Look Ahead Distributed Planning For Application Management In Cloud

Farzin Zaker
Lassonde School of Engineering
York University
Toronto, Canada
fzaker@yorku.ca

Marin Litoiu
Lassonde School of Engineering
York University
Toronto, Canada
mlitoiu@yorku.ca

Mark Shtern
Lassonde School of Engineering
York University
Toronto, Canada
mark@cse.yorku.ca

***Abstract*—In this paper, we propose and implement a distributed autonomic manager to maintain service level agreements (SLA) for each application' scenario. The proposed autonomic manager seeks to support SLAs by configuring bandwidth ratios for each application scenario using overlay network before provisioning more computing resources. The most important aspect of the proposed autonomic manager is scalability which allows us to deal with geographically distributed cloud-based applications and large volume of computation. This can be useful in look ahead optimization and when using complex models, such as machine learning. Through experiments on Amazon AWS cloud, we demonstrate the elasticity of the autonomic manager.**

***Index Terms*—Distributed Planning, Autonomic Systems, Cloud Resource Management, Machine Learning, Self-Testing**

I. Introduction

Driven by the increasing popularity of information technology in our society, the number of distributed heterogeneous software systems is rapidly growing. A distributed system is composed of multiple cooperating components that communicate through message passing [1] over a network. Versatility, flexibility, scalability, and low-cost management are essential requirements of a distributed system to cope with its increasing complexity. Distributed systems need to manage their behavior by exhibiting self-adaptive properties [2] to achieve the desired run-time qualities. Self-adaptive software systems modify their behavior in response to changes in operating environments [3]. The mechanism for achieving run-time adaptation is embodied through feedback loops in the form of Monitor-Analyze-Plan-Execute (MAPE) architecture [4]. Feedback loops can be implemented in either a centralized or decentralized manner depending on the design features and requirements.

Nowadays, more and more applications are deployed on clouds that facilitate elastic resources to plan and execute management changes dynamically [5]. It has been shown that in cloud computing deployments, the performance goals of applications, such as maintaining service level agreements (SLA), can be achieved through many different run-time changes [6]. Most approaches only consider provisioning/deprovisioning computing resources, and few try to maintain SLAs using other parameters such as network bandwidth configuration to reduce the costs. In this paper, we use both resource provisioning and dynamic bandwidth configuration to maintain the performance of managed applications. Previously proposed mechanisms using bandwidth reconfiguration use a central autonomic manager which measures and monitors system parameters and applies corrective actions accordingly. However, a centralized autonomic manager can become a bottleneck itself, especially when it needs to run complex planning mechanisms over a look ahead window.

In this paper, we address the following research question:
RQ. How can we design and implement a scalable look ahead planning mechanism to maintain performance metrics of cloud applications?
We take advantages of the *Actor Model* [7] and *machine learning models* to design our distributed planning mechanism. The proposed solution evaluates a domain-specific set of adaptation options at run-time and examines possible consequences of each adaptation option in the future, aka a look ahead window. We use machine learning (ML) to model and predict both future workloads and application performance. At high level, our distributed autonomic manager implements the MAPE-K control loop composed of Monitor, Analyze, Plan and Execute functions supported by a Knowledge Base.

The remainder of this paper is organized as follows. We explore related works in section II. In Section III, we introduce the purposed method and architecture, including machine learning models, and adaptation algorithms. Section IV is devoted to explaining experiment setups and analyzing results. Finally, section VI concludes achievements and states future works.

II. Related Work

Various approaches have been proposed so far to deal with the automatic cloud resource management problem [8], [9]. The work is motivated by the need to maintain SLAs in response to continuously changing workloads. Some of the conventional approaches in auto-scaling mechanisms are threshold-based rules, reinforcement learning, queuing theory, control theory, and time-series analysis [10].

Besides approaches that only focus on provisioning new computing resources in response to changes in workloads, some studies pay more attention to cost-less options such as network bandwidth allocations. For instance, in [11], a hill-climbing heuristic at run-time has been used to dynamically adapt bandwidth of application flows to postpone provisioning virtual machines for as long as possible. The overall performance of the autonomic manager has been improved in [12] by using machine learning models. Authors in both [11] and [12] introduce a *centralized autonomic manager* similar to most other auto-scaling approaches. Considering previous works, we design a novel distributed autonomic manager taking both resource provisioning and bandwidth configuration options into account.

