Anomaly Detection using Resource Behaviour Analysis for Autoscaling systems

Rajsimman Ravichandiran, Hadi Bannazadeh, Alberto Leon-Garcia
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
Email: {rajsimman.ravichandiran, hadi.bannazadeh, alberto.leongarcia}@utoronto.ca

***Abstract*—In a cloud environment, autoscaling systems alleviate applications when additional resources are required. However, an illegitimate or malicious workload may force the system to automatically provision resources when they are not needed, thus leading to two key problems: economic denial of sustainability (eDoS) and wastage of resources. In this paper, we propose an anomaly detection mechanism using resource behaviour analysis to prevent these issues. We build univariate autoregressive statistical models to analyze resource behaviours for each microservice on the platform. The use of multiple models helps us discern unusual anomalies rather than a sudden increase in certain properties. We implemented the anomaly detection for the Elascale autoscaling engine on SAVI Testbed and evaluated the detection mechanisms against different attacks. From the results, we conclude that the models can accurately detect anomalous behaviour for applications (with cyclical trends) on the autoscaling platform.**

I. Introduction

Autoscaling is one of the most essential components required in cloud environments. The end-user/owner of the application does not need to decide when to up/down-scale resources. The system will automatically provision resources during peak periods and downscale when the applications become idle or do not utilize many resources. On the rise in microservice-based architectures, we developed the Elascale autoscaling solution to monitor both micro/macroservices [1]. A microservice is basically a container that runs a specific component of an application/service. For example, a web application server can have multiple components such as user interface (UI), database, server back-end etc. In that case, the UI can be a microservice. Subsequently, we define a macroservice as basically a virtual machine (VM) that runs multiple microservices of the same application. For instance, all pieces of the web application (explained above), can run on a single macroservice. Hence, if more microservices are being scaled to meet demands, the macroservice that hosts the modules must scale as well. Elascale monitors both micro/macroservices and scales them based on their resource usage. We deployed it on a cloud research platform (SAVI Testbed) and open-sourced the tool on GitHub for the public research community/industry to utilize [2].

Autoscaling may also be used to thwart (Distributed) Denial of Service (DDoS) attacks, as it can withstand/absorb application layer traffic attacks by provisioning more resources [3]. However, this causes a major drawback. First, they obey being forced to pay for those scaled resources (which leads to eDoS), even though the application has been serving a majority of useless requests. Second, the infrastructure provider wastes resources to handle the unnecessary workload, which could have been used for other applications. For example, Google's autoscaling system can be tricked by Yo-Yo based DoS attacks [4], which produces eDoS for owners and wastes resources.

The attacks may not be limited to DDoS attacks. An intruder may simply stress the system (using Linux tools) to increase CPU utilization and deceive the autoscaling system to provide more resources. We implemented a variant of this attack on a microservice and defined it as a *Microservice Stress Attack*. We discuss these attacks further in the evaluation section.

We specifically focused on resource patterns with cyclic trends as applications that involve human interactions usually contain this type of usage. For example, [5] explored workload patterns of a public web server from Michigan State University, which encountered around a million requests during the period. The access intensity peaked at 10 AM and stays high until late afternoon and decreased over time to the lowest point at 3 AM. The pattern repeated each day during the investigation period. Hence, we simulated traffic patterns with cyclic trends for our experiment setup in order to emulate real-world application use cases.

The main contribution of this paper is an anomaly detection mechanism that utilizes autoregressive models to perform resource behaviour analysis on microservices. By performing time series based forecasting, we can detect anomalies on cyclic resource usage patterns. This greatly helps autoscaling systems in reducing resources wastage during eDoS attacks.

The organization of this paper is as follows. In section 2, we present the background concepts for our work in this paper. Related work is presented in section 3. In section 4, we describe the system design of our anomaly detection. In section 5, we discuss the evaluation of our solution and in the final section, we present our conclusion and future work.

II. Background

In this section, we briefly describe key background research work to this paper.

