Self-managed Elastic Scale Hybrid Server Using Budget Input and User Feedback

Jose T. Painumkal Rui Wu Sergiu M. Dascalu Frederick C. Harris, Jr

Department of Computer Science and Engineering
University of Nevada, Reno
Reno, NV, USA

josepainumkal@nevada.unr.edu {rui, dascalus, fred.harris}@cse.unr.edu

***Abstract*—To simplify implementation and management of web-based applications, project managers usually choose to rent powerful servers from third party companies, such as Amazon (Amazon Web Services) and Microsoft (Azure). However, in this case, it is rather hard to follow an efficiency plan based on the available budget and the users' feedback. In particular, it could be costly when the manager forgets to shut down the servers when not using them, as the related price may become ridiculously high. Even though some third party companies offer budget notification services, such as Amazon Budgets, it is still difficult to control the usage of the rented servers. Furthermore, the managers need to change their budget based on the feedback received from their own users. For example, if the users complain the waiting time is too long, the managers need to increase the allocated server budget. In this paper, we propose a modified elastic scale hybrid server method based on the available budget and the users' feedback. The servers considered in the paper are called hybrid because they include both owned and rented servers. The proposed method uses queueing theory and user feedback to dynamically adjust the number of servers. The manager can obtain the approximate queue length and the user waiting time by inputting the budget. Using the proposed method, a prototype system was built to run environmental models. The results show that the proposed approach is better, and has lower job waiting time and queue length compared to the traditional FIFO approach. Also, the prototype can shut down the rented servers automatically when they are not being used and can predict both the queue length and the job waiting time.**

Keywords – elastic scale servers; hybrid servers; budget-based planning; environmental models; modified queueing theory

I. INTRODUCTION

The server-client architecture is widely used in the web-based applications. When a Model-View-Controller (MVC) design is applied to a web-based application, it can make the client side very light [1]. For example, the Wildfire Simulation System presented in [2] is a web-based application that uses this design. This means that the user is able to run a complex program with only some necessary software installed and some data stored on the client side. However, this method is not perfect. It requires powerful servers to deal with complex tasks and to communicate with the client side (usually through RESTful APIs). If the servers are not powerful enough, the requests (such as searching and item from a database storing huge data) may not be finished within the desired response time and the user may notice the delay.

To build a powerful server cluster, the project managers can purchase machines and set up servers locally or they can rent servers from the third party companies. The prevalent choices are AWS (Amazon Web Service) from Amazon and Azure from Microsoft. In our opinion, a practical method is to have a core cluster of owned servers and rented servers from the third party companies based on the actual needs. However, when this idea is applied in the real world, a question cannot be avoided: what is the trade-off between the rental budget and the server performance? To our knowledge, there is not yet available a method that deals with this problem efficiently. Some companies provide budget notifications and offer surveys to help project managers understand various operational situations. However, these functions cannot predict the user waiting time and the queue length.

In this paper, we propose a self-managed method for an elastic scale hybrid server model. It can be applied to server clusters containing both owned servers and rented servers. The method uses modified queueing theory to predict the job waiting time and the queue length based on the budget. The system can evolve based on the average job time consumption and notify the project manager to adjust the rental budget based on the user feedback. More details are presented later, in the Proposed Method and Prototype System sections.

The rest of the paper is organized as follows: Section II introduces the research background; Section III presents the proposed method aimed at estimating the queueing length and the user waiting time; Section IV describes a prototype system we built and presents the techniques used in the system; Section V provides a comparison of the results obtained between the proposed method and the traditional FIFO method; and Section VI contains the paper's conclusions and outlines planned future work.

II. BACKGROUND AND RELATED WORK

The concept of "elastic server" has been around for a while. Some third party companies such as Amazon provide this service. Amazon EC2 (Elastic Compute Cloud) is one of the most commonly used elastic services, updated in December 2016 to support auto-scaling [3]. Previously, the rented cluster was removed on termination. This means that when one

removes each machine in the cluster, everything will be removed and the project manager will need to manually scale up and scale down the number of servers. Now, Amazon EC2 Auto Scaling enables the servers to scale up and scale down automatically. There are two main methods to achieve this: (i) Based on events – for example, if the CPU utilization passes a certain threshold, Amazon will spin up EC2 instances to lower the CPU utilization and when the CPU utilization comes down, the EC2 instances will be shut down; and (ii) Based on schedule – for example, when normally most users use a website during the day and consequently the website manager sets a rule based on time, such as that from 7:00 am to 7:00 pm more servers should be rented. The Amazon EC2 Auto Scaling is also very easy to use. The project manager needs to group instances into auto scale groups and set operation rules.