Actor Model [7] is a conceptual model to deal with concurrent computations in distributed systems. It minimizes tight-dependencies between system components by assigning system tasks to actors that communicate with each other through message passing. More resources allocated to a well-designed scalable distributed autonomic manager leads to more reliable decisions based on precise foresight. Stated characteristics of the *Actor Model* makes it a good fit for the requirements of our distributed autonomic manager.

One of the key challenges in designing and implementation of adaptive systems is providing assurances that the principal goals of self-adaptive systems are maintained when the autonomic system changes its behavior [13]. *Self-testing* as a common approach to providing assurances for self-adaptive systems is the ability of a self-adaptive system to test its behavior during and after applying an adaptation [14]. Some previous works such as [15], [16], and [17] are dedicated to automatically running Self-Testing mechanisms on change requests at run-time. Although many pieces of research targeted different Self-testing requirements, and various tools have been developed so far, our searches failed to find an appropriate toolset or approach for applying Self-testing in a distributed system. In our proposed approach, the distributed autonomic manager implements the replication with validation method for performance evaluation of available adaptation options at run-time.

Automated testing mechanisms necessitate *Test Data* and *Test Oracle* to validate test results. Workload and performance models can be adapted to generate Test Data and create proper Test Oracle at run-time. Machine learning models have been widely employed in modeling the performance of different systems. For instance, Bodik et al. [18] applied curve-fitting and local regression machine learning approaches to model the performance model of an Internet data center based on current workload and system configuration. Li et al. in [19], Maggio et al. in [20], and Gambi et al. in [21] proposed other applications of machine learning models in designing and implementation of elastic cloud environments. Previously proposed methods take advantage of centralized or distributed machine learning algorithms to support decisions made by a centralized autonomic manager. On the contrary, our proposed autonomic manager applies machine learning models to make decisions in a distributed manner.

III. Method And Architecture

We target web applications hosted in cloud serving different categories of incoming requests, also known as *Scenarios*. Each scenario has an upper bound response time defined by SLAs. The autonomic manager observes the application performance and plans necessary modifications in the application resources or configuration parameters to maintain SLAs. To introduce the distributed planning mechanism, we start with its conceptual architecture. Afterward, we use an illustrative example to demonstrate the original idea. Since we use predictive machine learning models, a subsection is devoted to explaining how designated ML models are structured. Later, we take a more in-depth look into the algorithms used for implementing the distributed planning mechanism. The complexity level of the proposed distributed algorithm demands formal verification to prove that the algorithm is deadlock-free, adaptations are mutually exclusive, and all planning routines eventually come up with a decision.

A. Method Overview

Figure 1 illustrates how the proposed distributed look ahead planning works. The autonomic system is composed of multiple *Autonomic Actor*s illustrated in Fig. 1 with blue color. Each autonomic actor monitors response times of a specific scenario and once its SLA is violated initiates the adaptation routine (red actors). Afterward, the autonomic actors will examine available *adaptation options* over a look ahead window by creating shadow copies of themselves also called as *Test actors*. Each adaptation option will be assigned to a separate test actor (orange actors). This Self-Testing method is called replication with validation. Essentially, each *Test Actor* tests benefits of a specific adaption option in response to the workload predicted at that point in time (on the horizontal axis, the orange circles are shown at discrete times). Machine learning models are employed to generate workloads (Test Data) and predict application response time (Test Oracle). An *adaptation option* or *control points vector* is a set of network bandwidth allocation per scenario and the number of computing resources assigned to the application. At the very end, an optimal adaptation option is found and applied (green actors). In other words, the autonomic manager creates multiple instances of the model for all adaptation options and solves those models.

