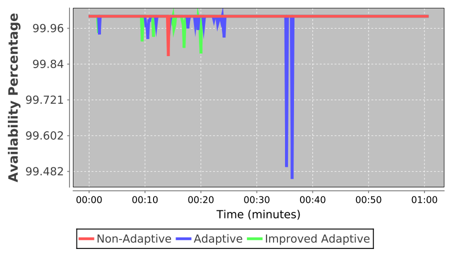
(a) Throughput



(b) Cost



(c) Response Time



(d) Availability

Under those circumstances, Figure 4 illustrates the behavior of the different solutions. Figure 4 (a) (throughput) depicts the number of requests processed and one can notice that the values are similar across the approaches under study. Only during the high-peak requests (between the 7^{th} and 23^{rd} minute) a slightly higher throughput is observed for the Non-adaptive approach. This can be easily understood as the Non-adaptive scenario has all four servers continuously active, active while the other approaches only activate two additional servers to reach similar levels of availability with much lower cost. This is apparent by observing Figure 4 (b) that shows the number of active servers for each scenario.

Figure 4 (c) depicts the response time of each version and it is observable that both adaptive and non-adaptive approaches present similar values during the low activity period, sharp rise (0 to 6 minutes) and the ramp down (24 to 60 minutes). However, during the high-peak of requests both adaptive approaches present high oscillations in response times due to the adaptations being performed which results in mean higher response times and also in sporadic lower availability as illustrated in Figure 4 (d) (remember that availability is measured as the capability to serve requests inside a predefined time frame).

Table 2. Complemented data of Quality Attributes

				Active Resources				
	Requests	Successful Requests	Availability (%)	Load Balancer	Database	Web Servers	Oracle	Total
Non-adaptive	1538524	1538515	99.99	1	1	4	0	6
Adaptation	1439647	1439576	99.99	1	1	1.89	1	4.89
Improved Adaptation	1452550	1452502	99.99	1	1	1.36	1	4.36

Table 2 complements Figure 4 in which we can conclude that almost of all quality attribute values are similar, differing just in the number of processed requests and in the number of resources used. The former varies according to the response time of the request in an inverse relation (*i.e.*, the lower the response time, the higher the number of requests the server can process). The latter varies only in adaptive approaches in which they enlist and discharge servers according to the defined quality goals. As displayed in Figure 2 of Section 4, our configuration requires up to seven virtual machines that can be used by the different approaches. It is mandatory to have a Load Balancer, one Database server and at least one (of four) active web server. In case of adaptive approaches, it is required to have one more machine activated where Oracle process is allocated.

To achieve the goals presented above, we need 6, 4.89 and 4.36 virtual machines by Non-Adaptive, Adaptation and Improved Adaptation, respectively. Thus, we can conclude that *Improved Adaptation* is the most suitable approach, once it is be able to comply with the stated quality goals and operate using less resources (active virtual machines).

5.2 Performance Assessment

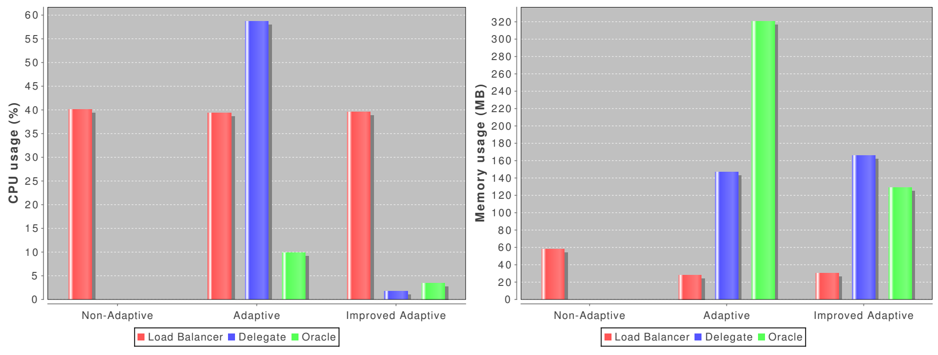
We shall now present the resources consumed by the adaptation services and its improvement. Then, we will perform an analysis to compare all approaches under study.

These improvements were only possible with the insights from the detailing profiling performed, allowing us to fix some implementation issues during the Monitor phase which were having a negative impact on the CPU usage. Regarding memory, we also found a memory leak in the Plan and Knowledge phases of MAPE-K loop. However, this issue was located in a unaccessible block of code. The Rainbow development team was contacted and they promptly sent us a revised version of the library.