A. Smart Applications on Virtual Infrastructure

Smart Applications on Virtual Infrastructure (SAVI) Testbed is a multi-tiered, distributed platform that provides resources for research and development on future network protocols and architectures. It spans eight Canadian universities, and it is composed of several Core data centers and many "Smart Edge" data centers [6]. In addition to compute resources, it contains various virtualized heterogeneous resources such as Field Programmable Gate Arrays (FPGAs), Graphical Processing Units (GPUs), Software-Defined Radios (SDRs) etc. The SAVI Testbed architecture is based on Software-Defined Infrastructure (SDI) concepts. It is built using OpenStack open-source cloud computing framework and Software-Defined Networking (SDN) for networking. We use the SAVI Testbed to research, develop and implement our Elascale autoscaling solution (as well as anomaly detection) for macro/microservices deployed on the platform.

B. Elascale

Elascale is an autoscaling engine (deployed as a Docker microservice) that scales resources for both micro/macroservices. It is part of the platform that monitors, stores and visualizes collected metrics. Hence, we refer to the environment as the Elascale platform. The architecture of the Elascale platform is shown in figure 1. Instead of designing our platform from square one, we incorporated multiple open-source tools that adhere to our requirements to build our system. Elascale uses Docker Swarm for clustering, scheduling, and scaling microservices. Subsequently, it uses Docker machine to manage and provision macroservices. Moreover, Elascale UI shows visualization dashboards along with autoscaling configuration options (such as threshold policies) for the application user. Metricbeat and Dockbeat monitor macroservices and microservices (respectively). Elasticsearch stores the monitoring statistics and Kibana is used for presenting data visually. Furthermore, NGINX controls access and encrypts communication channels using (Secure Sockets Layer) SSL certificates (to prevent eavesdropping/tampering) for Elasticsearch and Kibana.

III. Related Work

In this section, we describe previous work in eDoS mitigation methods and resource behaviour analysis for autoscaling systems deployed in the cloud.

1) eDoS Mitigation Methods: [7] provides a mitigation solution for eDoS by directing suspicious traffic to a scrubber service (secondary location) and use client puzzles to detect legitimate users. The solution works on web server applications and suspicious traffic is determined by threshold policies (eg. requests/sec). This solution mainly depends on the applications running on the platform and it intercepts the interactions of the application. This may be seen as intrusive or dependent on the application layer. [8] uses two-step methods to mitigate eDoS: use a virtual firewall with blacklist and use Turing Test to verify suspected users. As malicious users are detected, the blacklist gets updated. The solution uses a random check to choose suspected users and furthermore, the blacklist can grow exponentially when it is attacked by a horde of botnets.

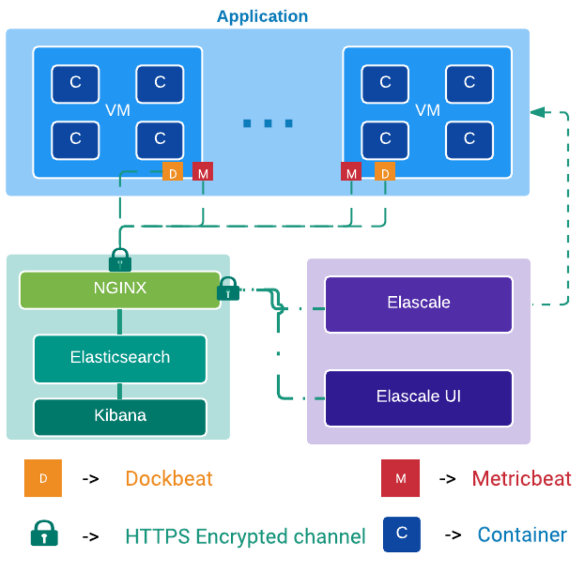

Fig. 1. Elascale Platform Architecture

2) Resource behaviour analysis: [?] utilizes characteristic curves of VM's resources including CPU usage, network, disk usage and uses a pre-motion step to forecast the properties. This allows the VMotion scheduling system to efficiently place VMs without affecting the quality of service (QoS) of other running applications. VMotion is VMware's proprietary product and scheduling strategies are different for other platforms. In [9], Jiang et al. proposed a scheme for web applications in the cloud. They used machine learning to analyze the historical distribution of web requests and forecast future workload and autoscale accordingly. A drawback to this innovation is it works only for web applications.

IV. System Design

In this section, we present the design details of the anomaly detection mechanism for Elascale autoscaling engine.

