A Model-based Application Autonomic Manager with Fine Granular Bandwidth Control

Nasim Beigi-Mohammadi, Mark Shtern, and Marin Litoiu
Department of Computer Science, York University, Canada
Email: {nbm, mark, marin}@yorku.ca

In this paper, we propose and implement a machine learning based application autonomic management system that controls the bandwidth rates allocated to each scenario of a web application to postpone scaling out for as long as possible. Through experiments on Amazon AWS cloud, we demonstrate that the autonomic manager is able to quickly meet Service level Agreement (SLA) and reduce the SLA violations by 56% compared to a previous heuristic-based approach.

***Index Terms*—Auto scaling, machine learning, SDN, SLA, bandwidth management.**

I. Introduction

In previous work [1], we proposed to dynamically adapt bandwidth of application flows to postpone scaling out the application for as long as possible. Using experiments on real cloud, we showed how our approach successfully helps applications to maintain response time SLAs of all scenarios without provisioning extra resources for a long time. However, in [1], we used a hill-climbing heuristic at run-time to find the appropriate bandwidth rates for different scenarios. Due to the exploratory nature of the hill-climbing algorithm, it would take time to find the bandwidth rates that satisfy the SLAs. Since all happen at run-time, we need to speed up the run-time reactions and hence overcome the limitation of [1]. Therefore, in this paper, we develop a machine learning model that the application autonomic manger uses to quickly predict the appropriate bandwidth rates of all application scenarios at once. We show that the use of model accelerates run-time reactions in meeting service level objectives. Thus our contributions in this paper are summarized as below:

- we propose a model-driven application autonomic manager for managing SLAs of cloud applications through smart and fine granular bandwidth management. We show that the model accurately predicts the appropriate bandwidth for the application scenarios given the application metrics;
- we implement and verify our solution through experiments on AWS. We compare our model-driven approach with previous work which is a heuristic-based bandwidth management and demonstrate that the new approach reduces SLA violations by 56%.

The rest of the paper is presented as follows: in Section II, we describe the methodology. Section III presents the experiments and results. In section IV, we overview related work. And finally, Section V concludes the paper.

II. Methodology

In our approach, application autonomic manager defines policies to be executed at the network layer. When autonomic manager detects the application is overloaded, it dynamically changes the bandwidth to resolve the performance bottleneck on the application. We make following assumptions:(a) there is dependency between scenarios due to using shared resources;(b) we consider a three tier application where the application tier is scaled in/out on demand and use nodes with high capacity in other tiers to prevent saturation in other tiers;(c) we use homogeneous nodes in the application tier and the requests in the application tier are equally distributed between workers.

Algorithm 1 illustrates the adaptation process of autonomic manager. Autonomic manager uses two criteria to adapt the bandwidth rates which are: (a) the response time of at least one of the scenarios should be below certain threshold with respect to its response time SLA, which we call candidacy threshold (CT); (b) there exists at least one scenario whose response time SLA has not been breached for the past *x* time units. We call this Time From Last Violation (TFLV). If there exists at least one scenario that meet such criteria, autonomic manger performs bandwidth adaptation (line 3 in Algorithm 1), otherwise, it will scale out the application. To prevent oscillation, autonomic manager uses a heat mechanism before acting on a violation or when scaling in the application (lines 2 and 15 respectively).

To capture the non-linear relationship between the response times, workload, and the bandwidth rates, we use a machine learning model formally shown in Equation 1. The inputs of the model are the workload vector (W) which includes the number of users per scenario and the vector of scenarios' response times (R). The output of the model is a vector of bandwidth rates of scenarios (BW); that is given W, if BW is applied to the scenarios, the response times of scenarios will be R. We define the effective workload (W_{eff}) according to Equation 2 where n is the number of nodes (i.e., virtual machines (VMs)) in the application tier. Therefore, in order to use the model for when we have only one node in the application tier, we use Equation 3. Now $BW^{'}$ is applicable to W_{eff} workload; in order to find BW for W, we use a linear model presented in Equation 4. We adopt this formula from our assumptions where we only scale out the application tier with homogeneous VMs and our load balancer uses round-