However, this service is not perfect, especially when auto-scaling servers based only on events or schedules. The main issues are: (i) When the servers are auto-scaled, it is also hard to control the budget. The service may scale up the servers and go over the budget; and (ii) the auto-scale service can shut down some servers when the CPU utilization or network utilization are below a threshold. However, these events cannot truly represent that these servers are not needed. It is possible that they are just busy with some low-CPU or low-network jobs. Thus, the best way should be for the servers to control themselves because no one knows better than the servers about what is going on inside of the machines. Therefore, the method proposed in this paper shuts down the servers based upon an evaluation performed when each job is completed.

There are many studies conducted in the field of dynamic provisioning of computing resources in a cloud environment. Calheiros *et al.* [4] proposed an adaptive provisioning technique based on analytical performance and workload information to offer end users the guaranteed Quality of Services (QoS). The QoS targets were application specific and were based on requests service time, rejection rate of requests and utilization of available resources. The proposed model uses the observed system performance and predicted load information to estimate the number of virtual machine instances to be allocated for each application. Zhu *et al.* [5] proposed a feedback control based dynamic resource provisioning algorithm for allocating computing resources with budget constraints. The proposed model was intended to maximize the application QoS requirements by meeting both time and budget constraints. This was facilitated through the dynamic provisioning of CPU cycles and memory to multiple virtual machines in the cluster. However the proposed model requires the reconfiguration of computing resources in the available machine instances rather than the addition/removal of virtual machines from cloud provider. Bi *et al.* [6] proposed a dynamic provisioning technique to optimize the resource provisioning in cluster-based virtualized multitier applications using a hybrid queueing model. The goal of the research was to predict the number of VMs required for a virtualized multitier application such that the all the incoming requests can be serviced with a given response time.

In this paper, we propose a self-managed server system to facilitate elastic scaling in a hybrid server environment, based on the budget amount and user feedbacks, using an improved queuing model. In the proposed approach, waiting time and queue length of the job requests were estimated based on the budget input. The feedback from users are continuously monitored and offers a facility to improve the QoS target. Since cloud providers follow usage based payment structure, our approach is more useful as it relates budget amount directly with the desired QoS targets (here, waiting time for the jobs) and thereby helps managers in making budget decisions more easily.

III. PROPOSED METHOD

This section introduces a method to estimate the job queue length and the job waiting time. For simplified presentation purposes, an hour is used as the time unit. In the practical use, it can be any other time units.

A. Original Queuing Model

One of the simplest queuing models is the classical first-in first-out (FIFO) approach [7]. The idea is that the first job that comes into the queue will leave first, and so on. In this paper, we have compared our proposed method with this queuing model, and the results are shown in Section V.

An M/M/1/1/∞/∞ queuing model is another simple and practical queuing model [8]. This queuing model represents the following situation with a Poisson distribution [9] where jobs arrive at a rate of λ/hour and the server processes the jobs at a rate of μ/hour (typically, this is considered exponential). There is one server; the queue length can be infinite, and the population (maximum number of the jobs at the same time) can be infinite. Based on [8], when λ (job arrivals rate) is less than μ (server processes rate), the expected queue length is:

$$L = \frac{\lambda^2}{\mu^2 - \lambda * \mu} \quad (1)$$

and the average job waiting time in the queue is:

$$T = \frac{\lambda}{\mu^2 - \lambda * \mu} \quad (2)$$

B. Applied Modified Queuing Model

The M/M/1/1/∞/∞ queuing model was modified and applied in our prototype system. The idea is to develop a formula to estimate the average job waiting time and queue length in a hybrid server environment using the budget amount, budget period, cost of rented instances, and the average time for job execution. The modified queuing model processes jobs with owned servers and rented servers. For the given budget, B and average job execution time, T_{own}, the owned servers can process a maximum of $(N_0 * T_b)/T_{own}$ jobs during the budget period T_b, where N_o denotes the number of owned servers.

