J Netw Syst Manage (2016) 24:285–308
DOI 10.1007/s10922-015-9352-x

An Analytical Model for Estimating Cloud Resources of Elastic Services

Khaled Salah[1] · **Khalid Elbadawi**[2] · **Raouf Boutaba**[3,4]

Received: 10 November 2014 / Accepted: 3 August 2015 / Published online: 7 August 2015

Abstract In the cloud, ensuring proper elasticity for hosted applications and services is a challenging problem and far from being solved. To achieve proper elasticity, the *minimal* number of cloud resources that are needed to satisfy a particular service level objective (SLO) requirement has to be determined. In this paper, we present an analytical model based on Markov chains to predict the number of cloud instances or virtual machines (VMs) needed to satisfy a given SLO performance requirement such as response time, throughput, or request loss probability. For the estimation of these SLO performance metrics, our analytical model takes the offered workload, the number of VM instances as an input, and the capacity of each VM instance. The correctness of the model has been verified using discrete-event simulation. Our model has also been validated using experimental measurements conducted on the Amazon Web Services cloud platform.

Keywords Cloud computing · Capacity engineering · Resource management · Auto-scaling · Elasticity · Performance modeling and analysis

✉ Khaled Salah
khaled.salah@kustar.ac.ae

Khalid Elbadawi
badawi@cdm.depaul.edu

Raouf Boutaba
rboutaba@cs.uwaterloo.ca

[1] Electrical and Computer Engineering Department, Khalifa University of Science, Technology and Research (KUSTAR), Abu Dhabi, UAE

[2] School of Computing, DePaul University, Chicago, IL, USA

[3] David R. Cheriton School of CS, University of Waterloo, Waterloo, Canada

[4] Division of IT Convergence Engineering, POSTECH, Pohang, Korea

1 Introduction

Elasticity is a key characteristic of the cloud computing paradigm. Elasticity is typically defined as the ability to dynamically scale (or auto-scale) cloud IT resources as required and in accordance to the presented workload demand. We define “Elastic Services” as those cloud-hosted services or applications with elasticity property. For these elastic services, the cloud compute resources allocated to them need to be dynamically scaled so that specific service level objective (SLO) performance requirements can be satisfied. The SLO performance requirements can include response time, throughput and service request probability loss. Examples of elastic services are web services, email services, query and search engines, database services, financial services, multimedia systems, and many others.

In a typical cloud datacenter, as depicted in Fig. 1, elasticity for cloud-hosted applications or services is realized using a Load Balancer (LB) node. Other than load-balancing functionality, the LB node is also responsible for managing and automating the provisioning and de-provisioning of the underlying cloud resources. The cloud resources are typically compute (or worker) VMs that may require access to storage or database servers. The LB, which fronts the worker VMs, is responsible for communicating continuously with the provisioned worker VMs to determine the availability of each VM to accept new requests. The LB attempts to dispatch requests equally among all VMs [1, 2]. In addition, the LB monitors the presented workload and the utilization state of each worker VM, and accordingly, it decides to provision (scale up) or de-provision (scale down) VM instances in order to achieve *proper elasticity* by which the minimal worker VMS are allocated to satisfy the SLO requirements. It is worth mentioning that there are different approaches to conduct cloud monitoring [3]. A complete and comprehensive survey on existing cloud monitoring mechanisms, approaches, and the available commercial and open-source platforms and services can be found in [3]. Also the authors in [3] have surveyed

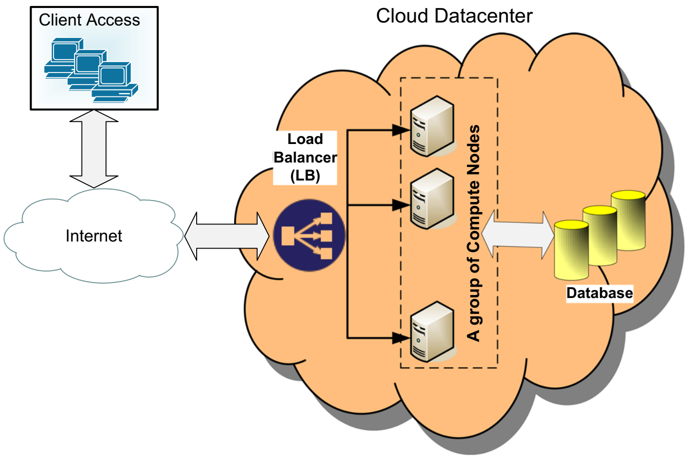

Fig. 1 A typical cloud deployment architecture of an elastic service

many types of monitoring implementations for VM's CPU and physical memory, network workloads, and cloud-based middleware and apps.

The responsibility of implementing elasticity depends on the cloud service model (IaaS or PaaS) offered by the cloud provider. For an Infrastructure-as-a-Service (IaaS) cloud model such as Amazon Web Services (AWS) [4] and Google Compute Engine (GCE) [5], the implementation of elasticity is the responsibility of the cloud service deployer. However, the implementation is transparent to the cloud service deployer when hosting the service on a Platform-as-a-Service (PaaS) or Software-as-a-Service (SaaS) cloud as Google App Engine (GAE) [6]. In the latter case, the elasticity mechanism and implementation are the responsibility of the PaaS cloud provider. For both hosting service platforms, an effective elasticity mechanism has to *accurately* predict the *minimum* required cloud resources to satisfy a given SLO requirement based on the current workload and the capacity of each cloud resource. Allocating more cloud resources than required to satisfy the SLO will result in over-provisioning and higher costs to the service owner. On the other hand, allocating fewer resources than required will lead to under-provisioning whereby the SLO requirements are violated. Therefore, any elasticity algorithm must be able to properly estimate the needed cloud resources while taking into account the current workload and the SLO requirements.

It is worth noting that over provisioning of cloud resources or following a trial and error approach for determining the needed number of cloud resources would lead to higher costs and a prolonged period of SLO violations until the minimal number of cloud resources are determined [7]. Under a heavy workload that entails more cloud resources to be provisioned, a *group* of VMs may need to be provisioned to satisfy the violated SLO (as in the Amazon AWS Auto Scale [1, 2]). If the size of this group is not accurately and promptly predicted, this may lead to prolonging the period of SLO violations. This is primarily due to the time it takes to provision and instantiate new VM instances. Such provisioning time is quite significant and it may take between 30 and 96 s [8, 9]. Therefore, accurate and prompt prediction of the size and the number of compute instances for elastic services become a critical performance and resource management issue. A review of the techniques for auto-scaling cloud-based applications was conducted in [7]. In general, the surveyed techniques address the scalability implementation for different types of applications and services that can be hosted in a cloud computing environment.

In this paper, we present an analytical model that *promptly* estimates (based on derived mathematical formulas) the minimum required cloud resources to satisfy a given SLO performance criterion for an elastic service. Specifically, for a known or measured processing capacity of VM instances and a measured incoming workload (in terms of the arrival request rate), and by using derived formulas, the model can immediately estimate the mean SLO key performance criteria (such as the response time) for a given number of VM instances. The service response time is a common SLO performance criterion [10, 11]. We refer to the service response time as the sojourn time which includes processing/execution time and queueing or waiting time at the datacenter. This sojourn time is part of the end-to-end response time that can include other link and node network delays [12, 13].