A. Proposed Approach

We tackled the anomaly detection problem by analyzing historical datasets of the microservices resources and build statistical models based on their "normal" behaviours. We define "normal" behaviour as actions performed by the application when handling a legitimate workload. We assume all requests or interactions with the application were performed by justifiable users or systems. Once the models are constructed, we use them to detect anomalies by forecasting resource behaviours and evaluating them against actual values. If the real value differs by some threshold from the expected behaviour, then it is considered an anomaly. Note that we build univariate statistical models for each resource in order to accurately detect anomalies. In further sections, we describe the design methodology using only CPU resource statistics. But, the same approach can be used to build models for other resources as well.

B. Experiment Setup

In order to train the model, we wanted to use a real production-based workload. However, we were unable to find a dataset that profiled regular users and particular attack vectors that we were evaluating against our system. Hence, we created a web server and traffic generators on SAVI Testbed (based on OpenStack-based cloud environment). This imitates a web application on a cloud platform with legitimate users accessing the web server. We deployed an NGINX Docker container on a m1.medium flavor-based (5 GB memory, 20 GB disks, 2 vCPUs), VM with an Ubuntu 16.04 image. The container is a simple HTTP server that serves a static web page upon request. For traffic generators, we wrapped a simple stress tool [10] on a Docker container and deployed three instances on a different VM (with the same specifications as above). Figure 2 showcases the setup of our application. Furthermore, we deployed these two VMs on different physical servers in order to account for traffic delay, throughput and processing latency in network interfaces and in-between switches and routers.

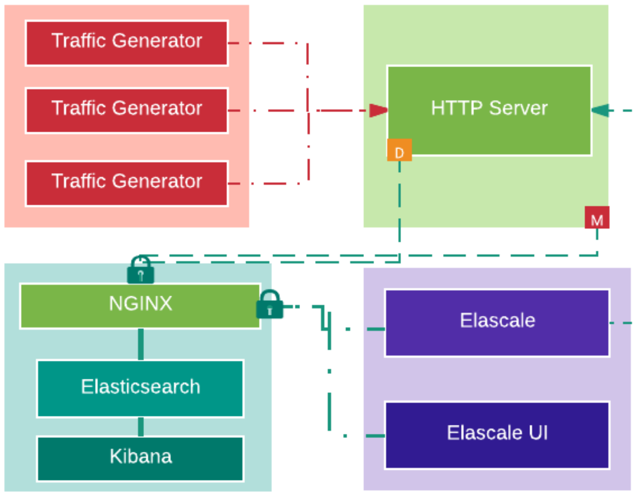

Fig. 2. Experiment Design Architecture

In order to get some historical dataset, we perform this experiment for 14 hours. The number of hours is selected based on accuracy/processing time trade-off (explained in implementation section). As mentioned in earlier sections, Dockbeat collects microservice statistics every ten seconds. However, the dataset is too granular to find key patterns for our time series models. Hence, we downsample the dataset by taking an average of all six samples occurring within a minute.

C. Model Forecast

After downsampling the dataset, we build different statistical Autoregressive (AR) models (based on lag periods) in order to evaluate the forecast predictions. First, we separate the dataset for training and testing with 90/10 ratio. We train an AR model on the first 90% of the dataset and predict the rest of the 10%.

In order to evaluate our model, we use mean square error (MSE), root mean square error (RMSE) and mean absolute error (MAE). We compared models with different lags and as shown in table I, the simplest model AR(p=1) outperforms other lagged models based on accuracy and processing time.

TABLE I
AR MODEL COMPARISON

Criteria	AR(p=1)	AR(p=10)	AR(p=20)
MSE	4.95×10^{-4}	5.94×10^{-4}	7.69×10^{-4}
RMSE	2.18×10^{-2}	2.42×10^{-2}	2.75×10^{-2}
MAE	1.47×10^{-2}	1.67×10^{-2}	2.04×10^{-2}
Processing Time (secs)	4.46×10^{-3}	10.79	427.99

Hence, we finalized our forecasting model to be AR(p=1) model. Figure 3 showcases the results of our predicted using our AR model and observed values for the last 60 minutes.