If a rented instance costs \$P for an hour of usage, then $B/(P*T_{rent})$ is the total number of jobs that can be processed with rented servers for the given budget amount B. To achieve a stable service, the usage of rented servers are distributed uniformly during the budget time period T_b, which means the project manager should rent a server at every time

interval, $T_{int} = (T_b * T_{rent} * P)/B$, if the owned servers are busy. At every T_{int} interval, if the owned servers are available (which means the job queue is empty), then the system needs not spin up a rental server for the incoming job. The system will also increment a counter variable, so that later, if a job comes in and the owned servers are busy, the system will rent a server immediately. This way, the proposed approach ensures that rented workers were utilized judiciously throughout the entire budget period. During the budget period, the hybrid server system could process a maximum of $(N_0 * T_b)/T_{own} + B/(P * T_{rent})$ jobs.

Therefore, based on formula (1), the expected queue length is:

$$L = \frac{\lambda^2}{\left(\frac{N_0}{T_{own}} + \frac{B}{P * T_{rent}}\right)^2 - \lambda * \left(\frac{N_0}{T_{own}} + \frac{B}{P * T_{rent}}\right)} \quad (3)$$

and based on formula (2), the average job waiting time in the queue is:

$$T = \frac{\lambda}{\left(\frac{N_0}{T_{own}} + \frac{B}{P * T_{rent}}\right)^2 - \lambda * \left(\frac{N_0}{T_{own}} + \frac{B}{P * T_{rent}}\right)} \quad (4)$$

As an example assume that the project manager has five owned servers and each job takes an average of 10 minutes to finish the job on both owned and rented servers. Thus, the owned servers can finish 30 jobs in an hour. If the manager has $100 for a budget period of one hour and a rented instance costs $1 to rent a machine per hour, then the $100 can be used to rent 600 jobs during the one hour budget period. (The rented instance is stopped on the completion of the assigned job). A uniform distribution of server rentals means the project manager has to rent a server every 0.1 minute, if there is at least one job in the job queue. If no jobs are in the queue, then we record the occasion to a counter variable and wait for the next job. Thus in this scenario, the hybrid server system can process a total of 630 jobs (i.e. $\mu = 630$) during the budget period of one hour. If 500 jobs arrive per hour (i.e., $\lambda=500$) then, based on (3) and (4), the expected queue length is 3.0525 and the average job waiting time is 0.0061.

C. User Feedback

User feedback is one of the most important inputs for the project manager. To collect this information, the system requests the user to complete a survey about the system performance, such as the waiting time and the response speed. The users may not want to fill the survey if they are satisfied. However, if they are not satisfied (e.g., waiting for a long time), there is a high possibility that they will complain through the survey. The project manager should give weights to each question and options. For example, "Do you think you are waiting too long for the service?" has 0.8 weight factor and this question has three choices: Yes (worth +1 point), Not sure (worth 0 point), and No (worth -1 point). If the user chooses "Yes", the feedback collector will add 0.8*1=0.8 point to the global feedback value. The weights are decided by experience in our current prototype system and we are working to classify user feedback into different weight categories with machine learning techniques. If the feedback value passes a certain threshold, the system will send a notification email to the project manager to raise the budget.

IV. PROTOTYPE SYSTEM

In this section, we give an overview of the developed system and describe how the proposed elastic-scale approach was implemented. Part of the larger NSF EPSCoR-funded Virtual Watershed project [10] and [11], the objective of the developed system has been to provide a computing platform for hydrologists to run different environmental models and acquire results of various *model runs*. The Precipitation Runoff Modeling System (PRMS) [12] is one of the environmental models supported in the Virtual Watershed platform and represents the workload for this study.

The platform is structured with a micro-service architecture, in which each service is highly independent in nature and is designed to perform a particular task. We used the latest container based virtualization technology, Docker [13], to implement various services required for the system. Using Docker, a service can be packaged into a self-contained lightweight software container. The container provides an isolated running environment which will have all the required libraries and dependencies to run the service. In our system, we packaged the services such as user authentication, data storage, model data processing, etc. into separate Docker containers. The containers communicate between each other through a restful API service. All the containers reside on a single host machine.

