Adaptive Service Management for Cloud Applications Using Overlay Networks

Nasim Beigi-Mohammadi, Hamzeh Khazaei, Mark Shtern, Cornel Barna and Marin Litoiu
Department of Computer Science, York University
Toronto, Ontario, Canada
Email: {nbm, hkh, mark, cornel, mlitoiu}@yorku.ca

***Abstract*—This paper presents an adaptive service management mechanism that maintains service level agreement through use of overlay networks that are deployed over the cloud provider network. The application autonomic manager strives to maintain the SLA without provisioning new resources for as long as possible. Through continuous monitoring and analysis, autonomic manager uses software defined networking (SDN) to dynamically apply policies to the flows of requests that travel through the application components. We implement and evaluate the proposed method on a hybrid cloud environment. Through extensive experiments, we show that the management mechanism can successfully maintain the SLA of services while it avoids provisioning extra resources which is the common approach in cloud.**

***Index Terms*—Adaptive applications, SDN, Overlay networks, SLA, Bandwidth.**

I. Introduction

Service Level Agreements (SLA) represent the contract which captures the agreed guarantees between a service provider and its customers. In a volatile environment such as cloud, it is not easy to maintain the quality of service (QoS) specified in the SLA while avoiding over provisioning [1]. Robust mechanisms should be in place to guarantee SLA compliance for different types of services in cloud.

The most common approach to maintain SLA at high load or partial failure is to leverage the cloud elasticity feature that provides on demand provisioning of computing/storage resources [2]. With mature virtualization technologies in computing, applications can easily adjust their computing/storage demands in cloud. However, adding more computing resources translates into higher cost for application providers as well as increased carbon footprint in cloud data centers. Moreover adding/removing computing nodes is not an easy task for all applications; for example NoSQL datastores as well as big data analytic platforms such as Apache Spark[1] can not be scaled without experiencing non-negligible overhead associated with data replication, maintaining consistency or job rescheduling [3]. More specifically in down scaling scenario, the *decommissioning* process of resources might take a long time which renders the whole downsizing ineffective [4], [5].

With advancements in network virtualizations, applications can leverage overlay networks without being locked in a specific cloud provider. Then using SDN, the flow of application requests through a service can be controlled dynamically based on application policies and requirements. An overlay network on top of cloud provider network brings about more flexibility to application managers to have a fine granular control over their flows.

Hence, in this paper, we propose a design, implementation and an algorithm for an application autonomic manager that leverages overlay networks and SDN to dynamically control the bandwidth of application flows to meet the SLAs of its services. The idea of using bandwidth provisioning to improve SLA is not new; however, in this paper, we are controlling the bandwidth on application scenario[2](or service) level . In a nutshell, we strive to respond to the following research question:

Research Question: *is it possible for a distributed application on cloud to autonomically use bandwidth control mechanisms for imposing delay on selected flows to maintain SLA withou scaling out?*
While, generally, the intuitive strategy to resolve overload or bottleneck problem in cloud is to increase the amount of resources including bandwidth, it has been shown that reducing the bandwidth may solve the overload problem in some scenarios [6], [7].

Therefore, in this paper, we investigate the idea of regulating SLAs through imposing delay on some flows within applications overlay networks. To this end, we propose an adaptive management mechanism for applications on cloud that takes advantage of network programmability and network virtualization to improve the overall performance while reducing the cost and footprint of applications. We deploy overlay networks and virtual endpoints on different components of the application. Network controller programs the virtual end points on each application node according to the application policies. The management mechanism uses a heuristic to select flows of some services that can tolerate delay to improve the response time of other services. Thus, we present an end-to-end architecture of the management mechanism and implement it on a hybrid cloud. Through extensive experiments, we illustrate that our mechanism improves the overall application performance without imposing extra cost to application owners. Therefore, we answer the research questions with following contributions:

[1]http://spark.apache.org

[2]We use scenarios and services interchangablely hereinafter

- We propose a fine granular badwith control mechanism for cloud applications that improves the application performance without increasing the cost for applications owners.
- We implement and evaluate our cloud-agnostic mechanism in a hybrid cloud setting where extensive experiments are carried out to show the feasibility and advantages of our badwith management methodology.