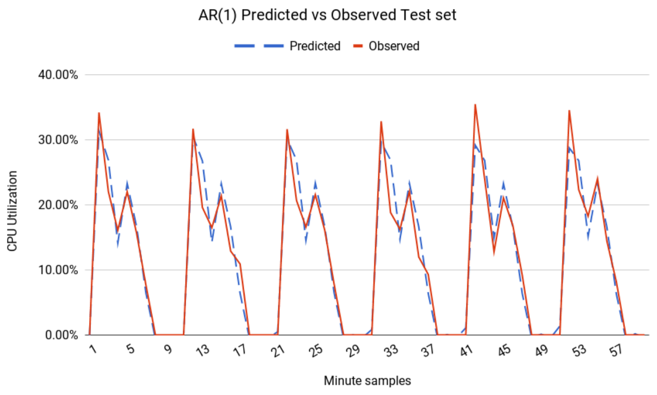

Fig. 3. Predicted vs. Observed plot

D. Implementation on Elascale

In order to build the AR model, we use the last 14 hours of data (from the triggered event) to create the model and test it against the last few samples. We chose the size of the dataset based on two criteria: accuracy and processing time. In other words, how accurate can the dataset represent the resource usage patterns and how long does it take to process the dataset. From our research, we found that 14 hours of data accurately represents the application's patterns and can be processed within 5 seconds. Furthermore, we configured Dockbeat logs to not exceed 10 MB per file and keep only 7 files on the system. Hence, the size of the dataset that needs to be processed is always 70 MB for any microservice. Therefore, the anomaly detection mechanism is scalable for this architecture. The design is summarized in algorithm 1.

As shown in the algorithm, we chose **0.15** to be the error threshold as this value produced the least false positives and negatives (based on ROC curves for other values). Furthermore, we create multiple, univariate AR models for

Algorithm 1: Anomaly Detector Algorithm

```
1  if high or low cpu_usage then
2      - Get last 14 hours of data from Dockbeat logs;
3      - Downsample into 1 minute interval bins;
4      - Split 90/10 Training and Testing data;
5      - Build AR model and predict last few values;
6      - $error = abs(predicted - observed)$;
7      if $error >= 0.15$ then
8          Report it as anomaly
9      else
10         Return False
```

different properties. We use algorithm 1 to build a model for network received packets/sec data (network_rx) as well. Since the traffic generators create more requests to increase CPU usage, the network traffic increases simultaneously. Hence, both network and CPU utilization have very similar cyclic trends, in terms of usage. The only differences we made to network dataset was normalizing the data (because of high variance between data points) and having a different error threshold between predicted and observed value (based on ROC curves using network data). Hence, we can use multiple models and check for anomalies on different resources. The implementation methodology of the models to Elascale is shown in algorithm 2. Based on our dataset, we chose 20% to be considered high CPU usage, because the traffic generators increased the web server around 20-40%. Hence, in order to trigger the anomaly detection mechanism, we had to keep the criteria at 20% and low CPU usage to be 5%.

Algorithm 2: Algorithm integration into Elascale

```
1  for each micro/macroservice do
2      if cpu_usage within acceptable threshold then
3          Do nothing
4      else if $cpu >= 20\%$ then
5          if cpu or net AR models detect anomaly then
6              Don't upscale microservice
7          else
8              Allow upscaling the microservice
9      else
10         if cpu or net AR models detect anomaly then
11             Don't downscale microservice
12         else
13             Allow downscaling the microservice
```

V. Experimental Evaluation

In this section, we simulate two attack scenarios and evaluate the anomaly detection in these use-cases.

A. Simple DoS Attack

In this scenario, we simulate a simple DoS attack by using Apache HTTP server benchmarking tool ***ab***. We perform 100000 requests with 10 concurrency requests simultaneously.

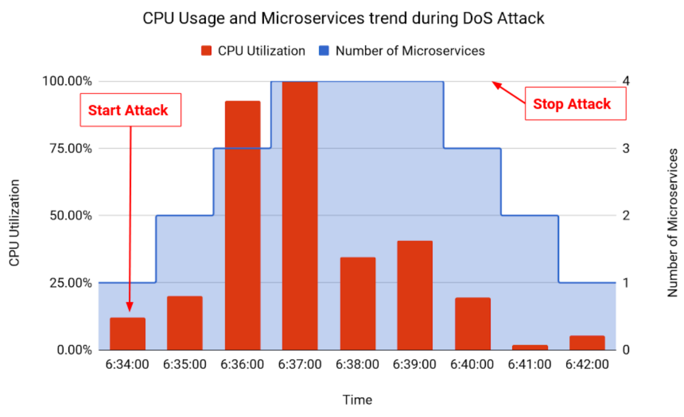

Fig. 4. CPU Usage and number of microservices for Web Server during simple DoS Attack when Anomaly Detection is not enabled on Elascale