The number of simulation steps is specified by the look ahead window. During the simulation run, each *Test Actor* gathers required audit parameters - such as the total number of SLA violations or adaptations - for evaluating adaptation options. When the simulation is over, the autonomic manager applies a scoring function to estimate the benefits of each adaptation option. Equation 1 displays generic composition of a scoring function with c_i as effective weight of each audit parameter P_i. For example, we can utilize the weight of each parameter to adjust the relative impact of SLA violations

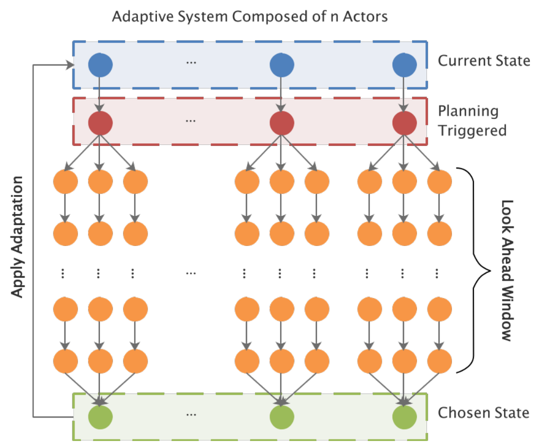


Fig. 1. Look Ahead Distributed Planning

and adaptations in the final decision made by the autonomic manager.

$$\mathcal{S} = Avg(\sum_{i=1}^{n} c_i \times P_i) \tag{1}$$

The proposed distributed planning mechanism needs a limited set of predefined adaptation options. Without limiting the number of available adaptation options, the state space explosion problem threatens performance of the autonomic manager. In practical terms, the type of adaptations available (bandwidth, virtual machines or container scaling, threads or other parameters tuning, etc.) is limited by technological constraints and therefore the state explosion is not a concern. In addition, experts' knowledge of application, cloud, and network management will further limit the adaptation options.

B. Illustrative Example

In order to make the distributed planning more clear, consider an e-commerce web application with three different scenarios and their corresponding response time SLAs:

- Browse: Exploring different categories and products (SLA: 1000 ms)
- Basket: Shopping basket manipulation (SLA: 400 ms)
- Admin: Data entry and order management (SLA: 2000 ms)

SLA definitions come from business needs based on the priority of each scenario. In this case, three *Autonomic Actors* continuously monitor the workload and response time of specified scenarios. Suppose that the response time of Basket scenario violates 400 ms SLA while other indicators for Browse and Admin scenarios are below 500 ms and 1000 ms respectively. As a result, the second *Autonomic Actor* (initiator) triggers distributed planning routine and creates *Test Actor*s for available adaptation options as shown in Figure 2(a). In this case, available adaptation options are: (1) decreasing the relative bandwidth allocation of Browse; or (2) of Admin scenarios to delay processing those requests and free resources for the Basket scenario; and (3) provisioning new server nodes.

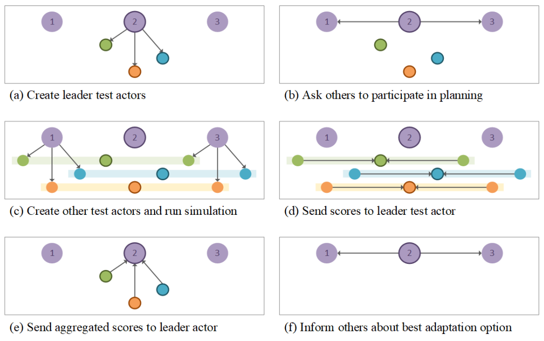

(a) Create leader test actors
(b) Ask others to participate in planning
(c) Create other test actors and run simulation
(d) Send scores to leader test actor
(e) Send aggregated scores to leader actor
(f) Inform others about best adaptation option

Fig. 2. Distributed Planning

The next step is shown in Figure 2(b), where the initiator invites other *Autonomic Actors* to participate in the distributed planning routine. Other *Autonomic Actors*, similar to the initiator actor, create *Test Actor*s for simulating all available adaptation options. Afterward, *Test Actor*s handling the same adaptation option shape networks, similar to what is presented in Figure 2(c) with different colors. *Test Actor*s utilize two different machine learning models to predict future workloads (*Test Data*) and foretell response times (*Test Oracle*).