Algorithm 1: Adaptation Algorithm:

input : $\mathcal{S}$—vector of Scenarios.
input : $\mathcal{R}$—vector of response times of scenarios.
input : SLA—vector of response time SLAs of scenarios.
input : $\mathcal{W}$—vector of workload.
input : n—number of application servers.
input : CPU—average CPU utilization of application servers.
input : $heat_o$, $heat_u$—number of consecutive SLA breaches (overload) and underloaded situations respectively.
output : BW—vector of bandwidth rates that satisfies $\mathcal{R}$.
output : S'— a subset of S that meet the bandwidth adaptation criteria .

```
foreach Scenario s ∈ S do
    if R_s > SLA_s for heat_o(s) times then
        S' ← checkBWCriteria({S − {s}});
        if S' ≠ ∅ then
            W ← W/n;
            BW ← F(W, R) × n;
            Apply BW to S;
            heat_o(s) ← 0;
            return;
        else
            // bandwidth adaptation cannot fix
               violations
            Scale out application tier;
            n ← n + 1;
            heat_o(s) ← 0;
            return;
if CPU < CPU^lo for heat_u times then
    // application tier underloaded
    Remove VM;
    heat_u ← 0;
    n ← n − 1;
    return;
```

robin policy to equally distribute the workload between the application servers. Therefore, using our method, we only need to collect data for one application setting (i.e., having specific number of nodes in the application tier) and use equations 1-4 to use the model for varying number of nodes in the application tier. As part of application performance testing that is done before operation, we tried various values of workload and bandwidth rates. To prevent combinatorial explosion problem, we used the knowledge from application behavior. We use a multiple output decision tree regression as our model.

$$\overrightarrow{BW} = F(\overrightarrow{W}, \overrightarrow{R}) \tag{1}$$

$$\overrightarrow{W}_{eff} = \overrightarrow{W}/n \tag{2}$$

$$\overrightarrow{BW}' = F(\overrightarrow{W}_{eff}, \overrightarrow{R}) \tag{3}$$

$$\overrightarrow{BW} = \overrightarrow{BW}' * n \tag{4}$$

III. Experiment

To implement the autonomic manager, we extended Hogna [2] such that it dynamically establishes an overlay network and perform bandwidth adaptation as well. We equip our autonomic manager with a multiple output decision tree regression mode that we developed in Python3 using scikit-learn [3]. The hyper-parameters set for the model include criterion=mse, splitter=best, max_depth=5, and random_state=0, which attain score function of 0.974. We performed two sets of experiments

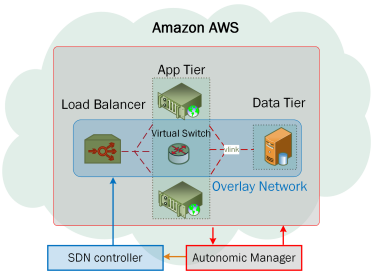

Figure 1: *Experiment setup on Amazon AWS.*

on AWS using the setup shown in Figure 1; the first one presented in Section III-A serves as the baseline that illustrates the hill-climbing heuristic previously proposed in [1]; the second one in Section III-B demonstrates our model-based approach and shows how it successfully overcomes the limitations of the heuristic one.

For the application, we used a three tier ebookstore application previously reported in [2]. Users send requests and wait for the reply and then think for some time (i.e., think-time) and send the next request. Application scenarios include: *adding to shopping cart*, *payment*, and *browse*. Each application scenarios has its own response time SLA. We adapt the bandwidth rates of responses from the load balancer node toward the clients. We performed the experiments using two scenarios Scenario 1 being *adding to shopping cart* and Scenario 2 *browse the catalog* as we trained our model using these two scenarios. Table I illustrates the parameters set for the scenarios in the experiments.