As shown in Figure 4, when anomaly detection is not enabled on Elascale, more microservices are deployed to handle the workload as the CPU Utilization increases. Note that we set a maximum threshold of four instances for each microservice, in order to prevent massive scaling of resources. Therefore, this attack resulted in wasting three microservices. Once we stop the attack, the number of microservices decreases.

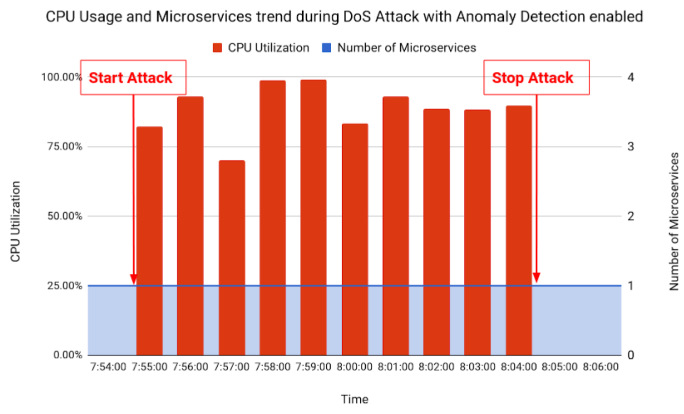

Fig. 5. CPU Usage and number of microservices for Web Server during simple DoS Attack when Anomaly Detection is enabled on Elascale

Figure 5 shows the states of the system when anomaly detection is enabled on Elascale. As the CPU utilization increases, our AR models were able to detect it as anomaly and Elascale did not autoscale the microservice. Hence, the number of microservices stayed flat at one, while the CPU Utilization stayed at 100%.

As seen on the plot, the microservice suffers from the attack regardless. This is because our algorithm does not focus on mitigation mechanisms and thus it is out of scope for this paper. However, one option is to send an alert to the application owner when an anomaly is detected. This will allow the user to decide whether to keep the application running and incur the economic cost or close the application.

B. Microservice Stress Attack

In this scenario, we emulate a malicious user who successfully enters the environment and stresses the microservice to generate false notion that the web server needs more resources. We can simulate this attack by executing the ***stress*** Linux command on the web server container.

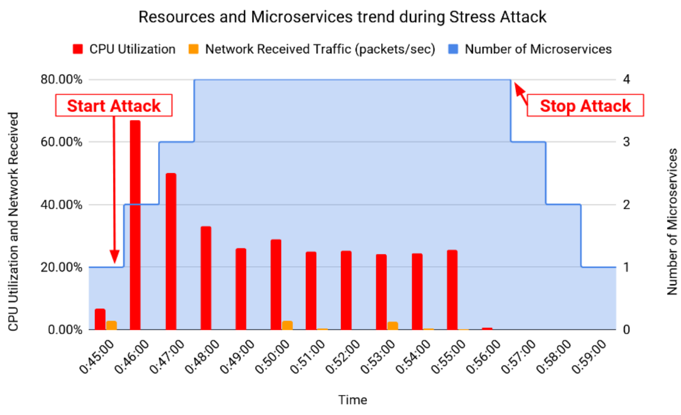

Fig. 6. CPU Usage, Network Received Packets/sec (in %) and number of microservices for Web Server during stress attack when Anomaly Detection is not enabled on Elascale