When the simulation is done, *Test Actors* use a predefined scoring function (cf. equation (1)) to evaluate each adaptation option. As presented in Figure 2(d), each *Test Actor* terminates itself after sending the calculated score to the initiator *Test Actor* in the same network. Figure 2(e) illustrates how the initiator *Test Actors* send aggregated scores to their parents before terminating themselves. The final step is shown in Figure 2(f) when the initiator *Autonomic Actor* picks the best adaptation option and notifies other actors about the decision made.

C. Machine Learning Models

While networks of *Test Actor*s are testing the performance of different adaptation options, each *Test Actor* utilizes two machine learning models to perform as *Test Data* generator and *Test Oracle*. Although those machine learning models have to be trained offline initially, *Autonomic Actor*s can continuously improve them serving them new data collected from the environment. Each *Autonomic Actor* shares its prepared machine learning models with newly created *Test Actors*.

1) Workload Model: The *Workload Model* supports each *Test Actor* in predicting the future workload of its assigned scenario as *Test Data*. As shown in Equation 2, the workload model ($\mathcal{W}$) receives discrete time as input and outputs likely requests traffic for a specific scenario.

$$\mathcal{W}(time) = \overrightarrow{WL} \tag{2}$$

2) Performance Model: *Test Actors* employ the *Performance Model* to estimate performance of the monitored system in response to varying workloads. According to Equation 3, the *Performance Model* ($\mathcal{P}$) is a function of control point values (CPI), such as the overlay network bandwidth allocation and amount of provisioned resources, current performance indicators (PIV_c) such as average response time, and current workload (WL). Outputs of the *Performance Model* can be used as *Test Oracle* to evaluate effectiveness of a specific adaptation option on maintaining system SLAs.

$$\boldsymbol{\mathcal{P}}(\overrightarrow{CPV}, \overrightarrow{WL}, \overrightarrow{PIV_c}) = \overrightarrow{PIV} \tag{3}$$

Having future workload estimated by Equations 2, the autonomic manager can apply search algorithms over the model described by Equation 3 and determine the proper control point values and calculate the resulting response time.

Both *Workload Model* and *Performance Model* are supposed to predict numeric data, which makes regression algorithms good candidates for the implementation. Since workload characteristics and deployment topology may change periodically, machine learning models need to employ classification algorithms to choose different regression formulas under different situations. In addition, both models support online learning to let the autonomic manager improve predictions precision by feeding monitored metrics at runtime.

D. Distributed Planning Algorithm

When an *Autonomic Actor* (initiator) detects the need for adaptation, it executes the sequence of actions presented in Algorithm 1. *Test Actor*s follow the instructions given in Algorithm 2. Lines 1 to 7 of Algorithm 1 initiate the adaptation process, create *Test Actor*s and invite other *Adaptive Actor*s to participate. When an Adaptive Actor receives such a message, creates required *Test Actor*s using the message server implemented in lines 8 to 12 of Algorithm 1. *Test Actor*s utilize the *Workload Model* to generate *Test Data*, validate results using the *Performance Model*, calculate score of the assigned adaptation option, and finally send calculated score to the leader *Test Actor* in lines 1 to 8 of Algorithm 2. Lines 9 to 17 of Algorithm 2 illustrate how the leader *Test Actor* calculates and sends aggregated score to the parent *Adaptive Actor*. Finally, the initiator *Adaptive Actor* finds the best adaptation option based on the calculated scores and asks other *Adaptive Actor*s to apply the selected adaption option as shown in lines 13 to 29 of Algorithm 1.

Algorithm 1 Planning In Autonomic Actors

```
1: $actorsCount \leftarrow neighbors$.length + 1
2: for all $option \in adaptationOptions$ do
3:     $testActor \leftarrow$
           new $TestActor(models,\ option,\ testActor,\ actorsCount)$
4:     for all $actor \in neighbors$ do
5:         send $< option, testActor >$ to $actor$
6:     end for
7: end for

8: $initiator \leftarrow$ null;
9: on message $< option, testLeader >$ do
10:    $testActor \leftarrow$
           new $TestActor(models,\ option,\ testLeader,\ actorsCount)$
11:    $initiator \leftarrow sender$
12: end message

13: $scores \leftarrow Map < Option, Score >$
14: on message $< option, score >$ do
15:    $scores[option] \leftarrow score$
16:    if $scores$.length = $adaptationOptions$.length then
17:        $bestScore \leftarrow \infty$
18:        $bestOption \leftarrow$ null
19:        for all $option \in adaptationOptions$ do
20:            if ( then$scores[option]$ >$bestScore$)
21:                $bestScore \leftarrow scores[option]$
22:                $bestOption \leftarrow option$
23:            end if
24:        end for
25:        for all $actor \in (neighbors + self)$ do
26:            send $bestOption$ to $actor$
27:        end for
28:    end if
29: end message
```

