Improving Response Time Stability of SDN with *Pending State*

Byungsoo Ko, Hayoung Choi, Jaewon Kim, Mingyu Jin, Sanggyu Nam
Korera Advanced Institute of Science and Technology
{kobiso, hayoungc, jwkim1993, jmingyu, sanggyu.nam}@kaist.ac.kr

ABSTRACT

Software Defined Networking has become a popular paradigm for centralized control in many modern networking environments such as data centers and cloud computing. To address the limitations of single controller, many researches have been done in SDN, and ElastiCon, which is our key paper, is one of them. ElastiCon automatically balances the load across controllers, and provides good response time regardless of traffic conditions. Instead of static mapping between controllers and switches, ElastiCon dynamically expands or shrinks the controller pool effectively by using switch migration. However, the experiment scenario in the paper is too naive to reflect the real pattern such as bursty traffic. Therefore, we propose a novel way to prevent unstable response time from exceptional traffic conditions. We devise the concept of pending state to minimize the boot-up delay. To show our experiments, we implemented our model by using Floodlight and Mininet which is the same as ElastiCon. The result indicates that the pending state is very effective to stabilize response time regardless of traffic patterns with minimal overhead.

1. INTRODUCTION

The advent of IoT (Internet of Things) has made massive amounts of data from billions of devices. In the present, researchers estimate that 9 billion devices are connected, and even 24 billion devices will be connected within 5 years [5]. The extension of the Internet is needed by many reasons. For example, transmission of information from IoT devices should be adapted to new post-PC services such as sensor virtualization, edge computing and cloudlet. However, the current IP-centric Internet infrastructure and network protocols suffer from several limitations such as scalability, flexibility and manageability.

To overcome such limitations, the Software Defined Networking (SDN) concept has been proposed. SDN can be defined as an emerging network architecture where the network control is decoupled and separated from the forwarding mechanism and is directly programmable [4]. The key scheme of SDN is separation between control plane and data plane. Centralized control plane(single controller architecture) enabled easier management and faster innovation. However, large-scale networks consist of hundreds of severs connected with thousands of switches cannot be easily managed by a single controller. In addition, efficiency is not guaranteed enough with just one controller, and if one adversary steal the access rights of the controller, it loses the whole management over the network.

Therefore, various architectures leveraging the multi-controller concept have been proposed. Every research tries to maximize the strength of multi-controller SDN architecture. Among them, ElastiCon [2] focused on a load-balancing by switch migration without performance loss. In addition, unlinke other concepts' static mapping between controllers and switches ElastiCon dynamically expand or shrink a controller pool according to traffic patterns. However, it still contains a problem where applied rules are not really elastic because of several threshold values. Also, the period of decision whether power on a new controller or power off it is 3 seconds which is quite long when there is worst case traffic. Thus, their algorithm cannot deal with a real world traffic properly. In other words, only one controller can be turned on regardless of situation, which may cause a huge impact on response time, or availability. In this paper, we discuss about ElastiCon in the sense of response time, and propose a new scheme to solve this limitation.

2. RELATED WORK

This research is based on the paper of ElastiCon [2] which maintains controller pool very dynamically and provides load balancing by switch migration. The paper also propose switch migration protocol to reduce the switching overhead. Although ElastiCon architecture provide stable respons time as the packet-in rate increases, it is limited to reflect the realistic traffic parttern. The idea of our approach to handle this problem is influenced by the auto-scaling mechanism for virtual resources in cloud computing by Ahn et al. [1]. Therefore, we briefly introduce about two researches in this section.

ElastiCon [2] is an architecture in which the controller pool is dynamically grown or shrunk according to traffic conditions. It maintains the cluster of physically distributed controllers, which is managed by distributed data store. It also proposes the 4-phased protocol for safe switch migration, the interface between controllers and applications for application state migration, and the load adaptation algorithm which involves resizing the controller pool and rebalancing the load. The algorithm checks every 3 seconds whether the set of currently active controllers can afford the current traffic load by balancing the average utilization of all controllers, and by adding or removing controllers when needed. In balancing process, the best switch migration is identified as which results in the most reduction in standard deviation of utilization across controllers. After then, resizing process is triggered for each overloaded or underloaded controllers. Overloaded controllers turn on another controller to divide loads, and underloaded controllers turn off an other controller to bear its load.