Prior related work reported in [12–21] employs general queueing models (as those of *M/M/1*, *M/G/1*, *M/M/m*, *M/G/m/K*, or Erlang formulas) to capture and analyze the behavior of elastic services. None of these models have focused on determining the number of VM instances required to meet SLO performance criteria for cloud-based services. Also, these models fall short of capturing the real behavior of the elastic services. Specifically, all of these models ignore the role of the LB. As stated earlier, the LB plays an important role in dispatching, monitoring, and tracking the availability of compute (or worker) instances at the cloud datacenter. This processing time at the LB can be significant. Hence, any analytical model should account for the role of LB in order to accurately model behavior and performance.

This paper is a major extension of our short 5-page preliminary version that appeared in [22]. In this extended version, we have included a discussion on the applicability and usefulness of our analytical model. We also have included detailed mathematical derivations for our analytical model along with a complete algorithm for deriving additional key performance formulas and measures. In this paper, a new section has been added on modeling and analyzing the scalability of the LB that can be, if ignored, a major performance bottleneck for elastic services. Another section was added on validating our analytical model and formulas. The validation was conducted using measurements of an experimental testbed deployed on the AWS cloud. More importantly, this paper includes numerical results of real world practical scenarios of cloud elastic services that include web service, Netflix video streaming, and the AWS cloud. The section on numerical results includes new figures and considerable discussion and interpretations on cloud resource estimation and capacity engineering aspects related to achieving proper elasticity for cloud services.

1.1 Usefulness and Applicability

Our proposed analytical model is general and can be applied to describe and capture the behavior of other similarly-behaving systems as those of elastic applications or services. *First*, the model can be used in auto-scaling or an elasticity algorithm in which the capacity and number of compute (or worker) instances to be provisioned are determined based on the required SLO requirement and the measured workload. Currently in Amazon’s AWS, auto scaling is based on setting lower and upper thresholds for the overall CPU utilization. We show in Sect. 4 that CPU utilization may not be an adequate measure for auto scaling. In addition, CPU utilization can be misleading as it could be affected by running agents, tasks or processes that are not related to the application workload. Examples of these tasks may include backup, monitoring, instance management, and migration. In [23], it has been shown that CPU utilization is a useless metric especially in a virtualized environment. *Second*, the techniques employed in our model can be used as a basis to model and predict the number of Hadoop cluster nodes (of a certain size and capacity) required to schedule and execute a MapReduce job. *Third*, our model can be used to compute the expected delay for a multi-tier service in which an LB and multiple compute nodes are involved. The aggregated delay at each tier can be used to compute the end-to-end response time. *Fourth,* the model can compute the required cloud instances and estimate the service response time and the required network bandwidth when deploying a Virtual Data Center (VDC) and an Amazon EC2 Fleet in which multiple VM instances (or EC2) are provisioned based on the given SLO requirements. *Fifth*, the model can be used in CCA (Cloud Call Admission) to accept or deny user requests to provision a single or multiple VM instances. As VM instances continuously get provisioned and de-provisioned (due to auto-scaling, migration, and termination), the mean lifetime for an instance can be estimated. Our model can then be used to determine key performance measures for the CCA to grant or deny a request. The measures may include cloud resource utilization, network bandwidth, overall service time, throughput, and blocking probability.

The rest of the paper is organized as follows. Section 2 describes our analytical model to capture the behavior of an elastic service hosted on a cloud. The section derives formulas that can be used in predicting key performance measures that can be used in achieving proper elasticity in terms of minimal cloud resources that satisfy SLO requirements. Section 3 discusses simulation and an experimental testbed used to verify and validate our analytical model. Section 4 presents numerical examples for realistic scenarios in which elastic cloud services can be deployed. It also compares analysis results with simulation and experimental measurements. Finally, Sect. 5 concludes the paper and outlines future work.

2 Markovian Analytical Model

The handling of an incoming request for an elastic service hosted on a cloud is illustrated in Fig. 2. As shown, an arriving request gets first queued in a finite buffer and then dequeued by the LB with a mean service time $1/r$. The LB will dequeue the request for processing if and only if one of the compute/worker instances is readily available to handle a new request. This could happen if the compute instance has

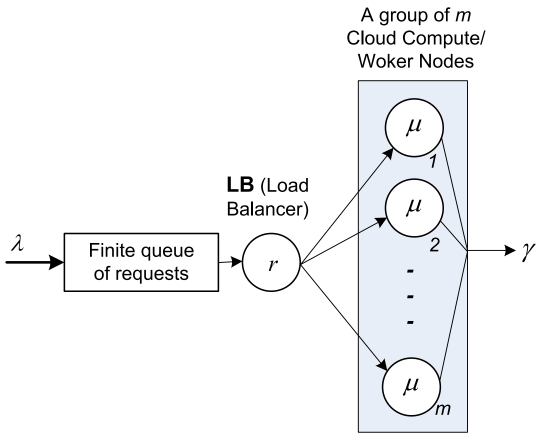

Fig. 2 Finite queueing system model with a LB and *m* worker nodes

been newly launched or it has just finished servicing a request. The compute instance has to notify the LB upon the completion of a request. We assume that each compute instance can service and process dispatched requests from the LB with a mean service time $1/\mu$. We assume that the LB will distribute the load evenly (in a round-robin fashion) among all m provisioned compute instances. In this way, if we assume the incoming mean request rate is λ, the portion of the incoming rate to each individual compute instance will be λ/m. The mean departure rate of all instances is denoted by γ, which also represents the overall throughput of the system. For estimating the average workload λ, the reader can refer to prior work on estimating an instantaneous dynamic workload published in [23–26].

In order to approximately model the behavior and performance of the above system, we assume that incoming requests follow a Poisson arrival λ, and all of the service times are independent and exponentially distributed with means of $1/r$ and $1/\mu$. Requests are serviced according to First Come First Served (FCFS) discipline. In addition, we assume the LB only dispatches the request to the worker node only if the worker node has finished the processing of a previous request. This means the worker node has no queue. Therefore, no queueing delay is incurred at the worker node.

2.1 Limitations

Our analytical model assumes the request arrivals are Poisson, and the service times are all exponentially distributed. It was shown that arrival of HTTP requests for documents under a heavy load closely follow the Poisson process [27]. However, in other cases, the arrival rate of Web or XML requests does not always follow a Poisson process but is bursty [28–30]. Also in reality, service times are not necessarily always exponential. An analytical solution becomes intractable when considering bursty traffic and non-Poisson arrivals, and when also considering general service times. On the other hand, modeling under such assumptions has been extensively used in the literature and can provide adequate approximation of real systems [12–18, 27, 30–33]. Also in Sect. 4.3, we demonstrate that analytical results are in close agreement with experimental results. We would like to state that our model presented in this paper does not capture the behavior of all types of cloud-based applications and services. The analytical model and the derived formulas and algorithms can be applied for cloud-based elastic services that: (1) exhibit the dynamism and behavior exhibited in Fig. 2, and (2) follow the cloud deployment architecture exhibited in Fig. 1. Examples of these elastic services may include web service, email service, query and search engines, multimedia streaming services, and many others. It is also to be noted that our model is specifically designed for two-tier applications and services. However, our model can still be used to approximately study three-tier applications by combining the service of the second and third tiers into one tier of service. For this, we have to assume a linear and homogenous scaling for the second and third tier resources. Such assumptions may not be true for *all* three-tier applications and how they scale. This is a subject that warrants further study and investigation.