To run a model, the user has to use the API service to upload all the input files required by the specific environmental model to our system. The files will be stored in the storage location (Mongo DB) and a unique model id will be generated. The model id will be then placed onto an asynchronous job queue. The worker container which is configured to listen to the job queue will grab the model id from the queue, obtain the input files for the model run from the storage location, and start processing the model run. Once the model processing is finished, the generated output files will be stored into the storage location so that user can later download the files and evaluate the results. The length of the job queue is infinite and there can be any number of jobs in the queue. There are limitations on the number of jobs concurrently processed by the worker container. When more jobs are waiting in the queue, the users have to wait more time to start the processing of their job. This would cause frustration among the users and distract them from using the platform. This problem could be solved by adding more worker containers to the platform. However, the addition of more workers to the same host machine could cause serious performance issues. The Docker container consumes computing resources on the host machine, depending on what kind of job is being processed inside the container. Due to this dependency on the resources of the host machine, the addition of more worker containers on the same host machine is not feasible. Therefore, a solution to this problem is to incorporate more host machines into the current platform and distribute the worker containers among different machines. This way, all the workers would grab the job from the job queue, process

the jobs concurrently, and store the results at a common storage location for further reference.

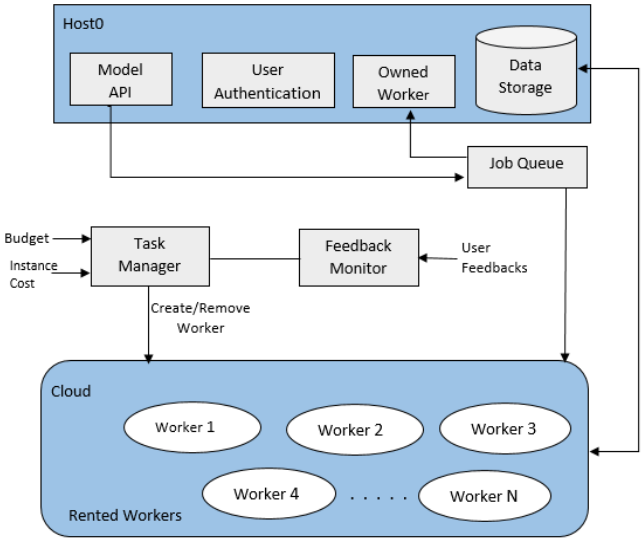

Figure 1. Proposed hybrid server system

To implement the proposed approach, a multi-host swarm cluster was created in which one machine will act as the swarm master. Any number of hosts can be added to the swarm cluster. The architecture of the system is shown in Figure 1. Host0 acts as the swarm master and it contains the owned worker container. The other Docker containers handling different services were also deployed on the Host0 machine. The rented workers are distributed across several remote machines. To keep the prototype system simple, only one worker container is allocated per machine and the number of jobs to be processed by the worker at a time is limited to one. An overlay network is used to connect the Docker containers across multiple hosts and hence the containers can communicate with each other. The worker containers are configured to listen to the job queue so that as long as the container is in an active state, it starts picking jobs from the job queue and initiates the processing of the jobs. The popular distributed task queue, Celery, was used to implement the job queue in the prototype system, which handles the execution of the jobs asynchronously. The task manager module is placed inside host0 and it can start, stop, and delete worker nodes. To facilitate the creation and deletion of Docker containers at will, the python library for Docker engine API, docker-py was used.

The proposed self-managed hybrid worker system autonomously decides when to use rented or owned workers taking into account the impact the action can bring on the waiting time of the jobs in the queue. The prototype system can delete the containers and shut down the servers when they finish their jobs. This is done by sending a signal from the working rented server after it finishes the execution of its job. In this way, the system ensures maximum productivity and avoids unnecessary expenses due to resource wastage. However, the prototype is not fully self-managed as it requires the manager to change the budget based on the users' feedback. In the current prototype system, the remote instances are represented using three physical machines. In a non-prototype system, the physical machines could be replaced with instances from cloud providers.