Ahn et al. [1] suggests an auto-scaling mechanism for virtual resources in cloud computing. They consider deadline-critical real-time medical data generated by sensor-based medical devices and point out that there are issues to be resolved caused by the fluctuation of the amount of data over time. To tackle those problems, they propose a mechanism to predict the volume of future data. It is based on their observation that most objects monitored by sensor-based devices typically show symptoms before transitioning to an abnormal state. If the root Real-Time Application (RTA) server does not have enough computing resources for all of given real-time tasks, it assigns those tasks to its child RTA server. It checks the available computing resources against a given real-time task by calculating the projected system response time and comparing it with a given absolute processing deadline. The root RTA server acts as an auto-scaling controller. In case of long boot-up delay in launching a new RTA server, a module named analyzer predicts the future resource usage by applying a moving average filter on the resource monitoring result and using the slope of the moving average over time. To avoid suboptimal resource utilization during the drastic oscillation of slope, the system introduces multiple logical states whose transition condition is based on the current state and the slope. On certain states, a new child RTA server is launched and pending real-time tasks will be assigned to it, or an existing child RTA is terminated and intercepts the workload of it.

3. PROBLEM STATEMENT

Figure 7 shows an effect of the resizing process in ElastiCon. At a quick look, it seems to work properly because the response time maintains relatively constant regardless of growth and shrinkage of the controller pool. However, the traffic pattern in the figure is a too naive one to reflect a real world situation since it goes up and down gradually over time. In the reality, the traffic can dramatically increase for example, when there happens massive ticketing, course registration, or sports online broadcasting. Furthermore, it does not consider a common situation where the traffic fluctuates around a certain amount. The fluctuating size of the controller pool can often cause a huge problem on response time.

As seen in Figure 2, the resizing decision depends simply on the fixed thresholds: *HIGH_UTIL_THRESH* and *LOW_UTIL_THRESH*. If values of the two thresholds are not set properly, turning-on and -off methods can be called alternately even in a normal traffic condition. In such a case, the response time will increase since controllers have boot-up delay. When controllers turn on, they need some time to set the environment followed by switch migration. This high delay hampers dynamic resizing of the controller pool. The fluctuation of the controller pool must not occur because the response time can be affected which is the important criterion in SDN. Therefore, we need to improve response time stability even in the bursty traffic situation.

4. DESIGN CONSIDERATIONS

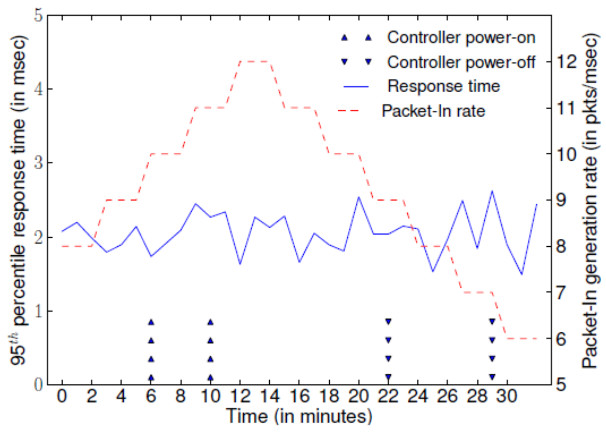

Figure 1: Effect of resizing on ElastiCon

Algorithm The resizing algorithm

```
procedure DORESIZING()
    for all $c$ in $controller\_list$ do
        if $c.util \geq HIGH\_UTIL\_THRESH$ then
            SWITCH_ON_CONTROLLER()
            return True
        end if
    end for
    $counter \leftarrow 0$
    for all $c$ in $controller\_list$ do
        if $c.util \leq LOW\_UTIL\_THRESH$ then
            $counter \leftarrow counter + 1$
        end if
    end for
    if $counter \geq 2$ then
        SWITCH_OFF_CONTROLLER()
        return True
    else
        return False
    end if
end procedure
```

Figure 2: Resizing algorithm of ElastiCon

In this section we describe the desired property of an SDN architecture designed for an environment with bursty traffic situation. In order to achieve the desired property, we devise three different approaches, verify whether they satisfy the desired property, and compare them.

4.1 Desired Property

Response Time. Response time in the context of computing is the total amount of time it takes to respond to a request for service. In general, response time in packet-switched networks is defined as a sum of the network latency and the time it takes to process the request in the destination. The way an SDN system controls the packets affects the network latency. In an SDN architecture with distributed controllers, the time it takes to resize its controller pool is one of the factors. An SDN system can handle more amount of network traffic by having multiple controllers available in the pool, which makes a distributed approach for an SDN architecture scalable. An SDN system may resize its controller pool according to network traffic conditions, then migrate some switches from one controllers to another. In general, this takes more time than simply forwarding a packet to one of existing controllers in the pool. The boot-up delay in a controller accounts for some part of the resizing time if an SDN system decides to turn on a controller which is powered off for resizing the controller pool. Also, the resizing time for the controller pool may affect the response time since some switches may have to wait for the end of migration.