Our analytical model uses the embedded Markov chain to represent the behavior of the queueing system, shown in Fig. 2, with a state space $S = \{(k, n), 0 \leq$

$k \leq K,\ n \in \{1, 2\}\}$, where k denotes the number of requests in the system and n denotes the type of processing taking place by either the LB or one of the compute instances. The queueing system has a buffer size of $K - m$. State (0, 0) represents the special case when the system is empty. States (k, 1) represent the states where the request is being handled by the LB. States (k, 2) represent the states where the request is being handled by one of the compute instances. The rate transition diagram is shown in Fig. 3.

Let $q_{k,n}$ be the steady-state probabilities at state (k, n). A system of difference equations can be written as follows. At states (0, 0), (1, 1), and (1, 2), we have

$$\begin{aligned} &-\lambda q_{0,0} + \mu q_{1,2} = 0, \\ &-(\lambda + \mu) q_{1,2} + r q_{1,1} = 0, \end{aligned}$$

and

$$-(\lambda + r) q_{1,1} + \lambda q_{0,0} + 2\mu q_{2,2} = 0,$$

respectively.

Therefore, the probabilities of $q_{1,2}$, $q_{1,1}$, and $q_{2,2}$ can be expressed as follows in terms of $q_{0,0}$

$$\begin{aligned} q_{1,2} &= \frac{\lambda}{\mu} q_{0,0}, \\ q_{1,1} &= \left(\frac{\lambda + \mu}{r}\right)\left(\frac{\lambda}{\mu}\right) q_{0,0}, \quad \text{and} \\ q_{2,2} &= \left(\frac{\lambda + r}{2\mu}\right) q_{1,1} - \left(\frac{\lambda}{2\mu}\right) q_{0,0}. \end{aligned} \tag{1}$$

At each state (k, 1), the difference equation is expressed as

$$\begin{array}{ll} -(\lambda + r) q_{k,1} + \lambda q_{k-1,1} + (k+1)\mu q_{k+1,2} = 0, & 2 \leq k \leq m-1 \\ -(\lambda + r) q_{k,1} + \lambda q_{k-1,1} + m\mu q_{k+1,2} = 0, & k \geq m \end{array} \tag{2}$$

At each state (k, 2), the difference equation is expressed as

$$\begin{array}{ll} -(\lambda + k\mu) q_{k,2} + r q_{k,1} + \lambda q_{k-1,2} = 0, & 2 \leq k \leq m-1 \\ -(\lambda + m\mu) q_{k,2} + r q_{k,1} + \lambda q_{k-1,2} = 0, & k \geq m \end{array} \tag{3}$$

Equation (3) can be rewritten as

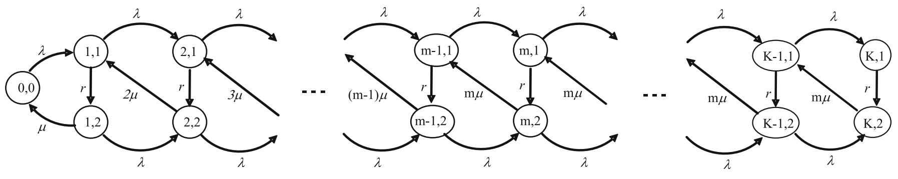

Fig. 3 State transition diagram to capture handling requests for elastic services

$$q_{k,1} = \begin{cases} \left(\dfrac{\lambda + k\mu}{r}\right) q_{k,2} - \left(\dfrac{\lambda}{r}\right) q_{k-1,2} & 2 \le k \le m-1 \\ \left(\dfrac{\lambda + m\mu}{r}\right) q_{k,2} - \left(\dfrac{\lambda}{r}\right) q_{k-1,2} & k \ge m \end{cases} \tag{4}$$

Equation (4) can be rewritten as

$$q_{k,2} = \begin{cases} \left(\dfrac{\lambda + r}{k\mu}\right) q_{k-1,1} - \left(\dfrac{\lambda}{k\mu}\right) q_{k-2,1} & 3 \le k \le m-1 \\ \left(\dfrac{\lambda + r}{m\mu}\right) q_{k-1,1} - \left(\dfrac{\lambda}{m\mu}\right) q_{k-2,1} & k \ge m \end{cases} \tag{5}$$

The boundary probabilities at states (K, 1) and (K, 0) are as follows

$$-rq_{K,1} + \lambda q_{K-1,1} = 0,$$

and

$$-m\mu q_{K,2} + \lambda q_{K-1,2} + rq_{K,1} = 0,$$

respectively.

Therefore,

$$\begin{aligned} q_{K,1} &= \frac{\lambda}{r} q_{K-1,1} \\ q_{K,2} &= \frac{\lambda}{m\mu}\left(q_{K-1,2} + q_{K-1,1}\right) \end{aligned} \quad \text{and,} \tag{6}$$

Using the normalization condition, $q_{0,0}$ can be determined as follows:

$$q_{0,0} + \sum_{k=1}^{K} q_{k,1} + q_{k,2} = 1$$

Dividing both sides by $q_{0,0}$, we get

$$p_0 = q_{0,0} = \frac{1}{1 + \sum_{k=1}^{K}\left(\frac{q_{k,1}}{q_{0,0}} + \frac{q_{k,2}}{q_{0,0}}\right)} \tag{7}$$

Note that $q_{0,0}$ denotes the probability that the system is empty, i.e., p_0.

Equation (7) enables us to compute $q_{0,0}$ by first computing the terms $q_{k,1}/q_{0,0}$ and $q_{k,2}/q_{0,0}$, which requires only λ, μ, r, m, K, N. Obtaining $q_{0,0}$ can then be used to find all other state probabilities $\{q_{k,n};\ 1 \le k \le K,\ n = 1, 2\}$.

Algorithm 1 shows how we can recursively obtain all state probabilities using Eqs. (1–7). The computation of Algorithm 1 is optimized by first computing loop invariants (as those expressions involve λ, μ, r and m) as shown in Line 4. Then, the algorithm computes the terms $q_{k,1}/q_{0,0}$ and $q_{k,2}/q_{0,0}$ recursively, as shown in Lines 5–21. In Line 22, the algorithm uses Eq. (7) to compute $q_{0,0}$. At the end, the algorithm updates the other state probabilities by multiplying the matrix Q with the scalar value $q_{0,0}$ as shown in Line 23.