The remainder of the paper is organized in these sections: in Section II, we present the end-to-end architecture of the management mechanism and explain the overall strategy through an algorithm. Section III presents the implementation and results of our solution on real clouds. In section IV, we overview the related work. And finally, Section V concludes the paper.

II. Methodology and Architecture

Although computing/storage adaptations (i.e., scaling in/out VMs and containers) have been shown to meet application requirements, they are not always the best solution; from application owners point of view, adding more resources translates into higher cost. At the same time, it raises the environmental effects of data centers. Besides, it is challenging for a number of applications such as Big Data and NoSQL data stores to scale out/in due to various reasons including data inconsistency, job rescheduling etc. [3], [4], [5]. Therefore, in this paper, we propose a different approach to meet application performance requirements by imposing delay on some of the services to meet SLA response time of other services within the application overlay network. The idea has the following explanation: by delaying some of the application services/scenarios, we shorten the queues to some congested resources and allow other services/scenarios to pass faster through the queues. More specifically, we propose to delay flows of services that have response times well below their SLAs. To implement the idea, we design and implement a management mechanism that uses overlay networks and SDN to dynamically throttle some services aiming to improve the response times of other services whose SLA is near violation. Our management mechanism is able to postpone adding extra computing resources to the application as long as possible. Our solution is cloud agnostics and applications can manage their network independent of cloud provider network because the application is deployed on top of the cloud provider network.

Figure 1 shows the high level architecture of our management mechanism. The autonomic manager follows the monitor-analyze-plan-execute-knowledge (MAPE-K) loop [8] to mange the *managed system*. The managed system consists of the application along with its virtual links and virtual endpoints. The actions include *bandwidth adaptation* and *add/remove* VMs.

In our solution, the autonomic manager automatically builds the application topology; it first instantiate the application nodes and then deploys an overlay network that connects the application components through creating virtual tunneling endpoints (VTEPs) and virtual interfaces (vNIC) on the nodes.

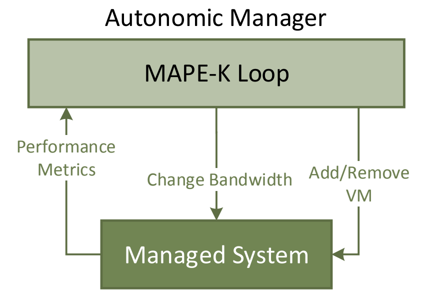


Figure 1: *Application autonomic manager and the managed system; autonomic manager manages the bandwidth and VMs.*

After establishing the overlay network, an SDN controller programs the overlay network on the fly based on application autonomic manager policies. The management mechanism can impose delay on any link within the overlay network to improve the response time of the services whose SLA are near violation. In order to achieve this objective, each application component should be able to distinguish different services. Therefore, before flows leave the egress interface of the application nodes, they are classified into different classes of services as shown in Figure 2; any node of the application that is to apply bandwidth policies should classify the flows based on services. The classification can be implemented on any number of application nodes. Network controller can then program the virtual ports on each node implementing classification to delay certain flows. Figure 2 illustrates how the classification can happen at any node of the application. Depending on the method used to apply bandwidth policies, a declassification component might also be needed not to interfere with application layer processing.

A. Autonomic Manager

The autonomic manger continuously monitors the response time of application services and checks if the SLA of any service is about to be violated. If that is the case, actions will be planned to handle the situation that are basically as follows:

1) The autonomic manager first tries to correct the response time by following a greedy hill climbing heuristic to manipulate the bandwidth of application flows.
2) If the badwith control was not successful, the application is scaled out to prevent SLA violation.