4.2 Comparison of Possible Approaches

Various aspects on resizing the controller pool can be considered. Resizing algorithm should be scalable, which means it should work properly even when the controller pool becomes fairly large. It also should provide resource saving property and reasonable (or shorter) response time. Active controllers themselves consume power energy, and turning on and off them consistently wastes power, computation, bandwidth resources due to load rebalancing and switch migration.

In this paper, we consider the response time most as emphasized in Section 3. The architecture, still, should work on real world situations which may be common or exceptional. For example, the controller pool should be immediately expanded when the Packet-In rate increases drastically, and it should not be immediately shrunk because the Packet-In rate may be fluctuating sometimes. Otherwise, the system could not be able to handle all the requirements because of boot-up delay. Three simple solutions can be come up with to address this problem as follow.

4.2.1 Delayed turning-off

The response time can be stabilized by minimizing boot-up delay when controllers need to turn on. One way to minimize boot-up delay is to maintain some active controllers which do not manage switches. Those controllers are just in waiting status, so that they can be easily used without boot-up process whenever packet requests increase rapidly. another way is to delay turning-off when the node turns off its power. In the delayed-time, It can easily cancel turning-off to handle sudden increase of packet requests.

This has also scalability since only need to be changed is turning-off time. Power consumption gets higher than Elasticon's, but when the delayed time, controller does and manages nothing and staying in idle state. Therefore, by the number of idle state controllers increases, power consumption growth will be inevitable.

4.2.2 Dynamic threshold

ElastiCon sets fixed thresholds to determine whether the controller pool needs to grow or shrink. The fixed thresholds cannot properly deal with exceptional situations. Managing the threshold dynamically can be a solution for this problem. Supposing a situation when the Packet-In rate increases rapidly, we just lower the threshold so that the controller can be eager to turn on. On the other hand, we can increase the threshold when there is not many of packet requests.

This has also scalability because every node decides whether they turn their power off or on by their own threshold and only need to do is to change fixed thresholds to dynamic thresholds. The response time also can be lower since more nodes will be turned on during sudden spikes of requests than the original one. However, deciding a level of threshold requires extra system for monitoring the overall network traffic and mis-judged threshold can make the response time worse. Also, it can not solve the boot-up delay of controllers.

4.2.3 Simultaneous turning-on by tendency

The algorithm of managing controllers in ElastiCon checks load degree of every controller so that it can scale up or down up to the situation. However, examining controllers one by one can be critical when the Packet-In rate increases drastically. In order to solve this problem, we can turn on multiple controllers at the same time regarding to *tendency*. By calculating the tendency of Packet-In rate increase, we can turn on amount of controllers proportional to the tendency.

In order to use the solution, whole request packets are gathered and then calculated. Therefore, it needs central server to monitor all the requests on the network. It also takes extra time to calculate tendency and transfer the tendency result to each controllers. It can't also solve the boot-up delay.

5. PROPOSED SCHEME

Among those approaches, we chose "Delayed turning-off" to solve our problems. Compare with other approaches, "Delayed turning-off" has no ambiguity in algorithm such as level of threshold. It can also solve boot-up delay to reduce response time.

In this section, we devise pending state between active and inactive state to deal with delaying turn-off. We explain what the pending state is, and suggest algorithm for the pending state.

5.1 Pending State

As addressed in Section 5.1.1, activated controllers cannot handle requests of switches immediately since it requires the environment to be set. In the proposed scheme, controllers do not directly go to inactive state when they decide to turn off themselves. Instead, they go to pending state first to be prepared for potential request increase. We introduce the concept of using pending state and examine the benefits of the scheme as followed.