Table 3 supports the analysis described above and introduces some additional information, namely the standard deviation. This parameter shows that there is a huge oscillation of requests in the course of the running-time defined by the workload.

The 95th percentile shows the upper value of the parameter during 95 % of the workload. For example, the 95th percentile of CPU in Load Balancer of Non-

Fig. 5. Comparing resources values between Non-adaptive and Adaptation approaches



(a) CPU (b) Memory

Table 3. Comparing resource values between Non-adaptive and Adaptation approaches

		CPU (%)			Memory (MB)		
		Average	Standard Deviation	95th Percentile	Average	Standard Deviation	95th Percentile
Load Balancer	Non-Adaptive	40.12	18.55	83.9	57.9	12.99	62.97
	Adaptation	39.35	21.24	88	28.33	8.20	48.64
	Improved Adaptation	39.57	21.15	89.4	30.25	10.57	52.22
Delegate	Adaptation	58.72	23.08	81.8	147.03	35.03	162
	Improved Adaptation	1.78	2.60	4	166.14	29.96	181
Oracle	Adaptation	9.87	16.28	39.90	320.69	113.09	455
	Improved Adaptation	3.50	5.78	14	128.80	11.23	138

adaptive approach is 83.9 %, means that during 95 % of the workload, the CPU usage was less than 83.9 %.

The values of 95th percentile of the Load Balancer are very similar for all approaches. They are the maximum values as well, once they represent the high peak phase of workload and it is in this phase that the Load Balancer has more work to do. In the Delegate process, the Improved Adaptation scenario has the lowest CPU usage, and a slight increase in memory consumption. Finally, our Improved Adaptation solution presents the best results in memory usage for the Oracle process.

5.3 Cost Impact

From the experimental observations presented above, using the Improved Adaptation system it is possible to achieve the desired quality goals with a significant decrease in computing power. With the overhead eliminated, it is possible to eliminate one virtual machine of the baseline configuration. According to defined hardware specifications, each virtual machine contains 512 MB dedicated of memory, and as virtual machines are replicated, we can merge the Load Balancer with the Delegate and Oracle, and therefore get in average near to 44.8 % and 325 MB consumption of CPU and memory, respectively. Table 4 illustrates the actual cost that each approach would involve per year in the Amazon AWS store (as of June 2014). As a result, per year, the Adaptation solution would represent a 18.1 % cost reduction while the Improved Adaptation would reach a staggering 44.1 % cost reduction which is even more appellative to users of this kind of systems.

Table 4. Cost comparison of Virtual Machines between Non-adaptive and Adaptation approaches

	Number of Virtual Machines	Price per year (US$)
Non-adaptive	6	3698.28
Adaptation	4.9	3027.36
Improved Adaptation	3.36	2066.4

6 Contributions

We list below the major contributions of this paper to the Self-Adaptive Computing research field.

- Validate the effectiveness of Rainbow - The results presented in Section 5 allow one to conclude that Rainbow incurs low overhead in terms of CPU in its optimized version. Thus, we validate the Cheng's statement [4] that the delegate process consumes in average 2 % of CPU.

- Assess the overhead of the controller - In this work we intended to evaluate the resource overhead of the controller due to the lack of research evidence on the subject. Section 5 shows that the overhead of having a unit that manages the adaptation process is low, registering an average of 3.5 % of CPU and close to 129 MB of memory.

- Identify the overhead in each phase of the MAPE-K - This study provides information about which phases of the self-adaptive loop require more focus in future research or development effort in future releases. Prior to performing any improvements to the system, it is necessary to assess it through extensive profiling. From our observations we could conclude that the monitoring phase consumes a large fraction of resources, compared to any other phase. This is where efforts should be put to improve the effectiveness of self-adaptation techniques.

- Comparison between non-adaptive and self-adaptive approaches - In this study we showed that self-adaptive systems have a advantages when compared to non-adaptive approaches, specially in terms of cost. In addition, such systems require low resources and can be applied to wide variety of scenarios.

7 Limitations

- Monitoring can increase CPU overhead. In this case-study we monitored 4 different runtime properties (response time, throughput, successfulness of the request and which server handled the requests). In more demanding scenarios with an increased number of monitored properties, the system may experience a higher overhead of computational resources. Such analysis will be performed in future work.

- Hidden VM cost. Each machine that is activated or deactivated takes less than 3 seconds to begin serving requests. However, in the results presented we only consider that a machine is active if it responds to requests. Thus, the cost of activation and deactivation (in VM·hours), representing less than 3 seconds, is hidden from those results.