Algorithm 1: Determining all state probabilities including $q_{0,0}$

```
Input:  The values of $\lambda, \mu, r, m, K, N$
Output: $q_{0,0}$ and Matrix $Q[1..K,1..2]$

01  for all $i$ and $j$ such that $1 \le i \le K$ and $1 \le j \le 2$ do
02      $P[i,j] = 0$
03  end for
04  $C_1 = \lambda/\mu$; $C_2 = \lambda/r$; $C_3 = \mu/r$
05  $Q[1,2] = C_1$
06  $Q[1,1] = C_1 \times (C_2 + C_3)$
07  $Q[2,2] = ((C_1 \times C_3 + 1) \times Q[1,1]/C_3 - C_1)/2$
08  for $i$= 2 to $m$-1
09      $Q[i,1] = (C_2 + i \times C_3) \times Q[i,2] - C_2 \times Q[i-1,2]$
10  end for
11  for $i$= $m$ to $K$-1
12      $Q[i,1] = (C_2 + i \times C_3) \times Q[i,2] - C_2 \times Q[i-1,2]$
13  end for
14  for $i$= 3 to $m$-1
15      $Q[i,2] = [(1 + 1/(C_1 \times C_3)) \times Q[i-1,1] - Q[i-2,1]] \times C_1/k$
16  end for
17  for $i$= $m$ to $K$
18      $Q[i,2] = [(1 + 1/(C_1 \times C_3)) \times Q[i-1,1] - Q[i-2,1] \times C_1/m$
19  end for
20  $Q[K,1] = C_2 \times Q[K-1,1]$
21  $Q[K,2] = C_1/m \times (Q[K-1,2] + Q[K-1,2])$
22  $q_{0,0} = 1/(1 + Sum(Q))$
23  $Q = q_{0,0} \times Q$
24  return  $q_{0,0}$ and $Q$
```

Consequently, key features and performance measures can be derived as follows. First, the mean system throughput γ is basically the departure rate, or equivalently the rate at which the requests are being processed successfully by the compute instances, that is

$$\gamma = \mu \sum_{k=1}^{K} q_{k,2}. \tag{8}$$

The mean throughput γ can be used in engineering the mean network bandwidth required for both internal and external network communication. Given the network bandwidth required for each request, the total capacity can be estimated and, accordingly, the business cost in terms of bandwidth can be determined. For a cloud provider, the estimation of throughput γ can also be used in computing the profit, if the charges for each request are known.

The probability P_{loss} is the loss or blocking probability. P_{loss} can be expressed as the probability of being in either state (K, 1) or state (K, 2), that is

$$P_{loss} = q_{K,1} + q_{K,2}. \tag{9}$$

The loss rate of requests is given by λP_{loss} requests per time unit. This can be useful in estimating and quantifying business loss as requests are being served.

The probability of queueing $P_{Queueing}$ (or the probability of all instances including the LB are busy) can be expressed as

$$P_{Queueing} = P(\geq m \text{ requests in system}) = \sum_{k=m+1}^{K} q_{k,1} + q_{k,2}.$$

The mean number of requests $E[n]$ in the system can be expressed as

$$E[n] = \sum_{k=0,n=1,2}^{K} kq_{k,n} = \sum\nolimits_{k=1}^{K} k\big(q_{k,1} + q_{k,2}\big).$$

At the cloud datacenter, the metric $E[n]$ can be useful in estimating the required network, database, or storage resources if these required resources for each request are known a priori.

Using Little's formula, the mean time spent in the system by a request succeeding in entering the queue can be expressed as

$$W = \frac{E[n]}{\gamma} = \frac{1}{\gamma}\sum_{k=0}^{K-1} k\big(q_{k,1} + q_{k,2}\big) + \frac{K}{\gamma}\big(q_{K,1} + q_{K,2}\big). \tag{10}$$

This metric W is the mean service time or sojourn time. It is this metric that the SLO has to satisfy and from it the number of VM instances can be determined, as will be demonstrated in Sect. 4.

This gives the mean time spent waiting in the queue W_q as

$$W_q = W - 1/r - 1/\mu = \frac{1}{\gamma}\sum_{k=0}^{K-1} k\big(q_{k,1} + q_{k,2}\big) + \frac{K}{\gamma}\big(q_{K,1} + q_{K,2}\big) - 1/r - 1/\mu. \tag{11}$$

We can also derive the mean number of requests in the queue $E[n_q]$ as

$$E\big[n_q\big] = \sum_{k=m+1,n=1,2}^{K} kq_{k,n} = \sum_{k=m+1}^{K} k\big(q_{k,1} + q_{k,2}\big). \tag{12}$$

Using Little's result, W_q can also be derived as

$$W_q = \frac{E[n_q]}{\gamma} = \frac{1}{\gamma} \sum_{k=m+1}^{K-1} k\left(q_{k,1} + q_{k,2}\right) + \frac{K}{\gamma}\left(q_{K,1} + q_{K,2}\right). \tag{13}$$

The CPU utilization of each VM instance can be expressed as follows

$$U_{VM} = \frac{\gamma}{m\mu}, \tag{14}$$

where γ is expressed in Eq. (8). Also the CPU utilization of the LB can be expressed as

$$U_{LB} = \frac{\gamma}{r}. \tag{15}$$

The CPU utilizations of U_{VM} and U_{LB} reflect the utilization level of compute resources, and it is a measure currently being used in Amazon AWS auto scaling [2]. A mean low level of utilization over a long period of time indicates poor utilization of resources, resulting in a high cost and inefficient system.

2.2 Scaling the Load Balancer (LB)

Our model estimates key performance metrics regardless of how small or large the processing capacity of the LB when compared to the processing capacity of all servers. That is, the derived equations are valid under the conditions $r \geq m\mu$ and $r < m\mu$. As an intuitive design principle, the LB must not be a performance bottleneck. Specifically, the service time for handling the request at the LB should not be the dominating factor in estimating the mean service time or other performance measures. Rather, the overall performance should be dominated by the processing capacity and the number of provisioned VM instances. The LB service time should be kept at a minimum, and it should have negligible impact on the overall performance of the system. To accomplish this, the LB processing capacity r should be always greater than the processing capacity of all servers $m\mu$. This means the condition $r \geq m\mu$ must always be satisfied.

However, considering the dynamism of the system due to the provisioning of more VM instances as the incoming load λ increases, the cloud may be subject to the following two situations. The first situation arises when the departure rate of the LB (i.e., $r\sum_{i=1}^{K} q_{i,1}$) is approaching the processing capacity of all servers $m\mu$ and no more VM instances can be provisioned. The second situation arises when the incoming load λ is approaching the processing capacity r of the LB. In these situations, the LB has to be scaled. Two questions arise when scaling the LB. First, what type of scaling must be done to increase the processing capacity of the LB? Second, when should such scaling should be triggered?