Hill climbing heuristic tries to maximize (or minimize) a target function. At each iteration, it adjusts a single element and determines whether the change improves the value of the target function. Similarly, when the response time of a service is near violation, our management mechanism at every iteration searches among application flows to find a flow that meets certain criteria. If it can find such a flow, it slows the bandwidth of the chosen flow one step down with the goal of improving the response time of a service whose SLA is about to be violated. If no such flow is found, it checks if flows of this service has been delayed previously. If yes, it increases the bandwidth of that flow one step up and checks if this action resolves the problem. The heuristic repeats these actions until it exhausts all its options. If the problem is not still resolved, the management mechanism adds extra resources to prevent SLA violation. The management mechanism is summarized in Algorithm 1. Before getting into the details of the algorithm, let us define some thresholds that are used in the heuristic upon which the autonomic manager defines the actions:

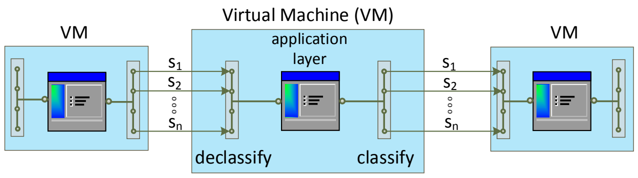


Figure 2: *Classification and declassification of services based on the management policy; flows are classified into services and receive a specific bandwidth.*

- **SLA threshold**: it defines the maximum legitimate service response time;
- **Trigger threshold**: the goal of the management mechanism is to maintain the response time of each service below its trigger threshold. When response time of a service reaches its trigger threshold, it triggers our bandwidth adaptation mechanism.
- **Candidacy threshold**: this threshold determines if it is possible to delay a service. If the response time of a service is above its candidacy threshold, its flow will not be selected to be delayed.
- **Selection criteria**: The flow selected to be delayed has to meet these two conditions: (a) the service that this flow belongs to should have response time below the candidacy threshold; (b) the flow should have the highest throughput compared to other flows in the service.

In Algorithm 1, when the response time of a service violates its trigger threshold (line 2), the autonomic manager first performs the hill climbing heuristic with a greedy selection criteria to find a flow to delay (line 5). If such a flow does not exist, the algorithm checks if the service whose response time is about to be violated, has been delayed in previous iterations. If it finds such flows within the service, it goes one step back and increases their bandwidth one step up one by one (line 12). If the algorithm exhausts all its search options, it adds extra VMs to prevent the SLA violation (line 15). In addition, the algorithm continuously checks the CPU utilization of the application. If it reaches below certain threshold, it removes the extra resources (line 24).

Algorithm 1: Bandwidth adaptation Algorithm:

```
input : $\mathcal{S}$—vector of services.
input : $\mathcal{R}$—vector of average response times for services.
input : $SLA$—vector of response times SLA for services.
input : $trigger$—vector of service response time thresholds that
        trigger bandwidth adaptation
input : $candidacy$—vector of service response time thresholds
        based on which a service is selected to slow down
input : $CPU^{lo}$—low CPU utilization threshold.
input : $CPU$—average CPU utilization of web servers.
input : $\mathcal{H}$—vector of heats for services, where $h_s$ is the heat for
        service $s \in \mathcal{S}$.
input : $heat$ — a control number for removing VM
input : $n, N$ — the number of consecutive violations required to
        trigger an adaptation for bandwidth and VMs respectively.

1  foreach service $s \in \mathcal{S}$ do
2      if $R_s > trigger_s$ then
3          if $h_s = n$ then
4              $h_s \leftarrow 0$;
5              $f \leftarrow \{F \in \mathcal{S} - \{s\}$ who meets selection criteria$\}$;
6              if $f \neq \emptyset$ then
7                  Decrease bandwidth for $f$;
8                  return;
9              else
10                 $F \leftarrow$ {flows belong to service $s$ whose bandwidth
                     can be increased};
11                 if $F \neq \emptyset$ then
12                     Increase bandwidth for one flow in $F$;
13                     return;
14                 else
                       // bandwidth adaptation
                          exhausted all options
15                     Add VM;
16                     return;
17         else
               // move one step toward bw adaptation
18             $h_s \leftarrow h_s + 1$;
19     else
           // reset any buildup for bw adaptation for
              this service
20         $h_s \leftarrow 0$;
21 if $CPU < CPU^{lo}$ then
       // cluster underload
22     $heat \leftarrow heat - 1$;
23     if $heat = -N$ then
24         Remove VM;
25         $heat \leftarrow 0$;
26         return;
```