Algorithm 1: Create Rent Worker

```
1  function rented_worker_creation (RW, UR, N, T_int);
   Input  : RW denotes number of rented workers; UR denotes unused
            rentals; N denotes maximum number of models processed with
            rented workers for the input budget; T_int denotes the time
            interval
2  if RW < N then
3      if Jobs in queue then
4          Create Worker;
5          RW = RW + 1;
6          while UR > 0 AND RW < N AND Jobs in queue do
7              Create Worker;
8              RW = RW + 1;
9              UR = UR - 1;
10         end
11         Sleep T_int and go to line 2
12     else
13         UR = UR + 1;
14     end
15 else
16     No more rented workers available;
17 end
```

Figure 2. Algorithm to create rented worker

Figure 2 shows the logic for creating new rented workers in the proposed system. The project manager can modify the budget in the middle of an execution and the system updates N accordingly with the changes in the budget amount. Every T_{int} interval, the system would check the job queue to determine whether there is any necessity for rented workers. If the queue is not empty, then a new worker container is created and added to the system. If the queue is empty during the evaluation at T_{int} interval, the system records such occasions to a counter, and later compensates for those unused occasions by creating more rented workers during busy times up to the counter. This process will be repeated until the number of jobs rented equals N, the maximum number of jobs that could be processed with rented containers for the given budget. The algorithm guarantees the cost will be within the budget because: 1) the algorithm only rents a worker when the job queue is not empty, which also means all owned workers are busy; 2) when there are no unused rentals, the algorithm checks the queue every time interval.

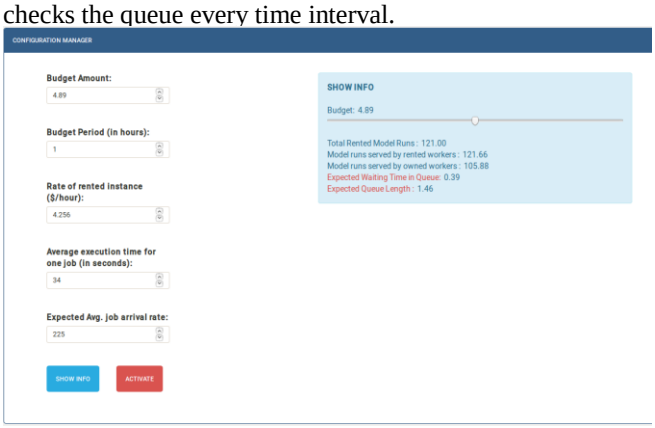

Figure 3. Screenshot of the configuration manager module

The proposed system includes a configuration manager module, where the manager could enter details such as the budget amount, price of machine instances, expected job arrival rate, and budget period. The system calculates the total jobs that can be rented and also provide estimations on the expected waiting time for the jobs with the given budget amount. The module also includes a slider tool, which helps the manager to easily figure out how much money to spend to get the desired waiting time for the jobs. Figure 3 shows a screenshot of the configuration manager module in the prototype system and this part uses Equation (3) and Equation (4) presented in Section III.

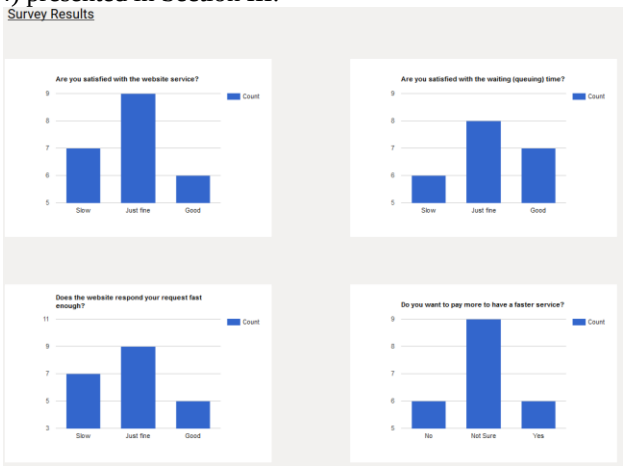

Figure 4. Screenshot of the user feedback score

The proposed system also includes a survey form where users can provide their feedback on the performance of the new approach. The survey is mainly intended to help managers to make policy decisions on the budget amount to be allocated. The survey results provide clear indications on whether the allocated budget is sufficient enough to provide a good user experience with the platform. The survey mainly collects feedback on the waiting time, responsiveness, and overall performance of the system. Figure 4 shows a screenshot of the feedback score visualization. If the score passes a preset threshold, the system will send an alert email to the project manager.