Table I: *Parameters set for the scenarios in all experiments. The autonomic manger is to maintain the 95^{th} percentile response time SLAs.*

Scenario	95^{th} SLA	CT	TFLV	$heat_o$	$heat_u$
Add to cart (Scenario 1)	40 ms	0.9 * 40 ms	1 min	2	10
Browse (Scenario 2)	700 ms	0.9 * 700 ms	1 min	2	

A. Experiment 1: Hill-climbing Heuristic

Figure 2.a presents the CPU utilization of the load balancer, application servers and database server that can be read from the left vertical axis. The horizontal axis shows the experiment iteration number (or the monitoring time) in all graphs. The autonomic manager monitors the application every 20 seconds. The number of application servers are shown at the right vertical axis. Figure 2.b shows the workload which corresponds to the number of users in each scenario and Figure 2.c illustrates

the bandwidth rates of the scenarios. Note that we only show the scenarios whose bandwidth have been adapted during the experiment. Unless mentioned otherwise, the bandwidth rates are the default values, which is set to 220Mbps. Figure 3.a shows the response time and arrival rate of Scenario 1 and Figure 3.b displays those of Scenario 2 where the response times can be read from the left vertical axis and the arrival rates from the right. The Red dashed lines in both of these graphs depict the 95^{th} percentile response time SLAs. The pink shaded background shows a bandwidth adaptation event and the gray shaded background illustrates an auto-scaling event. At around iteration 74 in Figure 2.b, the number of users of Scenario 1 increased from 59 to 69 which violates the SLA of scenario 1. Since the response time stays above the threshold for two consecutive iterations, autonomic manager starts off by reducing a proportional amount from the current bandwidth. We found that 15% of current bandwidth would be a good value w.r.t sizes of scenarios. The autonomic manger reduces the bandwidth rate of scenario 2 in Figure 2.c multiple times which finally fixes the response time of Scenario 1 at around iteration 86 in Figure 3.a.

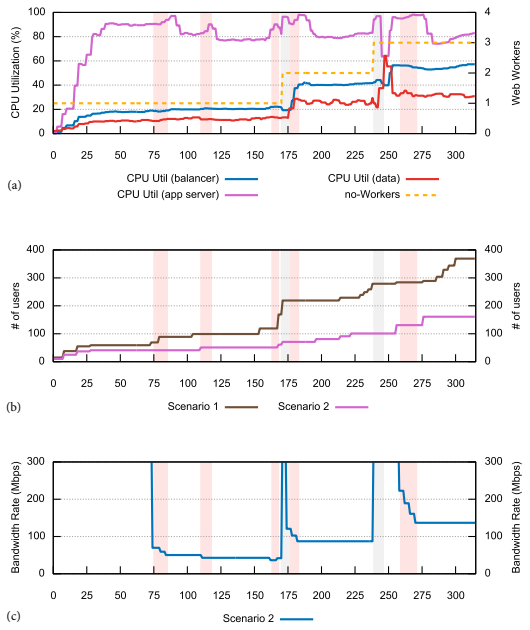

Figure 2: *Using the hill-climbing heuristic, bandwidth of Scenario 2 is adapted.*

Other adaptations happen after around iterations 111, 162, and 168 in Figure 2.c which successfully fix the violations. Then at around iteration 169, due to hight number of users, response times of both scenarios go above their SLA thresholds which triggers the auto-scaling policy. A new VM is added to the application tier increasing the number of servers from 1 to 2 at around iteration 172 in Figure 2.a. After a new VM is added, the bandwidth rates are reset. Even though a new VM is added to the application, the response time of Scenario 1 still tends to stay high which then triggers the bandwidth adaptation. We can see from Figure 2.c that the bandwidth