III. Implementation and Experiment

We implemented our solution on a hybrid cloud environment. Hybrid cloud environment can be used to provide more flexibility to application owners [9]. Also, we chose a hybrid cloud setting to show our solution is cloud agnostic and does not depend on underlying cloud infrastructure due to using overlay networks. We used Amazon EC2 as the public cloud and Smart Applications on Virtual Infrastructure (SAVI) cloud [10] as the private cloud. The application is hosted on Amazon EC2 public cloud and consists of a three-tier cluster using Apache 2.0 as load balancer, an eBook store web application in Tomcat 7, and a MySQL database. The application provides four services or use cases:

- **browse**: the user browses through the book catalog and clicks on various items to see the details;
- **buy**: the user adds a book to the shopping cart;
- **pay**: the user checks out and pays for the content of the shopping cart;
- **auto bundle**: upselling / discounting service, where the user receives the opportunity to bundle together related books based on the item currently viewed.

A. Experiment setup

Autonomic manger dynamically deploys an overlay network over SAVI and Amazon networks where all the application nodes are connected using Virtual Extensible LAN (VXLAN) technology [11]. Each node uses a virtual switch implementation where bandwidth policies can be configured on virtual ports on the nodes. We use Open Virtual Switch (OVS) [12] and its rate policing mechanism to apply the desired bandwidth rates. We have implemented the classification mechanism on *proxy* shown in Figure 3. The implementation of classification on other nodes within the application cluster is left for our future work.

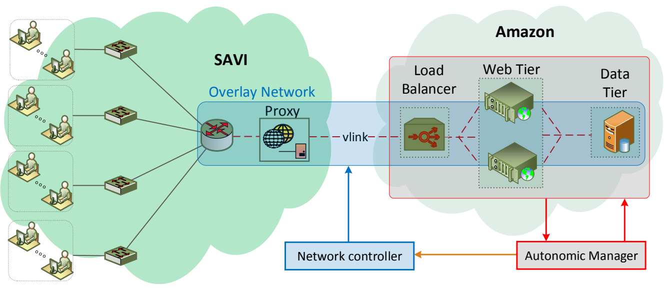


Figure 3: *Deployment model of the System for experiments.*

We have assigned users into four groups that are using the application's services. In our implementation, users from each group are behind a NAT gateway and application proxy sees only one IP address per group. We implemented this to be in tune with real scenarios where users from a household, department etc. use application services. In future, the grouping policy can be used to differentiate between various users (i.e., SLA per user). However, in this work, we do not differentiate between users in a group in terms of SLA. To emulate users, we use a workload generator such that users periodically send requests to different application services. When they receive the reply, they think for a random period of time (i.e., 500-550 ms as *think time*) and then send the next request. The number of active users in the system changes in range of [5 ⋯ 67] during the experiment. Table I shows the overall distribution of users in four groups.

Table I: *Distribution of users in groups.*

Group	G1	G2	G3	G4
Population	31%	36%	30%	3%

Table II: *Candidacy, Trigger, and SLA thresholds of response time for each application Service.*

Service	Candidacy thresh	Trigger thresh	SLA
Buy	158 ms	175 ms	250 ms
Browse	360 ms	400 ms	572 ms
Auto Bundle	540 ms	600 ms	857 ms
Pay	900 ms	1000 ms	1430 ms

B. Experiment Results

In this Section, we show how our bandwidth management mechanism helps the application to maintain its SLA while avoiding adding extra resources as long as possible. We create three major events in which we explore the bandwidth management mechanism. Hence, we first explain what we intend to show in each event. We then discuss each event in more details.

Event 1: in the beginning of the experiment, we show how by *decreasing* bandwidth of some services, our management mechanism maintains the desired response time for all services.

Event 2: in the second event, we demonstrate how the management mechanism successfully maintains the service response times below the desired threshold through *increasing* bandwidth of some flows. We show how our mechanism maintains the CPU utilization of web application as low as possible.