Algorithm 2 Planning In Test Actors

```
1: for $i \leftarrow 0$ to $simulatedIterations$ do
2:     Simulate system execution using $models$
3: end for
4: $score \leftarrow scoringFunction()$
5: send $score$ to $leader$
6: if $self$ <>$leader$ then
7:     terminate
8: end if

9: $scores \leftarrow List < Score >$
10: on message $score$ do
11:    $scores$.add($score$)
12:    if $scores$.length = $actorsCount$ then
13:        $averageScore \leftarrow average(scores)$
14:        send $< option, averageScore >$ to $parent$
15:        terminate
16:    end if
17: end message
```

IV. Experimental Validation

In this section, we examine the feasibility and effectiveness of the proposed approach through experiments on AWS EC2. We chose an E-Commerce application with about 1 million daily page views [22]. Our experiment examines the feasibility of the proposed distributed autonomic manager (**RQ**).

A brief analysis of the load balancer log files clarifies that more than 93% of incoming requests belong to one of the four major scenarios listed in Table I.

The application infrastructure is composed of various nodes hosted on AWS as shown in Figure 3[1]. A virtual switch manages the overlay network connecting all server nodes. The *Proxy Server* routes incoming requests to the *Load Balancer* via an overlay network. In Addition, the *Proxy Server* runs the autonomic manager to adjust bandwidth allocations and control number of web servers in the *Application Tier*. We employ *Akka* framework [23] to implement the *Proxy Server* on top of the *Actor Model*. Nginx plays the load balancer role and is configured to dispatch requests between available web servers on a round robin basis. All web servers connect to the same database server. MySQL server stores the whole website data, whereas MongoDB is partially in sync with MySQL server data and provides fast responses to product filtering queries. Technical specification of each node is provided in Table II.

We implement required machine learning models using the *IBK2* algorithm available in *Weka Library* [24]. *IBK2* is a non-parametric online learning method for classification and regression designed based on the k-nearest neighbors algorithm

[1]Experiments setup guides with references to the source codes and Amazon AMI images are available at:
https://github.com/FarzinZaker/LADP-Experiments-Setup

Scenario	Processing Type	Response Time SLA
Browse	Large database queries	1.2 secs
Product	Single database record fetch	0.8 secs
Basket	Database updates	0.5 secs
Static	Read and render files	1.0 secs

TABLE I
CHARACTERISTICS OF EACH SCENARIO

Node	Hardware	Software	Count
Proxy Server	t2.medium	Tomcat + Akka	≥ 1
Load Balancer	t2.medium	Nginx	1
Web Server(s)	m4.large	Tomcat	≥ 1
Database	m4.large	MySQL + MongoDB	1
Virtual Switch	t2.medium	Open vSwitch	1

TABLE II
E-COMMERCE WEBSITE DEPLOYMENT NODES

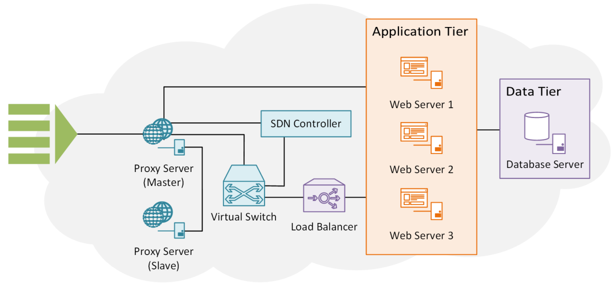


Fig. 3. Deployment View of The E-Commerce Website

(k-NN) and fits the requirements explained in section III-C. Both Workload and Response Time models have been trained using access logs of three randomly chosen dates. During the training phase, a total number of 2,871,544 requests were served, and their arrival rates and response times were fed into the prediction models. Our initial experiments show that the *IBK2* algorithm predicts arrival rate and response time with approximately 96% precision.