V. RESULTS

The proposed approach was evaluated by simulating a Poisson job arrival stream on the job queue. Each job constitutes one PRMS model run with real climate data. We have conducted experiments with different models and input data files, but because of the space limitation only one of them is shown. To execute one job, the worker takes an average of 34 seconds (this simulates one-month of climate modeling). The initial execution time is obtained from experience and it is replaced with the average job execution time after the server starts working. Since the experiment was conducted with physical machines instead of machine instances from cloud providers, the cost of the host machine and the budget amount were simulated. For the experimental study, the system was allocated with a budget amount of $1.63 for a budget period of 20 minutes and the cost of the rented instance was considered to be $4.256/hour which is the current cost for a high end compute node on AWS. With the provided budget and price of instances, a maximum of 40 models could be processed with rented workers and the time interval T_{int} was estimated to be 30 seconds. i.e. during the budget period of 20 minutes, the system would use a rented worker to execute the job every 30 seconds, provided the owned worker is busy at that time.

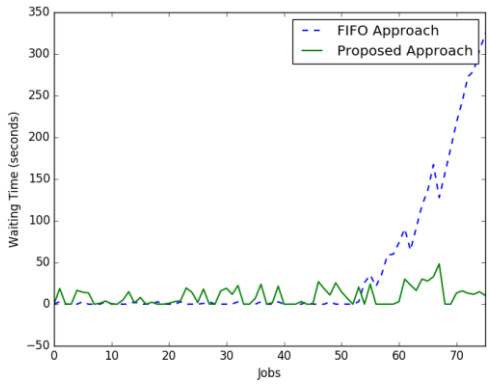

Figure 5. Comparison of waiting time of jobs

Figure 5 shows the comparison of the waiting time between the FIFO approach and the proposed elastic server approach. In the proposed hybrid elastic-server approach a rented container will be used only at regular time intervals, whereas in the FIFO approach a new rented worker container will be created and used to execute the job whenever the owned worker is busy. The drawback of the FIFO approach was that the rented workers may not last until the end of the budget period. Therefore, once the rented models are over, the incoming jobs have to wait more time in the queue causing a drastic increase in the waiting time. Whereas in the proposed approach, the rented jobs were used judiciously and hence the waiting time of the jobs was maintained at a controlled level throughout the budget period.

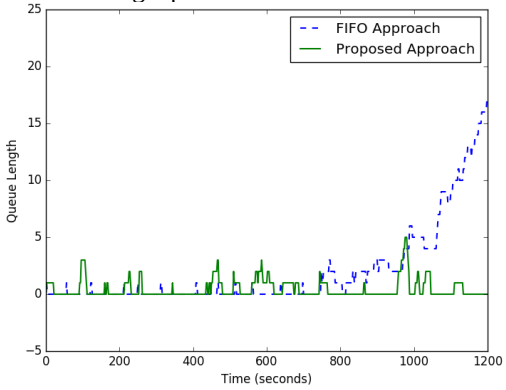

Figure 6. Comparison of number of jobs waiting in the queue

Figure 6 shows the comparison of the queue length between the FIFO approach and the proposed approach. In the FIFO approach, all the rented workers were finished around the 13th minute and it resulted in a steep increase of the queue length. In contrast, in the proposed approach the queue length was consistently maintained small throughout the budget period.

The waiting time was calculated as the time taken by the worker to start the job once the job is added to the queue. As seen in Figure 5, the waiting time of the jobs showed several fluctuations during the monitoring period, some of which impacting both the FIFO and the proposed method. In the experiment, the rented worker containers were created from scratch using the base image. Depending on the resource utilization on the host machine, the container creation consumed several milliseconds to seconds. After the creation of a worker container, the worker took few more seconds to establish a connection with the configured job queue and pick a job from it. We can also see fluctuations in the proposed approach since the rented workers were created at regular time intervals T_{int} and this would also be a reason for observing variations in the waiting time.

In the experiment, the FIFO approach ran out of the rented workers in 747.1 seconds. During this period, it finished 54 jobs, 14 of them being completed by the owned workers. Therefore, the utilization rate of the owned workers was 25.93%. In the same time period (747.1 seconds), the proposed system finished 40 jobs, 21 of them being completed by the owned workers. Thus, the utilization rate of the owned workers was 52.5%. It is evident from the results that the proposed method had a higher utilization rate of the owned workers and it saved more rented workers for later use.