- Validity of the experiments. *Znn.com* is widely used in the self-adaptive community as the reference scenario to perform tests and analyzes or to propose new techniques and algorithms on such systems [4, 5, 14, 15]. This case-study includes a LAMP stack (Linux, Apache, MySQL and PHP) which is representative of real world web systems. The Slashdot effect [13] workload represents a real world phenomenon of a traffic flood which exercises all the adaptation tactics present in our case-study. The workload executes for one hour, containing periods of low, moderate and high demand, sharp peaks as well as progressively decreasing load. A limitation of our analysis is that it focuses on only one self-adaptive system. Analyzing other systems would allow us to draw more general conclusions. Nevertheless, an initial examination identified high resource consumption as an issue, and after tracing and instrumenting the system we concluded that it was due to implementation decisions rather than the self-adaptive algorithm itself. After improving the implementation, Rainbow uses in average 1.8 % and 3.5 % of CPU and approximately 166 MB and 129 MB of memory in the Delegate and Oracle, respectively. This result increases our confidence in that the self-adaptive activities in the MAPE-K loop can be efficiently implemented without compromising performance.

8 Conclusions

In this paper we evaluated the performance of a self-adaptive system and identified the computational resources overhead in each of the adaptation phases. In addition, we proposed improvements that guarantee the stated quality goals using less resources.

In this study we evaluated a specific Self-Adaptive system (Rainbow) and we found that in fact there was unnecessary computational overhead. After discovering what phases of MAPE-K loop algorithm consumed more resources, we optimized the Monitor, Plan and Knowledge phases. This resulted in an optimized version of the self-adaptive solution, reducing cost by using less virtual machines while maintaining the intended quality goals.

In a future work we will analyze a more complex target system and evaluate the behavior of Rainbow in terms of its adaptation goals and computational resources. A further step is to extend this assessment to other Self-Adaptive systems.

Acknowledgment

This work was funded through FCT – Fundação para a Ciência e a Tecnologia in the scope of project DECAF: An Exploratory Study of Distributed Cloud Application Failures (reference EXPL/EEI-ESS/2542/2013) and also through the PhD research grant SFRH/BD/89702/2012. In addition, this material is supported by the QREN "Programa Operacional Regional do Centro" under Grant CENTRO-07-ST24-FEDER-002003, project ICIS - Intelligent Computing in the Internet of Services.

References

1. Weyns, D., Malek, S., Andersson, J.: Forms: Unifying reference model for formal specification of distributed self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **7**(1) (May 2012) 8:1–8:61
2. IBM, A.C.: An architectural blueprint for autonomic computing. IBM White Paper (June) (2006)
3. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54
4. Cheng, S.W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the Rainbow self-adaptive system. 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (2009) 132–141

5. Cheng, S.W.: Rainbow: Cost-effective Software Architecture-based Self-adaptation. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2008)
6. Salehie, M., Tahvildari, L.: Self-Adaptive Software: Landscape and Research Challenges. ACM Transactions on Autonomous and Adaptive Systems **4**(2) (2009) 1–42
7. Oreizy, P., Gorlick, M., Taylor, R., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14**(3) (May 1999) 54–62
8. Villegas, N.M., Müller, H.a., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11 **1** (2011) 80
9. Lemos, R., Giese, H., Muller, H.A., Shaw, M., Andersson, J., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., M. Goeschka, K., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Litoiu, M., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezze, M., Prehofe, C., Schäfer, W., Schlichting, R., Schmerl, B., B. Smith, D., P. Sousa, J., Tamura, G., Tahvildari, L., M. Villegas, N., Vogel, T., Weyns, D., Wong, K., Wuttke, J.: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: Software Engineering for Self-Adaptive Systems. Volume 7475 of Dagstuhl Seminar Proceedings. Springer (2013) 1–26
10. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: A research roadmap. In: Software Engineering for Self-Adaptive Systems. Springer-Verlag, Berlin, Heidelberg (2009) 1–26
11. Halili, E.: Apache JMeter. Packt Publishing (2008)
12. Hunt, C., John, B.: Java Performance. 1st edn. Prentice Hall Press, Upper Saddle River, NJ, USA (2011)
13. Slashdotting of mjuric/universe. `http://web.archive.org/web/20090227001212/http://www.astro.princeton.edu/~mjuric/universe/slashdotting/` (January 2004)
14. Camara, J., De Lemos, R.: Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In: Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on. (June 2012) 53–62
15. Casanova, P., Garlan, D., Schmerl, B., Abreu, R.: Diagnosing architectural run-time failures. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '13, Piscataway, NJ, USA, IEEE Press (2013) 103–112