During the following experiments, *Autonomic Actor*s initiate the distributed planning routine in case of two subsequent SLA violations. The distributed planning mechanism investigates the possibility of maintaining SLAs by modifying the bandwidth rate of each scenario. The assumption is that the application has a total bandwidth that can be dynamically re-allocated among scenarios in fractions of: [2.5, 0.5, 0.75, 1.0] before provisioning new resources. The scoring function designated for evaluating the performance of adaptation options is specified in equation 4. N_v is the number of SLA violations and N_a is the number of required adaptations during the look ahead window. c_v and c_a are constant weighting factors. The autonomic manager chooses the adaptation option with the lowest $\boldsymbol{\mathcal{S}}$.

$$\boldsymbol{\mathcal{S}} = Avg(c_v \times N_v + c_a \times N_a), c_v = 10, c_a = 1 \qquad (4)$$

According to the log files, the understudy website is experiencing most of its traffic between 8:00 am and 8:00 pm in the local timezone. We use available access logs for the same period on random dates in the experiments.

A. Qualitative Analysis

During this experiment run, the total number of 543,105 requests were fed into the system. The autonomic manager is configured to simulate the next 100 iterations (look ahead window) of serving incoming requests. Figure 4 illustrates the distribution of incoming requests. Horizontal axis labels in figure 4 show the number of iterations passed from the beginning of the experiment. Red dots show the points of time that the autonomic manager starts to determine the best adaptation option.

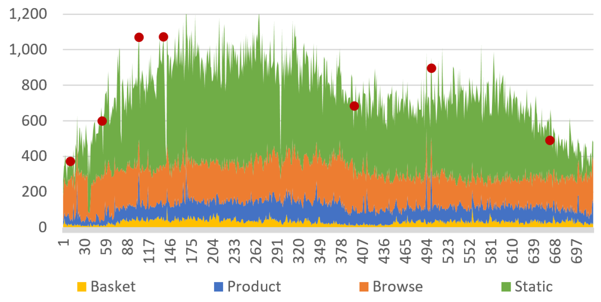
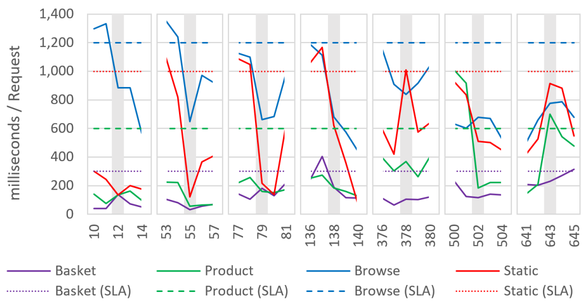


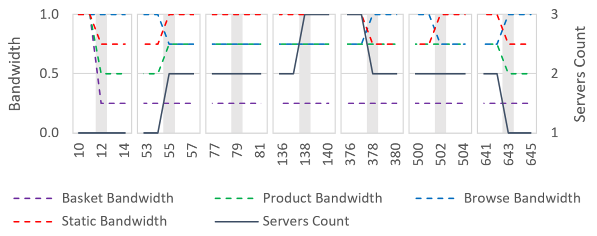
Fig. 4. Distribution of incoming requests in testing phase

In order to interpret experiment results, we present average response times, response time SLAs, and configuration changes in Figure 5. Configuration changes may combine any modification in bandwidth limits of each scenario (dashed lines) and any change in the number of web servers of the Application Tier (solid line). Figure 6 shows average CPU utilization of web servers during each iteration. Adaptation points are highlighted in both Figure 5 and Figure 6.