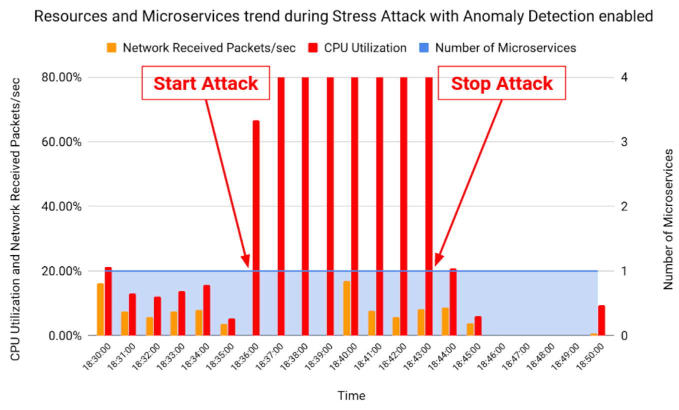

Fig. 7. CPU Usage, Network Received Packets/sec (in %) and number of microservices for Web Server during stress attack when Anomaly Detection is enabled on Elascale

Figure 6 shows a plot of the trends of CPU Utilization, Network Received packets/sec (in percentage) and microservices when anomaly detection is not enabled on Elascale. As seen on the plot, there is a high amount of CPU usage whereas there is a minimal usage of network activity. As the CPU usage increases, Elascale tries to reduce the workload by scaling more services. As more services get deployed, the CPU usage decreases. However, Elascale does not recognize that the workload is bogus and ends up wasting three microservices.

Figure 7 shows the plot of the resources and number of microservices trend when anomaly detection was enabled on Elascale. Since the attacker is not generating any network traffic, the AR model for the network property will detect this unusual low packets/sec property. Hence, this anomaly was detected by our algorithms and we can clearly see that the number of microservices stays flat at one, while the CPU usage stays at around or above 100%.

VI. Conclusion and Future Work

In this paper, we presented an anomaly detection mechanism by performing resource behavioural analysis on microservices using AR statistical models on autoscaling systems. This helps reduce eDoS attacks for application owners and prevents resources wastage on the infrastructure. The models work very well in detecting simple DoS and Microservice Stress attacks on our environment where we perceive cyclical trends of microservices usage of resources. This allows our time series forecasting algorithms to train and forecast with very low MSE. In our future work, we look into possible methodologies to detect anomalies that do not contain cyclical patterns. Furthermore, we will look into possible use-cases of validating our algorithms with real-world public datasets.

References

[1] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: autoscaling and monitoring as a service," in *CASCON*, 2017.
[2] Elascale_secure. [Online]. Available: https://github.com/RajsimmanRavi/Elascale_secure
[3] "Aws best practices for ddos resiliency," White Paper, Amazon, June 2016.
[4] A. Bremler-Barr, E. Brosh, and M. Sides, "Ddos attack on cloud auto-scaling mechanisms," *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pp. 1–9, 2017.
[5] X. Chen, P. Mohapatra, and H. Chen, "An admission control scheme for predictable server response time for web accesses," in *Proceedings of the 10th World Wide Web Conference for Web accesses, Hong Kong*, May 2001.
[6] T. Lin, B. Park, H. Bannazadeh, and A. Leon-Garcia, *SAVI Testbed Architecture and Federation*. Cham: Springer International Publishing, 2015, pp. 3–10. [Online]. Available: https://doi.org/10.1007/978-3-319-27072-2_1
[7] N. Kumar, P. Sujatha, V. Kalva, R. Nagori, A. Katukojwala, and M. Kumar, "Mitigating economic denial of sustainability (edos) in cloud computing using in-cloud scrubber service," in *Fourth International Conference on Computational Intelligence and Communication Networks*, 2012, pp. 535–539.
[8] Z. A. Baig, S. M. Sait, and F. Binbeshr, "Controlled access to cloud resources for mitigating economic denial of sustainability (edos) attacks," *Computer Networks*, vol. 97, no. Supplement C, pp. 31 – 47, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128616000050
[9] J. Jiang, J. Lu, G. Zhang, and G. Long, "Optimal cloud resource auto-scaling for web applications," pp. 58–65, 05 2013.
[10] cyweb_hammer. [Online]. Available: https://github.com/cyweb/hammer
[11] "An architectural blueprint for autonomic computing." White Paper, IBM, June 2005.
[12] A. Sun, T. Ji, and J. Wang, "Cloud Platform Scheduling Strategy Based on Virtual Machine Resource Behaviour Analysis," *Int. J. High Perform. Comput. Netw.*, vol. 9, no. 1/2, pp. 61–69, Feb. 2016. [Online]. Available: http://dx.doi.org/10.1504/IJHPCN.2016.074659