Event 3: in this event, we first show a situation where bandwidth adaptation mechanism exhausts its all options and can no longer improve the response times. In such a situation, the autonomic manager scales out the application and adds a VM to maintain the SLAs. After a while, the workload is decreased and since all response times are well below their SLA, the autonomic manger scales in the application by removing extra VMs.

Figure 4a shows the CPU utilization of application web worker and database on the left vertical axis and the number of web workers on the right vertical axis. Figure 4b shows the bandwidth actions that has been applied to different flows of services. The bandwidth rate has been adjusted in the range of [75 ⋯ 450] kbps as higher bandwidth rates did not have any impact on the flows' response time in our application. In Figure 4b, we only show the flows whose bandwidth has been changed during the experiment. The three shaded areas in Figure 4b represents the three events respectively.

Figure 5a shows the service response time of *Pay* service on the left vertical axis and the arrival rate of requests for the service on the right axis. Figure 5b depicts the service response time and arrival rate of requests for *Auto Bundle* service. Figure 5c, and 5d show the service response times and request arrival rates for *Browse* and *Buy* services respectively. The horizontal axis in all 6 figures illustrates the experiment iteration number where autonomic manager collects the monitored data, analyzes and initiates an action based on the monitored data. The duration of each iteration may depend on the frequency of monitoring and it can range from seconds to minutes. In our experiment, each iteration is set to one minute. In all plots in Figure 5, we have highlighted three horizontal bands for response time axis. The lower band (i.e., white color) indicates normal response time where no action needs to be taken. The middle band (i.e., dark gray) shows the trigger area in which autonomic manager is triggered to bring the response times back to the white area. And the upper band in Orange is the range at which the SLAs are violated. The autonomic manager should try to keep response times away from this zone. The SLA, trigger and candidacy thresholds for all services are presented in Table II. In our experiment, we set this trigger threshold to be 70% of the SLA threshold and set candidacy threshold to be 90% of the trigger threshold. Following we explain the various events of the experiments.

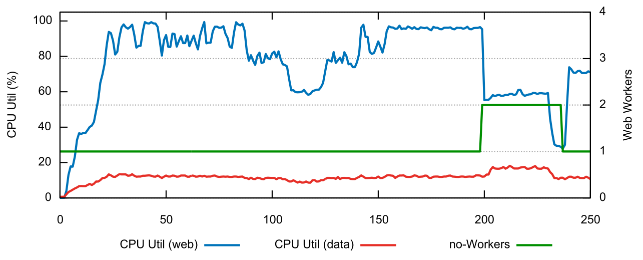
C. Event 1

In the beginning of the experiment, as shown in Figures 5a, 5b, 5c, 5d, up to iteration 25 all response times of services are below the trigger line and hence no adaptation is performed. At iteration 27, the request arrival rate for *auto bundle* service increases (see Figure 5b), and, as a result, the response time of *buy* service (Figure 5d) reaches the trigger line. Our algorithm does not take action right away, instead it waits for two more iterations to make sure the high response time is not a transient effect preventing the ping-pong effect. After the response time of *buy* service remains above trigger line for two more iterations, the algorithm starts to perform bandwidth adaptation at iteration 30. From iteration 30 to 32, the algorithm first chooses *pay* service to reduce the bandwidth of its flows because it meets the selection criteria. It can be seen in Figure 4b, the bandwidth of *pay* service goes one step down. After this action, since response time of *buy* service has not been corrected yet, at iteration 32, the algorithm performs the second adaptation and reduces bandwidth of *auto bundle* service that meets the selection criteria, too. We can see that at iteration 33, the response time of *buy* service is corrected and is below its trigger line. In addition, we can see in Figures 5a and 5b, during these iterations, *pay* and *auto bundle* services are delayed and hence their arrival rates decrease accordingly.