As shown in Figure 5 and Figure 6, total number of 7 adaption actions take place during 720 iterations. The first adaptation occurs at 12^{th} iteration when two subsequent SLA violations occur in serving the Browse scenario. The bandwidth allocations changes to increase the relative bandwidth of the Browse scenario compared to other scenarios. As a result, the average response time of the Browse scenario drops at 14^{th} iteration. At 55^{th} iteration, another adaptation starts in response to another couple of SLA violations in the Browse scenario. In order to maintain SLAs, the autonomic manager decides to add a web server to the Application tier beside modifying the bandwidth configuration, which decreases the average CPU utilization of web servers in next iterations. Although the autonomic manager detects two subsequent SLA violations at 79^{th} iteration in the Static scenario, decides to keep the system configuration unchanged. The next adaptation action takes place at 138^{th} iteration in response to another couple of SLA violations in the Static scenario. This time, re-configuring bandwidth allocation does not help. Instead, adding a new server to the Application tier causes a drop in average CPU utilization of web servers. The autonomic manager finds out that even by shutting down one of the servers at 378^{th}, it is possible to maintain SLAs. Since SLA of



(a) Response Times



(b) Configuration Changes

Fig. 5. Configuration Changes and Effects on Response Times

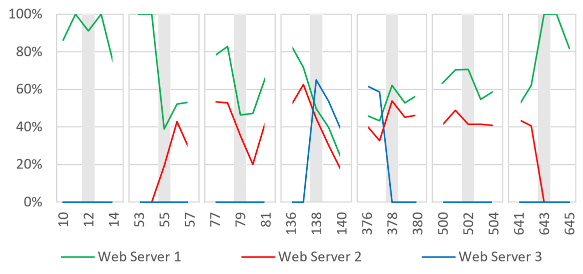


Fig. 6. CPU Utilization of Web Servers

the Static scenario is violated two times before 502^{th} iteration, another adaptation unfolds changes the system configuration. Finally, the last adaptation occurs at 643^{th} iteration, without any SLA violation to shut down another not needed web server.

This experiment confirms the feasibility and applicability of the proposed distributed autonomic manager (**RQ**). It applied seven adaptations in total to maintain SLAs in response to changes in workloads while serving 543,105 incoming requests.

B. *Scalability*

Since the autonomic manager is hosted on proxy servers, we monitor the number and utilization of proxy servers during the first experiment. When the autonomic manager starts the distributed planning phase, it may scale up automatically the number of proxy server nodes by adding new slave nodes according to the current workload and the look ahead window size. When there is no need for a slave node in the next

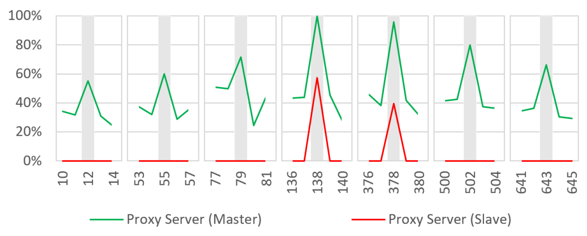


Fig. 7. CPU Utilization of Proxy Servers

adaptation planning, the autonomic manager automatically scales down the proxy server nodes.

As recorded in Figure 7, the autonomic manager scales up itself by adding a new slave node at iterations 138 and 378. During other adaptations, it determines that there is no need for additional resources. Since the incoming traffic can rise to any number of concurrent users, auto-scaling capability enables the proxy server to deal with the even massive incoming traffic (**RQ**).

V. Threats to Validity

We provide the autonomic manager with a predefined set of adaptation options. Moreover, the autonomic manager needs a well-defined scoring function to rank available options at run-time. However, applying a different scoring function or set of adaptation options may cause different outputs.

Although *Text Actor*s are lightweight pieces of code, running multiple instances of Test Actors needs computing resources that may not be available in all distributed systems. In such cases, we need additional servers to run simulations and send planning decisions back to *Autonomic Actor*s. This will impose negative effects on the performance and effectiveness of the algorithms.

VI. Conclusion And Future Work

In this paper, we proposed a look ahead distributed mechanism for autonomic cloud resource planning. Scalability is the essential characteristic of the proposed approach, which empowers the autonomic manager to deal with a large volume of computations. The proposed approach is designed based on the discrete *Actor Model* in order to minimize the coupling level of implemented components. Available adaptation options are being evaluated through replication with validation *Self-Testing*. Machine learning models are responsible to generate *Test Data* and the *Test Oracle* to validate test results. We examined the feasibility, and scalability of the proposed planning mechanism.