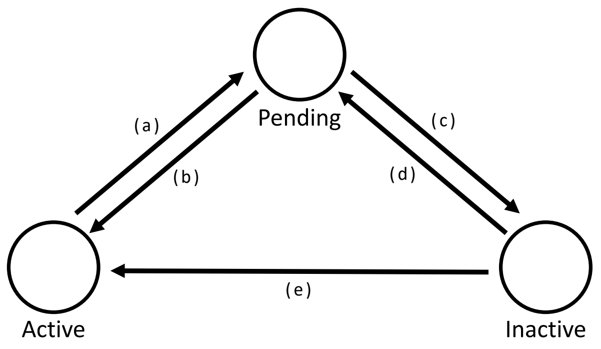

Figure 3: The transition diagram of a controller

As seen in Figure 3, there are three state of a controller in the architecture: active, inactive, and pending state. In active state, a controller normally handles requests of switches. In inactive state, a controller does not handle any requests because its power is off. In pending state, it also does not handle any requests, but its power is still on. It is a kind of ready state where if the amount of requests on the network increases, it immediately changes its state into active state without any turn-on delay.

Load measurement module periodically collect load measurement of each controller node. When the amount of requests on the network decreases, a controller of ElastiCon directly goes to inactive state, but that of the proposed scheme goes to pending state to be ready for potential request increase (a). If the set of currently active controllers cannot handle all the requests, controllers in pending state turn into active state and switches are migrated to them (b). Extra controllers in pending state still can fall to inactive state when pending state lasts for a certain amount of time for saving resources (c). Controllers in inactive state can directly go to active state when lack of controllers in pending state (e). However, the proposed architecture maintains

$$the\ number\ of\ currently\ using\ controllers \times x\%$$

of controllers in pending state in order to cope with a sudden increase of requests on the network (d).

The main advantage of this scheme is that we can reduce turn-on delay from inactive state so that the stable response time even in exceptional cases can be achieved. In addition, it makes the architecture flexible to deal with various situations such as sudden increase, fluctuation of the amount of requests, and etc. The flexible architecture provides higher availability than the former one, ElastiCon, with controllers in pending state.

5.2 Algorithm

The main loop of the algorithm should consider the new state $PENDING$ in the implementation of the state changes made to the network. For this reason, this scheme modifies the implementation of EXECUTE_POWER_ON_CONTROLLER and replaces the invocation of EXECUTE_POWER_OFF_CONTROLLER with the invocation of a newly defined function EXECUTE_SET_CONTROLLER_PENDING.

Algorithm 1 Modified function for resizing controller pool

```
procedure DORESIZING
    for all c in controller_list do
        if c.util ⩾ HIGH_UTIL_THRESH then
            if pending state exists then
                SWITCH_ON_PENDING_STATE_CONTROLLER()
            else
                SWITCH_ON_INACTIVE_STATE_CONTROLLER()
            end if
            return TRUE
        end if
    end for
    Counter ← 0
    for all c in controller_list do
        if c.util ⩽ LOW_UTIL_THRESH then
            Counter ← Counter + 1
        end if
    end for
    if Counter ⩾ 2 then
        MAKE_CONTROLLER_TO_PENDING_STATE()
        return TRUE
    else
        return FALSE
    end if
end procedure
```

We change the resizing algorithm of Elasticon which decides to turn on and off controllers. Main differences between the original function with Algorithm 1 are twofold. First, when the number of request is over threshold, original function just turn on random inactive controller, but in Algorithm 1, if pending state controller exists, the system makes pending state controller active first. if not, it's the same with original. Second, when the system decides to turn off a controller, it calls not SWITCH_OFF_CONTROLLER() but MAKE_CONTROLLER_TO_PENDING_STATE() to delay turn-off.

Algorithm 2 Modified function for powering on a controller

```
procedure EXECUTE_POWER_ON_CONTROLLER
    c ← the controller to be powered on
    if c.state = INACTIVE then
        Power on c
    else if c.state = INACTIVE then
        Stop c.timer
    end if
end procedure
```

The modified function EXECUTE_POWER_ON_CONTROLLER, described in Algorithm 2, checks the state of the controller to be powered on and determines if the system should actually power on the controller or just change the state of the controller. In the latter case, the timer associated with the controller is manipulated. This timer is explained in the next paragraph.

Algorithm 3 New function for setting a controller pending

```
procedure EXECUTE_SET_CONTROLLER_PENDING
    c ← the controller to be set pending
    procedure TASK
        c.state ← INACTIVE
        EXECUTE_POWER_OFF_CONTROLLER(c)
    end procedure
    c.timer ← schedule TASK after the delay PENDING_DURATION
end procedure
```

We define a new function in this algorithm, named EXECUTE_SET_CONTROLLER_PENDING. This function, shown in Algorithm 3, defines a task which sets the controller which is set to be at a pending state in the current loop inactive and powers off this controller. Then, this function schedules this task to be executed after the predefined delay and associates the timer for this schedule with this controller. This implementation lets a controller which is set to be at a pending state be powered off after a specific amount of time unless there is a request to power on a controller before the timer ends.