Two options exist for scaling the LB (and also compute VMs) processing capacity: vertical scaling and horizontal scaling. The former can be accomplished by adding more computer power (i.e., CPU power or CPU cores), physical RAM, Cache, hard disk, and network bandwidth. The latter can be accomplished by provisioning and launching new LB instances. In a cloud environment, scaling up is not a viable option. Most common operating systems do not support on-the-fly scaling up without rebooting [34], and the VM startup time can be significant and in the order of 30–100 s [8, 9]. Clearly, this will cause a major disruption in the operation of the elastic service. Therefore, horizontal scaling becomes the more practical option and can be carried out without noticeable disruption, especially if the instantiation of the new LB instances is triggered early, as it takes at least 30 s to instantiate a replica [8]. More importantly, the performance of both types of scaling is similar. Based on queueing theory performance results, a queueing system with a server that has a service rate $m\mu$ will give a comparable performance to the same queueing system but with m servers; each having a service rate μ [35–37]. Amazon AWS cloud already adopts a similar strategy using "Elastic Load Balancing" by providing replicated VM instances [2].

The triggering condition to scale out the LB is a key design issue. Early provisioning may lead to waste in cloud resources and poor utilization. As stated earlier, our aim is to ensure that the LB has negligible impact on the overall performance, and thereby ensures that the overall performance is dominated by the provisioned VM instances. To set the proper triggering condition for scaling out the LB, we apply the principle of the queueing theory to study the performance curves (e.g., throughput and response time) with respect to incoming request intensity $\rho = \lambda/\mu$.

Figure 4 shows the performance curves of an *M/M/1/K* queueing system with respect to request intensity ρ. As shown in the figure (and this can be shown for general cases), the LB performance, specifically in terms of service time, starts to be noticeably affected when the offered load ρ is greater than 0.70, but its performance in terms of throughput continues to increase. There is clearly a tradeoff between those performance measures: throughput, service time, CPU utilization and request

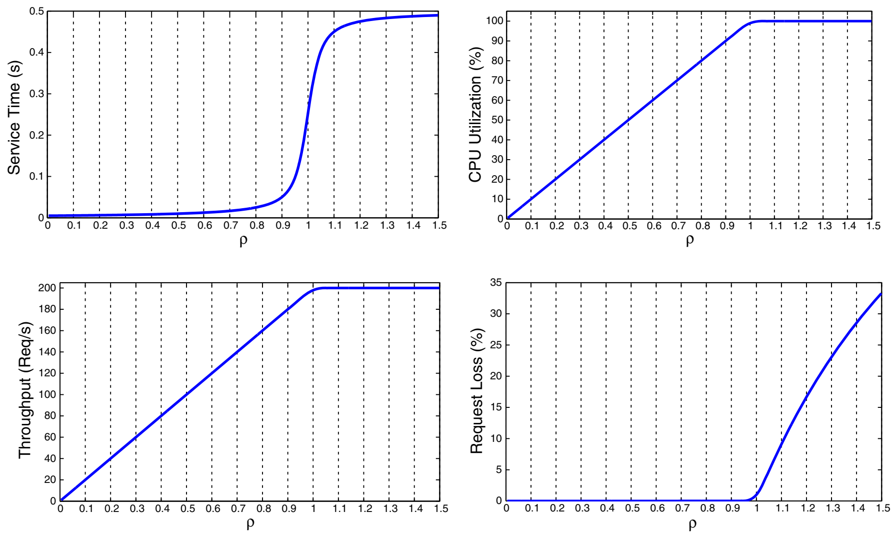

Fig. 4 Performance curves of *M/M/1/K* queueing system in relation to request intensity ρ

loss. As suggested by Kleinrock [39], we can define a function $f(\lambda) = (\gamma \times U)/W$ that identifies the operating points where the LB delivers its best performance in terms of throughput (γ), CPU utilization (U), and delay (W), collectively. The function f allows us to study the performance of the LB using a single homogeneous performance measure.

Now, let $f(\lambda)$ be a continuous and non-decreasing function. Then the function f is maximized at $\lambda = \lambda^*$ when $df/d\lambda|_{\lambda=\lambda^*} = 0$ and $d^2f/d\lambda^2|_{\lambda=\lambda^*} < 0$. Considering $M/M/1$ as we are interested only when $0 \leq \rho < 1$, the function f can be written as:

$$f = \frac{\lambda^2}{\mu}(\mu - \lambda).$$

Then,

$$\frac{df}{d\lambda} = 2\lambda - 3\frac{\lambda^2}{\mu}.$$

Thus, the critical points are $\lambda = 0$(rejected) and $\lambda = 2/(3\mu)$. Consequently, the LB delivers its best performance when $\rho = 0.67$ after which its performance degrades. It degrades dramatically when $\rho >= 0.75$. Therefore, as an acceptable engineering practice, a triggering condition to scale out the LB should take place when $\rho \approx 0.70$, which corresponds to a CPU utilization of 70 %. For implementation, measuring the overall CPU utilization over a reasonably adequate time period (e.g., 10–15 min as recommended by Amazon AWS [35]) may be used as a trigger. However, and as stated in Sect. 1, measurement of CPU utilization can be often misleading and such a practice should not be recommended, as the CPU utilization is impacted by many tasks and processes not related to the elastic service workload. Instead, measurement of ρ should be used [23].

Figure 5 illustrates the scaling out of the LB from one LB instance to two instances. After scaling, each LB will have its own queue and IP address, with each LB responsible for balancing the load among its group of m compute VM instances. Using our analytical model, the proper size of the group m can be determined based on the effective incoming load λ and a given SLO criterion as that of the service

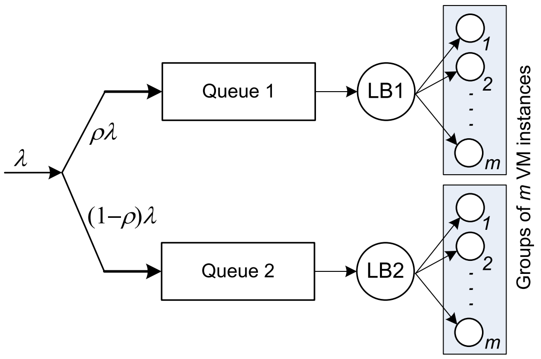

Fig. 5 Scaling out the LB instance to more than one replica

response time. The figure shows that incoming request flow for the elastic service should be shaped (or limited) so that a fraction of the total incoming arrival rate λ should not exceed $\rho\lambda$, or 0.70λ if we set $\rho = 0.70$. The rest of the traffic flow $(1 - \rho)\lambda$ should be directed to the newly provisioned and instantiated LB replica, i.e. LB2. The shaping or distribution of the traffic request flow can be done by the DNS server or Elasticity Controller [34, 38], or possibly via a traffic shaper implemented at the Internet gateway of the datacenter.

3 Verification and Validation

To verify the correctness of our analytical model, we have developed a discrete-event simulation to capture the behavior of an elastic service hosted in the cloud that comprises a finite queueing system with an LB and multiple servers. To verify the correctness of our analysis, we adopted the same assumptions as those of our analysis. The simulation was written in C language, and the code closely and carefully followed the guidelines given in [34]. We used the PMMLCG as our random number generator [40]. The simulation was automated to produce independent replications with different initial seeds that were ten million apart. During the simulation run, we checked for overlapping in the random number streams and ascertained that such a condition did not exist. The simulation was terminated when achieving a precision of no more than 10 % of the mean with a confidence of 95 %. We employed and implemented the *replication/deletion* approach for means discussed in [40]. We computed the length of the initial transient period using the Marginal Confidence Rule (MCR) heuristic developed by White [41]. Each replication run lasts for five times of the length the initial transient period.