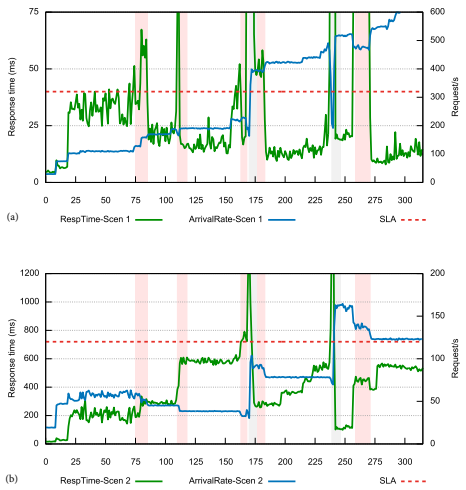

Figure 3: *Response time of scenarios in Hill-climbing heuristic.*

rate of scenario 2 is adapted three times until the response time of scenario 1 is fixed at around iteration 184. Due to large volume of requests, the autonomic manger adds another VM to the application tier to fix the response times at around iteration 240. Then at around iteration 258, the response time of scenario 1 goes over its SLA. We can see in Figure 2.c, the bandwidth of scenario 2 is adapted multiple times which finally fixes the response time of scenario 1 at around iteration 272.

B. Experiment 2: Model-based Adaptation

We keep the experiment setting including the workload and the application configuration exactly the same as previous experiment. In Figure 4.b, we can see that as time goes by, the number of users in both scenarios are increasing which increases the response times of both scenarios. Up to around iteration 62 in Figure 5.a, the response time of scenario 1 goes a few times above its SLA. However, since the policy is acting on two consecutive iterations, no actions are taken at these times. Around iteration 62, the number of users of scenario 1 increases (see Figure 4.b) which causes the response time of Scenario 1 to stays up for 2 iterations above its SLA. Using the model, autonomic manager applies the predicted bandwidth rates to fix the response time of Scenario 1. As can be seen in Figure 4.c, the bandwidth rate of scenario 2 is reduced from the default bandwidth to around 69Mbps. We can observe that the response time of scenario 1 gets fixed at around iteration 68. Again at around iteration 74, the bandwidth of scenario 2 is lowered to around 50Mbps that resolves the SLA violation of scenario 1.

At around iteration 105, the number of users in Scenario 1 increases further up to 119 users that breaches SLA of scenario 1. At this time, bandwidth adaptation criteria do not hold (i.e., response time of Scenario 2 is above CT). This means that the current application capacity does not support this workload and bandwidth adaptation cannot be performed. Hence autonomic manager scales out the application to restrain

further SLA violations (the number of application workers increases to two in Figure 4.a.) Now let's see if the model is still able to predict the bandwidth rates correctly after this scaling out event. When the number of users in both scenarios is increased from iteration 117 to 124, it breaches SLA of Scenario 1. After the response time of scenario 1 stays up for two back-to-back iterations in Figure 5.a. When the model output is actuated over the load balancer interfaces, we can see that the response time of scenario 1 gets adjusted at around iteration 127.

Next, more users are using Scenario 2 which then violates the response time of scenario 1 at around iteration 148. So the bandwidth rate of scenario 2 is reduced to 80Mbps which improves the response time of scenario 1 quickly at around iteration 149. It can be observed from Figure 4, that all SLAs are well maintained despite the increase in the workload up to around iteration 189, where both scenario response times gets violated, and the autonomic manger has no choice but to scale out the application. Hence the application is scaled out again (the number of web workers in Figure 4.a goes up to 3). However, it takes around 5 iterations for the response times of both scenarios to improve after the new VM is added. At around iteration 212, bandwidth of scenario 2 is adapted which then fixes the response time of Scenario 1 at around iteration 214. The number of users in both scenarios is increased further at around iteration 230 and onward, as we can see the autonomic manger is able to successfully maintain the SLAs of application scenarios which brings us to the end of the experiment.