6. IMPLEMENTATION

The evaluation of our solution should be compared with the baseline work, ElastiCon. Above all, we have to implement the same experiment environment of ElastiCon and then, modify it with our proposed scheme. The ElastiCon was implemented by using Hazelcast(an open source in-memory data grid based on Java), Mininet [6] and Floodlight [3] controller(an open SDN controller which supports Open Flow). Mininet is a network emulator which creates

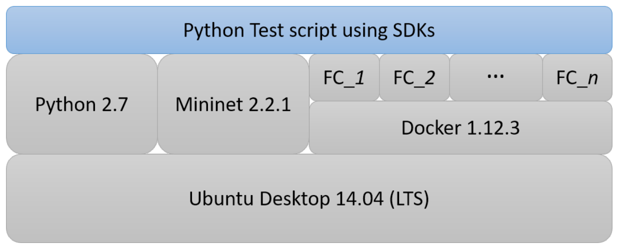

Figure 4: Implementation environment

a network of virtual hosts, switches, controllers and links. They have modified the network emulation framework and the base controller in order to implement ElastiCon methodologies upon them. However, we simplified the implementation of ElastiCon since making the exact same thing with ElastiCon is impossible. Also, the main purpose of our work is verifying the effect of pending state. Therefore, the core part of ElastiCon is enough.

Figure 4 shows an implementation environment of our proposed scheme. We implemented our proposed scheme by adding pending state of ElastiCon. Our scheme is built on Ubuntu Desktop with Floodlight controllers. To make multiple floodlight controllers, we made spawning docker which are run and stopped dynamically as the traffic changes. By using mininet, we constructed the topology of controllers and switches. In order to control the overall activation, the proposed SDN architecture is managed by Python test script using SDKs. The implementation we have done so far is on the Github repository[1].

7. EVALUATION

In this section, we set two cases of scenarios to evaluate our proposed scheme such as bursty packet-in traffic and consistent fluctuation of packet-in traffic. And we evaluate the performance of our proposed scheme by comparing with the ElastiCon prototype and then present our experimental results.

7.1 Experimental Scenarios

7.1.1 *Bursty packet-in traffic*

The controllers receive lots of packets in a very short period of time as described in figure 5. In this situation, many number of controllers need to newly turned on in order to handle this situation. However, since there will be much amount of boot-up delay, response time will be increased and there will be problem to deal with the situation.

7.1.2 *Consistent fluctuation of packet-in rate*

The amount of packets keep up and down around a certain point as described in figure 6. In this situation, controllers need to be turned on and off repeatedly. This will increase the response time because of boot-up delay and it will be problem to deal with the situation.

7.2 Experimental Results

First of all, we constructed the experiment environment by using Floodlight controller, mininet and docker as Elas-