To validate our analytical model, we also compared our analysis results to real world experimental measurements. Figure 6 illustrates the experimental setup and testbed. We used the AWS cloud platform to launch Amazon EC2 VM instances

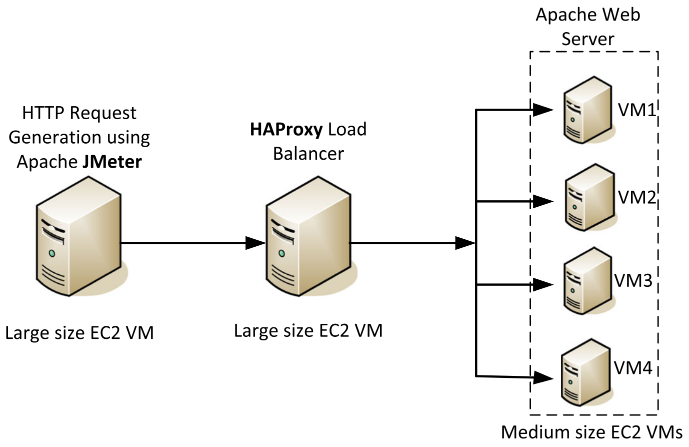

Fig. 6 Experimental setup deployed on AWS Cloud

[4]. The experiment comprised different sizes of EC2 VMs. We selected the size of EC2 VMs that are optimized for our processing and network connectivity needs. We used one large size EC2 VM for generating an HTTP request traffic, one large size EC2 VM for load balancing, and a medium size EC2 VM (namely "m1.medium") for servicing the HTTP requests. These types of VMs (when compared to micro and small VMs) offer high networking performance in terms of inter-VM latencies, lower network jitter and significantly higher packet per second performance. In the AWS cloud, a large size EC2 VM has two virtual CPUs (vCPUs) with 7.5 GB RAM, and a medium sized EC2 VM has one vCPU and 3.7 GB RAM. In AWS terminology, one vCPU is a hardware hyperthread from a 2.6 GHz Intel Xeon E5-2670 processor [42]. We ran Ubuntu Linux Server 13.10 as the underlying operating system for all of our instances. We used the popular Apache JMeter [43] to generate synthetic HTTP traffic which was first directed to an HAProxy load balancer [45] that was readily available at the AWS [46]. In addition to the basic features that come with JMeter by default, we added the standard set of plugins [44] to create aggregate summary reports. The HAProxy was configured to distribute the HTTP requests evenly in a round robin fashion among the available medium sized VM instances to process the HTTP requests. All of these VMs including the JMeter and HAProxy were hosted within the same AWS VPC (Virtual Private Cloud) [47]. With Amazon VPC, all hosts within the VPC are logically isolated. As an alternative to using Appache JMeter, the open-source D-ITG traffic generator [48, 49] can be used to generate realistic HTTP traffic with parameter values set as described in [50, 51].

Apache JMeter was configured to generate HTTP requests simultaneously, and performance measurements were taken after a period of 15 min. For our measurement, we measured the following performance metrics: response time, throughput, and CPU utilization. The average, minimum, and maximum measurements for response time and throughput were given by the JMeter aggregate summary report at the end of the run. As for CPU utilization, we used *sar* Linux utility and Perl scripting language to collect and aggregate the CPU utilization readings of the running VMs. The CPU utilization readings were taken in the stable period; namely from 8 to 12 min.

We measured the average processing time $1/r$ for the HAProxy load balancer and the average processing time $1/\mu$ for processing HTTP requests. In general and in reality, the measurement techniques laid out here can be used to gauge the processing capacity of any LB and VM instances. For measuring $1/r$, we generated HTTP traffic from JMeter to HAProxy and back to JMeter. For this, we had to run JMeter and Apache Web Server in the same large sized EC2 VM instance. Similarly, for measuring $1/\mu$, we generated traffic directly from the JMeter large instance to the EC2 medium size VM instance. We set up the Web server to return a web page of a size of 580 bytes when receiving an HTTP request from JMeter. The page size was tuned until we achieved close to 100 % CPU utilization when the instance receives 100 Req/s (Requests per second) which is our target capacity of our medium sized VM instance. Intuitively, this will give us around 10 ms for the average service time $1/\mu$ when servicing HTTP requests at a low rate. To verify this further, we measured the average service time of both $1/r$ and $1/\mu$ by computing the

JMeter response time average when sending an HTTP traffic rate λ of 10 Req/s. We found that the mean value for $1/r$ and $1/\mu$ was approximately 0.1 and 10 ms, respectively. Also, for our experiments, we have chosen the buffer size $K = 300$ which is the default size for the Linux network adapter's receiving buffer according to the header definition in the Linux kernel code header file*/net/drivers/tg3.h*

4 Numerical Results

In this section, we report numerical results obtained by using our analytical model, simulation, and experimental measurements. The analytical curves were obtained by MATLAB implementation of the equations derived from the analytical models. The simulation results were obtained using the discrete-event simulation described in Sect. 3. Likewise, the experimental measurements were obtained from the testbed described in Sect. 3. As depicted in Fig. 7, the results obtained from the simulation are represented by the red circles, whereas the analysis results are represented by solid blue curves. The figure shows that both simulation and analysis results are in close agreement, and thus imply that our analytical model is correct. In addition, Table 1 shows that our real experimental measurements are also in adequate agreement with analytical results, thereby validating the analytical model.