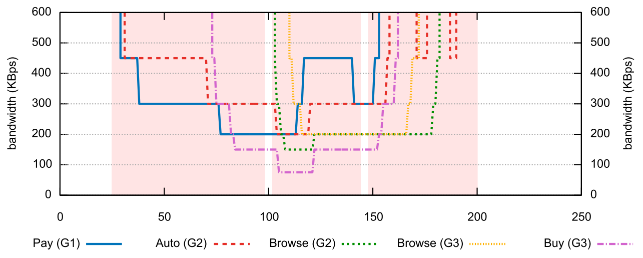
Again at iteration 37, the response time of *Buy* service reaches its trigger line and stays the same for 3 consecutive iterations. The algorithm chooses *pay* service to reduce its bandwidth that brings the response time of *buy* service to the white area. As can be seen in Figure 4b, the bandwidth of *pay* service flow goes one step down at iteration 39.

By applying the bandwidth management technique, we can see that up to iteration 72, the response times of all services remain below the trigger line and no action is taken up to this point. At iteration 72, the response time of *buy* service violates the trigger line, and hence the heuristic chooses *auto bundle* service to reduce its bandwidth (see Figure 4b). However, this adaptation does not correct the response time of *buy* service, therefore another adaptation action is selected which reduces the bandwidth of a flow belonging to *pay* service one step down. At iteration 74, the algorithm fixes the response time of *buy* service.

Around iteration 78 the response time of *auto bundle* service is in trigger range (ie dark gray) so that autonomic manager tries to compensate this by reducing the bandwidth of *buy* service by two steps, *pay* service by one step and again *buy* service by two steps. As can be seen, all above steps make the response time for all services in the normal range (i.e., white area) from iteration 80 to iteration 100.



(a) *CPU utilization and number of web workers.*



(b) *Bandwidth of services.*

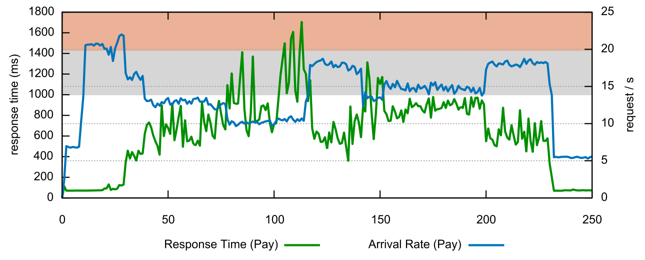
Figure 4: *CPU utilization, no of web workers and bandwidth adaptation. Shaded areas represent the bandwidth actions in 3 scenarios.*

D. Event 2

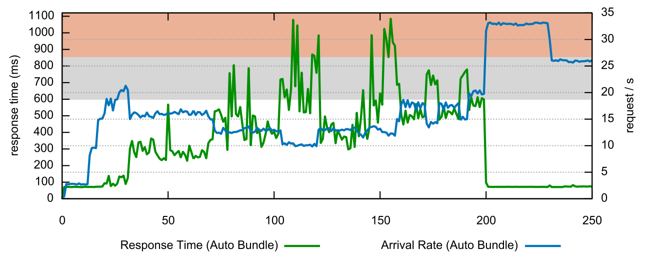
After iteration 100, we can see that the response time of *pay* service increases sharply and tends to remain above for a couple of iterations. Therefore, the heuristic tries to fix this by lowering the bandwidth of *buy*, *auto bundle* and *browser* services. Despite such actions, the response time of *pay* service still does not get below its trigger line. Hence, the algorithm increases the bandwidth of *pay* service two steps up one by one (see Figure 4b) at iteration 112 and 115 that fixes the response time of *pay* service at around iteration 117 (Figure 5a). Meanwhile the response time of *auto bundle* and *browse* services goes above their trigger lines due the delay imposed to them in previous iterations (Figures 5b and 5c). The algorithm increases their bandwidth one step up at around iteration 122 and 125. We can see that their response times improve and get below their corresponding trigger lines. Now, at around iteration 125, we can observe that our mechanism successfully maintains the services in good conditions for more than 20 iterations while the CPU utilization remains below 80%.