[1]https://github.com/kyukyukyu/cs543-final/tree/simulation

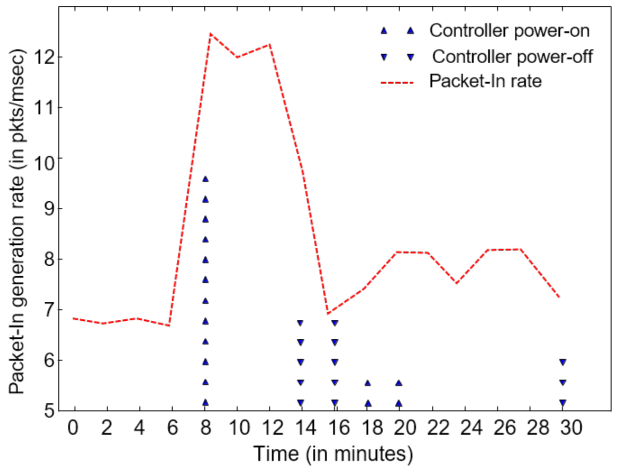

Figure 5: Bursty packet-in traffic

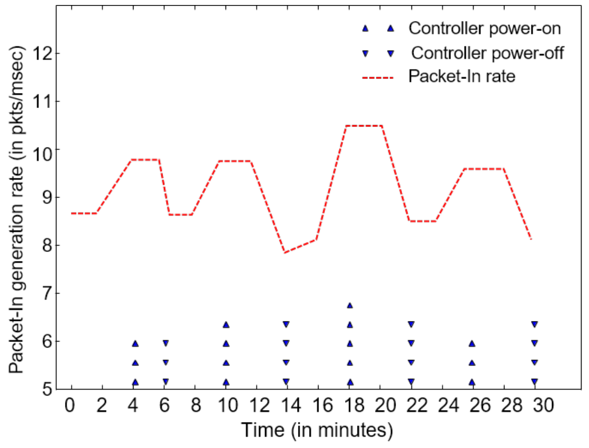

Figure 6: Consistent fluctuation of packet-in rate

tiCon did and implemented load balancing adaptation algorithm in a multi-controllers environment. By load balancing algorithm, if a CPU utilization of a controller went up and detected hard-burden for itself, the algorithm turns on a new controller and migrates some of switches to it. If a CPU utilization of a controller went down and detected wasteness, the algorithm turns off the controller and migrates all of switches of it to other controllers. We tested the load balancing algorithm by sending packets to other host and we could validate the load balancing algorithm of our simulator working well.

Figure 7 shows average response time of each packet when the Packet-In rate changes. We generated Packet-In requests constantly and monitored the response time of them. Most of the Packet-In rates are under 0.03 and their response times are under 7 millisecond. That is, about 30 packets are generated per second and most of the packets are handled in 7 millisecond.

However, we could not finish our experiment. Even though we increased the amount of packets movement on the network as much as we could, CPU utilization of each controllers was unpredictable. At the first time of turning a controller on, CPU utilization is about from 30% to 40%.

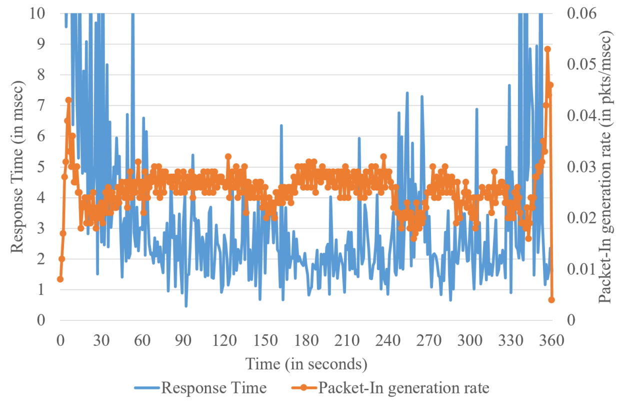

Figure 7: Response time transition by packet-in rate

And when it becomes stable, it goes down until from 0% to 10%. But the problem is that it sometimes bounces until 30% in stable state and it does not grow enough even though the amount of packets movement increases a lot. These problem made us impossible to decide proper threshold for the experiment.

8. CONCLUSION AND FUTURE WORK

With the increasing number of Internet-connected devices, users expect various requirements to be provided on the network architecture according to their applications or services. SDN can be a solution of their needs, so there have been many researches on SDN architecture. ElastiCon is a logically centralized architecture, which elaborates multiple controllers for scalability, flexibility and availability.

However, they evaluated their architecture on naive scenarios that cannot represents real world situations such as massive online events and traffic fluctuation around a certain amount. In this case, boot-up delay of controllers may hinder instant response to the increased traffic because extra controllers are already turned-off. Thus, we suggest a new scheme that a logical state is added to a controller where it is powered on but ready to take instant action when increased traffic is detected. We can reduce response time of controllers by removing turn-on delay with this scheme, and it is possible to take instant response to rapid increase or fluctuation in traffic.

However, we only deal with the situation of bursty traffic and flunctuation of requests that we made. As a future work, more realistic traffic patern should be needed such as real network capture file by using wireshark. In other words, synthetic traffic could not completely reflect the real world situation such as online ticketing. This will trigger various situations that we could not expected. Also, we can use packet generator tools such as Scapy for fine-tunes control. How to properly set threshols can be another issue. Optimal HIGH_UTIL_THRESH must be different depending on the traffic. Otherwise, our proposed scheme will not work quite well since we assumed ElastiCon already set threshold appropriately.

9. REFERENCES

[1] Y. Ahn, A. Cheng, J. Baek, M. Jo, and H.-H. Chen. An auto-scaling mechanism for virtual resources to support mobile, pervasive, real-time healthcare applications in cloud computing. *IEEE Network*, 27(5):62–68, 2013.

[2] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Elasticon: an elastic distributed sdn controller. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 17–28. ACM, 2014.

[3] D. Erickson. Floodlight java based openflow controller. *Last accessed, Ago*, 2012.

[4] O. Foundation. Software-defined networking (sdn) definition. http://goo.gl/O2eTti.

[5] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[6] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.