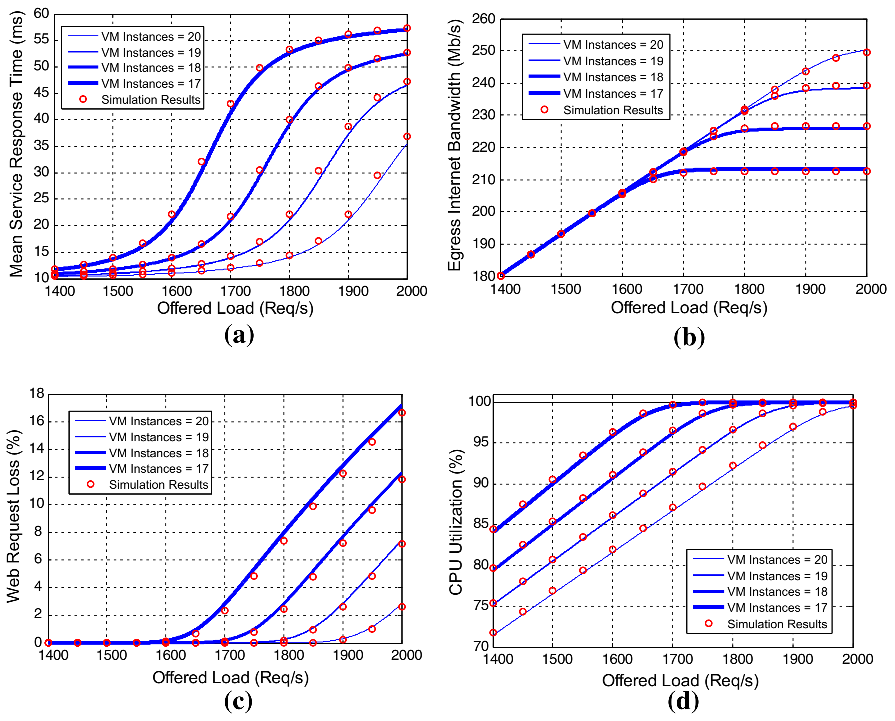

Fig. 7 Web performance curves using multiple instances

Table 1 Comparison of experimental results to analysis

	Response time (ms)				Throughput (Req/s)				CPU utilization (%)			
	Analysis	Experiment			Analysis	Experiment			Analysis	Experiment		
	Avg	Avg	Min	Max	Avg	Avg	Min	Max	Avg	Avg	Min	Max
5 VMs at a rate of 400 Req/s	15.54	17.8	12.23	21.61	400	379	360	392	80	73	54	87
10 VMs a rate of 800 Req/s	12.05	14.29	10.11	16.23	800	768	751	788	80	76	56	89
20 VMs at a rate of 1500 Req/s	10.3	14.88	9.71	17.74	1500	1461	1357	1488	75	66	51	84
30 VMs at a rate of 2500 Req/s	10.5	15.01	7.12	18.02	2500	2391	2202	2601	83	75	57	91

We examine and give numerical results for three realistic scenarios in which our analytical model is used to study the performance of elastic cloud services. The first scenario is hosting a cloud web service with different workloads. We present results for key performance metrics as a function of the incoming web workload. These metrics include mean response time, egress Internet bandwidth, request loss, and CPU utilization. In the second scenario, we examine the required bandwidth and business loss for Netflix's streaming video service hosted in the AWS cloud. In the third scenario, we validate our analysis by conducting a real experiment on the Amazon AWS cloud platform, as illustrated in our testbed of Fig. 6. Experiment measurements for latency, throughput, and CPU utilization are compared with the analytical results.

4.1 Web Service

In this example, we assume an elastic web service is being hosted on the cloud. The incoming web requests are dispatched by a load balancer equally to a number of VM instances running the web service. We fix the system size K to 300 requests. We fix $1/r = 0.2\,\text{ms}$ and $1/\mu = 10\,\text{ms}$. The mean service rate μ is realistic and consistent with the reported experimental rates in [18, 52–54], and this service includes CPU processing in addition to any required disk I/O or database access.

Figure 7 plots key performance metrics for the cloud web service hosted by multiple instances in relation to the incoming load. The figure plots mean response time, Internet network bandwidth, web request loss, and CPU utilization of used instances. Figure 7(a) shows that the mean response time for the given VM instances is relatively small under a light load, i.e. when λ is smaller than 1400 Req/s. However, as the offered load increases, the figure clearly shows the response time becomes highly affected by the number of allocated VM instances. As shown, a web service using 17 instances yields a significantly higher response time than a service that is using 20 instances. It is worth noting that the saturation point of the cloud service occurs when approximately the offered load λ approaches $m\mu$. For example, the saturation point when using 17 instances occurs at $m\mu = 17 \times 100 = 1700$ Req/s. From queueing theory properties [35], the mean response time reaches a plateau at $K/m\mu = 100/1700 = 58$ ms. The saturation points for different instances are clearly shown in Fig. 7c in which a significant increase of request loss is exhibited. Figure 7c shows that the saturation points are approximately 1700, 1800, 1900, and 2000 when using 17, 18, 19, and 20 instances, respectively. The request loss curves are computed using Eq. (9).

Figure 7b shows the bandwidth required in Mb/s for egress (or outgoing) Internet traffic. Such a performance metric can be used to properly engineer the network bandwidth in the cloud datacenter. If we assume each incoming request returns one HTML page of size around 1600 bytes, as reported in [52]. Under this assumption, the required egress bandwidth in Mb/s is approximated to be 1600 bytes in addition to TCP + IP + Ethernet headers, which all is equal to $20 + 20 + 26$. For an Ethernet frame of a maximum size of 1500 bytes, the HTML page has to be segmented into two packets resulting in a total size of $2 \times (20 + 20 + 26) + 16{,}000 \approx 16{,}092$ bytes or 128.8 Kbits. Given an incoming rate, the overall egress

Internet bandwidth can be computed by multiplying 128.8 Kbits times the mean throughput given by Eq. (8).

To illustrate how the model predicts the number of required VM instances needed to satisfy a given response time, we closely examine the curves for the mean system delay and CPU utilization as shown in Fig. 7a, d, respectively. We focus on the area of interest, in which the latency starts increasing, that is, when the arrival rate λ is between 1400 and 2000 Req/s. As expected, smaller latencies (and lower CPU utilizations) are exhibited with a larger number of VM instances. Given a measured offered load, the minimum number of VM instances can be determined to satisfy a given response time. For example, if the SLO service response time to satisfy (excluding network delays) is 15 ms, and the expected offered load (set at the start for the elastic service or measured later by the LB) is 1500 Req/s, then the needed VM instances will be 17, as shown in Fig. 7a. However, if the offered load grows to 1600 Req/s, the needed VM instances will be 18, and so on. Figure 7d shows the corresponding CPU utilizations are slightly over 90 % for 17 and 18 VM instances at an offered load of 1500 and 1600 Req/s, respectively. In AWS auto-scaling, a threshold of 85 % (set arbitrarily by the user) is typically recommended to trigger scaling out (i.e. adding more VM instances) so that a service response time can be met [2]. However, and as shown, this arbitrary threshold is not appropriate in determining the needed number of VM instances to guarantee a given service response time. As shown in Fig. 7d, at 1500 Req/s, and with 85 % CPU utilization, 20 VM instances are needed to maintain the response time below 15 ms. But according to Fig. 7a, 17 VM instances will satisfy the 15 ms latency requirement. This leads to unnecessary over provisioning and poor utilization of cloud resources, and thereby result in a higher system cost for the cloud customer. In conclusion, the VM instances should be sized based on the given response time.

4.2 Netflix Video Streaming

This example shows how our model can be used to estimate the required bandwidth and business loss of on-demand Internet streaming media services hosted in the cloud. Netflix is a leader provider of streaming movies and TV shows on a pay-per-stream basis. As of 2010, Netflix had migrated all its compute and storage infrastructures to the Amazon AWS cloud [42]. For our numerical example, we assume the viewing time of TV and video sessions is 25 min on average. This viewing time is reasonable for streaming sessions of TV shows and videos. Typically, viewers will watch a movie and pause and watch the rest at a later time. We also assume that each VM instance can handle 20 streams on average at one time. The handling includes streaming tasks such as processing, monitoring, logging and accounting. In addition, we assume that the mean required bandwidth per stream is 2600 Kb/s as reported in [54, 55]. This bandwidth is the total bandwidth including payload in addition to header bytes for RTP and IP protocols. We fix the buffer size K to 20 requests.