Based on Equation (3) and Equation (4), the expected queue length (number of job arrivals in the queue) was 1.46 and the real queue length was 0.622. Theoretically, each job needed to wait 0.39 minute and in fact each job waited 0.34 minute on average. This shows that the proposed method worked well in this job queue case. The experimental study was conducted with four machines with Intel i7 CPU, 16 GB DDR4 RAM, and 256 GB SSD. Multiple threads were used to handle the continuous monitoring of queue length and job status, the creation of rented workers, and the simulation of the Poisson job arrival stream. The time slicing between the different threads could also be a reason for fluctuations in the queue length and observed waiting time. The time consumption for starting and stopping a rented instance varies with the work load and the cloud hosting service. Normally, the starting time of an instance ranges between 30 seconds to 6 minutes. Since our goal was to prove the applicability of the proposed approach, the experiment was conducted with comparatively shorter jobs and hence T_{int} value is also relatively small (less than a minute). However in real world scenarios, while dealing with high time consuming jobs, the estimated T_{int} value would be sufficiently large enough to accommodate the varying VM start time.

VI. Conclusions and Future work

In this paper, we have proposed a method to estimate queue lengths and waiting time based on a modified $M/M/1/1/\infty/\infty$ queue model. The method was used for elastic scale hybrid servers, which are a combination of owned and rented servers. The experimental results showed that our proposed approach performed better than the traditional FIFO queue model in what regards the average waiting time and the expected queue length.

In its current version, the system can shut down the rented servers when they finish their jobs to save money. However, the project manager still needs to change the budget based on the users' feedback. In the future, we plan to use machine learning techniques to improve this part. Also, we will work on enhancing our method to support multiple job queues and multiple hybrid servers.

Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) under grant numbers IIA-1329469 and IIA-1301726, as well as by a University of Nevada, Reno Graduate Student Association research grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Leff, A. and Rayfield, J.T., 2001. Web-application development using the model/view/controller design pattern. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing (EDOC'01)*, pp. 118-127.

[2] Wu, R., Chen, C., Ahmad, S., Volk, J.M., Luca, C., Harris, F.C. and Dascalu, S.M., 2016. A Real-time Web-based Wildfire Simulation System. In *Proceedings of the 42nd Annual Conference of the IEEE (IECON-2016)*, IEEE Industrial Electronics Society, pp. 4964-4969.

[3] Amazon, "AWS | Auto Scaling", 2017. [Online]. Available: https://aws.amazon.com/autoscaling/. [Accessed: 11-Jan-2017]

[4] R. Calheiros, R. Ranjan and R. Buyya, "Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments", *2011 International Conference on Parallel Processing*, 2011.

[5] Q. Zhu and G. Agrawal, "Resource provisioning with budget constraints for adaptive applications in cloud environments", *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, 2010.

[6] Bi, J., Zhu, Z., Tian, R. and Wang, Q., 2010, July. Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center. In *Cloud Computing (CLOUD), 2010 IEEE 3rd international conference on* (pp. 370-377). IEEE.

[7] Kruse, R. and Tondo, C.L., 2007. *Data structures and program design*, C. Pearson Education India, p. 78.

[8] Karlin, S. and McGregor, J., 1958. Many server queueing processes with Poisson input and exponential service times. Pacific J. Math, 8(1), pp. 87-118.

[9] Boddy, R. and Smith, G., The Poisson Distribution. Statistical Methods in Practice for Scientists and Technologists, pp. 111-119.

[10] Dascalu, S., 2014. Scientific Collaboration in Virtual Environments: The Western Consortium Watershed Analysis, Visualization, and Exploration (WC-WAVE) Project. *Proceedings. of the International Conference on Collaborative Technologies and Systems (CTS-2014)*, pp. 560-561.

[11] Carthen, C., Rushton, T.J., Burfield, N., Johnson, C.M., Hesson, A.,Nielson, D., Worrell, B., Delparte, D., Chapman, T., Johansen, W.J., Lew, R., Wood, N.R., Ziegler, M., Anderson, J. W., Dascalu, S.M.. and Harris. F.C.. Jr.. 2016. September. Virtual Watershed Visualization for the WC-WAVE Project. *International Journal of Computers and Their Applications*, 23 (3): 195-2

[12] Leavesley, G.H., Lichty, R.W., Thoutman, B.M. and Saindon, L.G., 1983. *Precipitation-runoff modeling system: User's manual* (p. 207). Washington, DC: USGS.

[13] Docker, "Docker", Docker, 2017. [Online]. Available: https://www.docker.com/. [Accessed: 13- Jan-2017]