Although the proposed approach in this paper is on resource planning for cloud applications, we believe that it is feasible to apply a similar approach to a broader range of problems. For instance, we are going to use lessons learned in this research for an end to end delay management in another solution using the Fog-based IoT architecture.

References

[1] M. Sirjani and M. M. Jaghoori, "Ten years of analyzing actors: Rebeca experience," in *Formal modeling*. Springer-Verlag, 2011, pp. 20–56.

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

[3] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems and Their Applications*, vol. 14, no. 3, pp. 54–62, 1999.

[4] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. M. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops." *Software engineering for self-adaptive systems*, vol. 5525, pp. 48–70, 2009.

[5] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: state of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2018.

[6] P. Zoghi, M. Shtern, M. Litoiu, and H. Ghanbari, "Designing adaptive applications deployed on cloud environments," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 10, no. 4, p. 25, 2016.

[7] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, Tech. Rep., 1985.

[8] R. Weingärtner, G. B. Bräscher, and C. B. Westphall, "Cloud resource management: A survey on forecasting and profiling models," *Journal of Network and Computer Applications*, vol. 47, pp. 99–106, 2015.

[9] S. Singh and I. Chana, "A survey on resource scheduling in cloud computing: Issues and challenges," *Journal of grid computing*, vol. 14, no. 2, pp. 217–264, 2016.

[10] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 73, 2018.

[11] N. Beigi-Mohammadi, H. Khazaei, M. Shtern, C. Barna, and M. Litoiu, "Adaptive service management for cloud applications using overlay networks," in *15th IEEE International Symposium on Integrated Network Management (IM)*, 2017.

[12] N. Beigi-Mohammadi, M. Shtern, and M. Litoiu, "A model-based application autonomic manager with fine granular bandwidth control," in *IEEE 13th International Conference on Network and Service Management (CNSM 2017)*, Nov. 2017.

[13] D. Weyns, N. Bencomo, R. Calinescu, J. Camara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jezequel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli, *Perpetual Assurances in Self-Adaptive Systems*, ser. Lecture Notes in Computer Science. Springer, 2018, vol. 9640, pp. 31–63.

[14] T. M. King, D. Babich, J. Alava, P. J. Clarke, and R. Stevens, "Towards self-testing in autonomic computing systems," in *Autonomous Decentralized Systems, 2007. ISADS'07. Eighth International Symposium on*. IEEE, 2007, pp. 51–58.

[15] Y. M. Roopa and M. R. Babu, "Self-test framework for self-adaptive software architecture," in *Electronics, Communication and Aerospace Technology (ICECA), 2017 International conference of*, vol. 2. IEEE, 2017, pp. 669–674.

[16] A. D. da Costa, C. Nunes, V. T. da Silva, B. Fonseca, and C. J. de Lucena, "Jaaf+ t: a framework to implement self-adaptive agents that apply self-test," in *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 928–935.

[17] T. M. King and A. S. Ganti, "Migrating autonomic self-testing to the cloud," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. IEEE, 2010, pp. 438–443.

[18] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *Proc HotCloud*, 2009.

[19] H. Li and S. Venugopal, "Using reinforcement learning for controlling an elastic web application hosting platform," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 205–208.

[20] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva, "Comparison of decision-making strategies for self-optimization in autonomic computing systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 4, p. 36, 2012.

[21] A. Gambi, G. Toffetti, and M. Pezze, "Assurance of self-adaptive controllers for the cloud," in *Assurances for Self-Adaptive Systems*. Springer, 2013, pp. 311–339.

[22] F. Zaker. (2019) Online shopping store - web server logs. [Online]. Available: https://doi.org/10.7910/DVN/3QBYB5

[23] M. Gupta, *Akka essentials*. Packt Publishing Ltd, 2012.

[24] R. Dimov, M. Feld, D. M. Kipp, D. A. Ndiaye, and D. D. Heckmann, "Weka: Practical machine learning tools and techniques with java implementations," *AI Tools SeminarUniversity of Saarland, WS*, vol. 6, no. 07, 2007.