E. Event 3

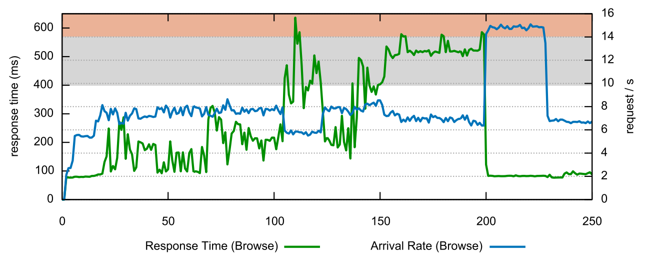
We increased the request arrivals of *buy* and *browse* services at around iteration 140 which then increases the response time of all services. The algorithm tries to correct the response times by managing the bandwidth within the services. It can successfully fix the response time of *pay* and *auto bundle* services by increasing their bandwidth step by step (see iteration 150 in Figure 4b). We can see that their response times get below their trigger lines at iteration 155 and 160 respectively. However, the response time of *browse* and *buy* services still remain high despite the algorithm trying to increase their



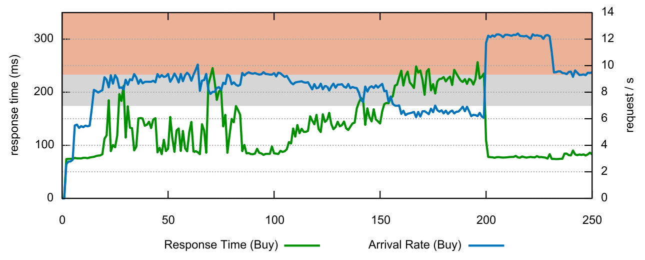
(a) *Pay service.*



(b) *Auto Bundle service.*



(c) *Browse service.*



(d) *Buy service (aka add to cart).*

Figure 5: *Response time and arrival rate of services during the experiment; the response time axis has been divided into three bands: (a) the white band indicates normal service time (b) gray shows the trigger area and (c) orange area highlights the SLA violation area.*

bandwidth (see Figure 4b). At iteration 195, the algorithm exhausts all bandwidth adaptations which leads to scaling out by adding one VM. The right axis in Figure 4a shows that the number of web workers is increased to 2 at iteration 198. The third shaded area in Figure 4b represents Scenario 3 up to iteration 200. Afterwards, a VM is added and the response time of all services get below their trigger line at iteration 200. We can see that our algorithm delayed adding more VMs as long as possible by managing bandwidth to adhere SLAs.

At the end of experiment, the workload is decreased and since the response times of all services are good, the autonomic manager scales in the application by removing one VM while SLAs are still maintained. It can be seen that the number of workers in Figure 4a goes down from 2 to 1 at around iteration 240.

IV. Related Work

One associated concern with adding/removing VMs is to perform it in a timely manner that has been proved to be challenging [13]–[18].

The idea of network management with respect to application policies have been discussed in [9], [19], [20]. Also with the emergence of SDN, cloud providers can expose APIs to cloud users so that users can manage their networking resources the same way they manage their computing and storage resources as also suggested in [21], [22]. However, most public cloud users do not have access to the cloud SDN APIs, if there is any [23].

Wickboldt et al. [22] 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. Most of the related work use cloud provider network to make dynamic networking configurations. However, our cloud agnostic solution takes advantage of overlay networks on top of cloud provider network that brings about higher flexibility and control to applications to manage their networking. In addition, our mechanism is a versatile solution where an autonomic manager continuously monitors the response times of distributed cloud applications and make corrective actions as needed by communicating to SDN controller. Using adaptive bandwidth management, our mechanism manages to maintain SLA while postponing adding extra resources which is the common approach for applications in cloud.

V. Conclusion

In this paper, we presented design, implementation and an algorithm for managing application performance in complex and dynamic cloud environments by dynamic management of network bandwidth. The proposed approach is based on adaptive bandwidth management that strives to maintain the response time SLAs across all services of an application. This is accomplished by applying flow control policies at the service flow level, which is orchestrated by an application autonomic manager within an overlay network. When bandwidth management is exhausted, the autonomic manager provisions new computing resources. We implemented and evaluated the proposed method on a hybrid cloud environment and showed that the management mechanism is able to successfully meet the application's SLA objectives.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Council of Canada (NSERC) CGSD and the Ontario Research Fund for Research Excellence under the Connected Vehicles and Smart Transportation (CVST) project.