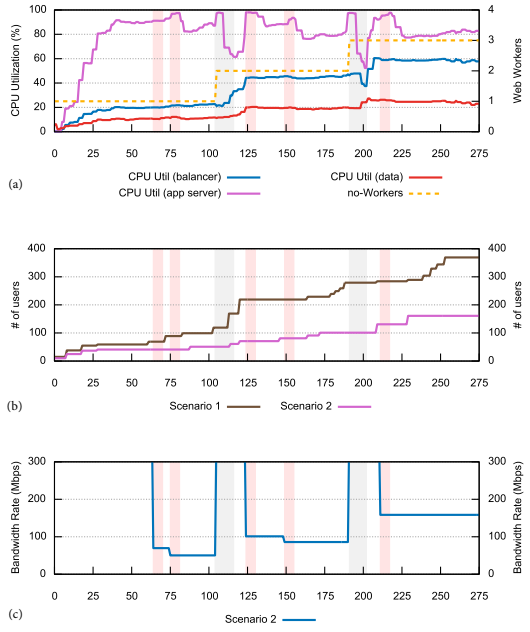

Figure 4: *Using the model, bandwidth of Scenario 2 is adapted.*

C. Analysis of the Results

The model-driven solution employs an on-line model and can quickly and accurately estimate the appropriate bandwidth

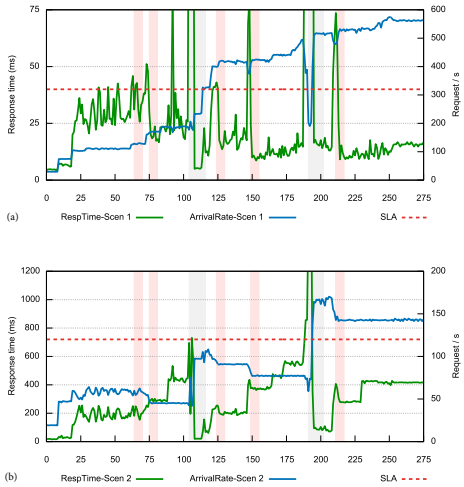

Figure 5: *Response time and arrival rates in model-based.*

rates that resolves SLA violations. From the results in the two experiments, we calculated the number of SLA violations of Scenario 1 that includes the two iterations that the autonomic manger waits to prevent oscillation as well. The model-based autonomic manger reduces the number of SLA violations by 56% compared to the heuristic one. Only the model-based satisfies the 95^{th} percentile SLA while the heuristic one only conforms to the 89^{th} percentile SLA response time. Compared with VM auto-scaling, the time to effect of bandwidth adaptation is much lower. Hence, we can conclude that bandwidth adaptation is an effective and efficient mechanism to adapt the response time of applications and using a model improves the quality of adaptation.

IV. Related Work

Iqbal et al. [4] utilize polynomial regression to model the number of servers in a tier as a function of the number of static and dynamic requests received by the RUBiS web application. Mirza et al. [5] apply machine learning to predict throughput of TCP using support vector machine. Wickboldt et al. [6] propose a design of a cloud platform that puts network on the same level with computation (CPU) and storage (disk) resources; this way the client applications can dynamically provision and de-provision network as needed.

V. Conclusion

In this paper, we presented a model-driven application autonomic manager for web applications in cloud utilizing smart bandwidth management within application cluster. Through experiments on AWS, we demonstrated that our adaptive control can quickly maintain SLA response times and reduce the SLA violations by 56% compared to the heuristic-based approach.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Council of Canada (NSERC) CGSD.

REFERENCES

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

[2] C. Barna, H. Ghanbari, M. Litoiu, and M. Shtern, "Hogna: A platform for self-adaptive applications in cloud environments," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10th International Symposium on*, May 2015, pp. 83–87.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[4] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 871–879, Jun. 2011.

[5] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to tcp throughput prediction," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '07. ACM, 2007, pp. 97–108.

[6] J. A. Wickboldt, L. Z. Granville, F. Schneider, D. Dudkowski, and M. Brunner, "Rethinking cloud platforms: Network-aware flexible resource allocation in iaas clouds," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, 2013, pp. 450–456.