References

[1] H. Khazaei, J. Misic, and V. B. Misic, "Performance analysis of cloud computing centers using m/g/m/m+r queuing systems," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 5, pp. 936–943, 2012.

[2] L. Zhao, S. Sakr, and A. Liu, "A framework for consumer-centric sla management of cloud-hosted databases," *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 534–549, July 2015.

[3] H. Khazaei, M. Fokaefs, S. Zareian, N. Beigi-Mohammadi, B. Ramprasad, M. Shtern, P. Gaikwad, and M. Litoiu, "How do i choose the right nosql solution? a comprehensive theoretical and experimental survey," *Accepted in Journal of Big Data and Information Analytics (BDIA)*, 2016.

[4] 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.

[5] M. Smit, M. Shtern, B. Simmons, and M. Litoiu, "Partitioning applications for hybrid and federated clouds," in *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '12. IBM Corp., 2012, pp. 27–41.

[6] K. Xu, K. Tang, R. Bagrodia, M. Gerla, and M. Bereschinsky, "Adaptive bandwidth management and qos provisioning in large scale ad hoc networks," in *Military Communications Conference, 2003. MILCOM'03. 2003 IEEE*, vol. 2. IEEE, 2003, pp. 1018–1023.

[7] M. Welsh and D. E. Culler, "Adaptive overload control for busy internet servers," in *USENIX Symposium on Internet Technologies and Systems*. Seattle, WA, 2003, pp. 4–4.

[8] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2005.

[9] N. Beigi-Mohammadi, C. Barna, M. Shtern, H. Khazaei, and M. Litoiu, "CAAMP: Completely automated DDoS attack mitigation platform in hybrid clouds," in *International Conference of Network and Service Management (CNSM)*. IEEE, 2016.

[10] SAVI, "Smart applications on virtual infrastructure," http://www.savinetwork.ca/.

[11] "VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," online, Access date: 14 Sep. 2016. [Online]. Available: https://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-00

[12] "Open vSwitch," online, Access date: 16 Sep. 2016. [Online]. Available: http://openvswitch.org/

[13] S. K. Mahalingam and N. Sengottaiyan, "Energy aware resource management in distributed cloud computing with overload avoidance," *Journal of Computational and Theoretical Nanoscience*, vol. 13, no. 1, pp. 50–57, 2016.

[14] Y. Liu, *Chameleon: Virtual Machine Migration Supporting Cascading Overload Management in Cloud*. Springer International Publishing, 2016, pp. 129–145.

[15] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, vol. 23, no. 3, pp. 567–619, 2015.

[16] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "Energy-efficient resource allocation and provisioning framework for cloud data centers," *IEEE Transactions on Network and Service Management*, vol. 12, no. 3, pp. 377–391, 2015.

[17] A. Nahir, A. Orda, and D. Raz, "Resource allocation and management in cloud computing," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 1078–1084.

[18] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future generation computer systems*, vol. 28, no. 5, pp. 755–768, 2012.

[19] N. Beigi-Mohammadi, H. Khazaei, M. Shtern, C. Barna, and M. Litoiu, "On efficiency and scalability of software defined infrastructure for adaptive applications," in *3^{th} IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2016.

[20] G. Wang, T. E. Ng, and A. Shaikh, "Programming your network at runtime for big data applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 103–108.

[21] W. Rankothge, J. Ma, F. Le, A. Russo, and J. Lobo, "Towards making network function virtualization a cloud computing service," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 89–97.

[22] 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.

[23] "Designing Virtual Network Security Architectures," online, Access date: 16 Sept. 2016. [Online]. Available: https://www.rsaconference.com/writable/presentations/file_upload/csv-r03-designing_virtual_network_security_architectures.pdf