Figure 8 plots the required video streaming bandwidth and request loss in relation to the offered load λ when running the streaming service under 10, 15, and 20 VM instances. The figure shows that respective saturation points for these VM instances

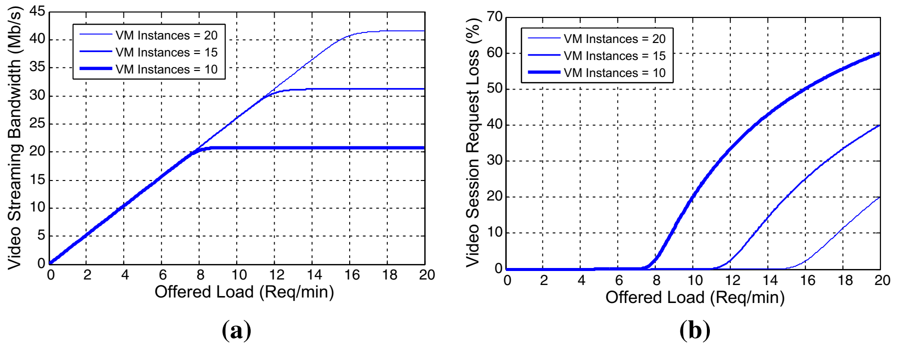

Fig. 8 Netflix video service performance curves

are approximately 8, 12, and 16 Req/min. This is in line with intuition as the saturation point ought to occur when the offered load λ approaches $m\mu$. For example, the saturation point for 20 VM instances will be approximately $m\mu = 20 \times (20/25) = 16$ Req/min. Figure 8a exhibits the Internet bandwidth that is needed for streaming out of the cloud datacenter to the Internet. This is the traffic that the ISP will have to carry to the Netflix customers. The required bandwidth is computed using the mean throughput given by Eq. (8) and multiplying it by 2600 Kb/s.

Figure 8b depicts the curves of a video session request loss. The figure shows that at a light load, there is almost no request loss. However, at a heavy load (beyond incoming rate of 8 Req/min), a significant loss occurs with a more noticeable loss when using smaller numbers of VM instances. This figure can be useful in estimating business loss given a certain capacity for cloud resources. For example, if Netflix has a pricing model where users are charged 2 ¢/min (i.e., 2 US cents per minute) of viewing. Next, we illustrate the idea of using our model to estimate business loss. In the example above, the business loss when using 10 cloud instances and at an incoming rate of 20 Req/s can be computed as $2 \times 20 \times P_{loss} = 2 \times 20 \times 60\ \% = 24$ ¢/min. Similarly, the business loss when using 15 and 20 instances is 16 and 8 ¢/min, respectively.

4.3 Experimental Measurements

In this subsection we report the experimental measures of the testbed described in Sect. 3. Table 1 validates our analytical model by comparing experimental results with analytical results for three performance measures of response time, throughput, and CPU utilization. The reported experimental results shown in the table are the average of five runs. We found that five runs are sufficient and yield adequate measurements. Conducting more runs would have little impact on the obtained aggregate results. As described in Sect. 3, the measurements were taken for each run with the Apache JMeter generating HTTP traffic flow for a duration of 15 min. We chose to run the traffic flow for 15 min to account for the fluctuation and variability exhibited in the AWS cloud environment due to virtualization and other workload and network activities that might be presented by other cloud-hosted services and applications. The measurements of these experiments were taken around 12 midnight GST time in the Amazon AWS zone located in Ireland. It is worth mentioning that at the end of each run, JMeter reports key statistics that include the request response time and throughput. However, JMeter, as of the time of this writing, does not report measurement of the request loss.

We report experimental results in Table 1 and compare these results with the average results obtained from Analysis. Appache JMeter For the shown experimental measurements, we recorded the minimum, maximum, and average values for the five runs. We considered four different numbers for compute VMs with specific workloads. The workloads were selected to be close to the saturation point of the VMs, i.e. when the VM's CPU utilization reached close to 80 %. Specifically, we considered 5, 10, 20, and 30 VMs, with 400, 800, 1500, and 2500 Req/s, respectively. A number of observations can be made from Table 1. First, the average experimental measurements followed the same trend and pattern of the analysis results. In a way, our analytical results are in good agreement with the experimental measurements. This obviously validates our analytical model. Second, it was observed that the response times obtained from the experiments, in all cases, were larger than those of the analysis. The reason for this is that the analysis average response times account for the one-way delays incurred at the LB and VMs, and it do not account for the additional processing delays incurred at the JMeter machine associated with the HTTP request generation and then the reception and statistics recording of the HTTP replies. So, an increase of 2–3 ms delays was expected. Third, it was observed that the throughput measurements were slightly less than those of yjr analysis. As shown from the table, the difference is in the order of 5 %. At a low rate of 400 Req/s, the average measured throughput was 379 Req/s, and at a high rate of 2500 Req/s, the average measured throughput is 2391 Req/s. It is not certain why this is so, but it can be attributed to the ability of JMeter to generate the exact rate. This can also be attributed to the virtualization and network conditions of the cloud environment. Finally, when examining the min and max values for the recorded measurements, it is clear that the experimental results exhibit considerable fluctuation and variability. The main reason for this is due to the virtualization technology implemented at different levels in the cloud data center. Virtualization can be applied to physical machines, network devices, and disk storage. In addition, with the AWS cloud, we have no control over where the VMs are co-located, i.e., within the same network or different networks. Moreover, significant fluctuation can be attributed to other activities carried out by other cloud-hosted services and applications that might be running simultaneously in the cloud. These cloud services and applications can introduce significant workload and traffic on the shared network devices and links, and therefore, impact the overall network delay and performance.

5 Conclusion

In this paper, we have presented an analytical model that can be used to achieve proper elasticity for cloud-hosted applications and services. Particularly, given the offered workload and the processing capacity of each VM, the model can predict the minimal number of VMs required to satisfy a particular SLO criterion. In addition, the model can predict the required number of load balancers needed to achieve proper elasticity. We have demonstrated how the model can be used in capacity engineering and the estimation of cloud compute and network resources for different real world scenarios that include cloud-hosted web services and Netflix streaming media service. We have validated our analytical model by obtaining experimental measurements from a testbed setup on the Amazon AWS cloud. Noticeable fluctuation and variability were exhibited in the experimental measurements, but the overall mean measurements were in adequate agreement with analysis results. The fluctuation in measurements can be attributed to virtualization of the cloud compute and network resources in addition to activities of other cloud-hosted services and applications. As a future work, plans are underway to build an elastic cloud-hosted web service on the Amazon AWS IaaS cloud platform. We will utilize our derived analytical formulas to estimate the minimal number of VMs needed to satisfy a given SLO response time. The elasticity and the performance of such a cloud service will be evaluated. Moreover, we plan to devise novel schemes to best estimate a suitable interval to trigger the adjustment of the required VMs as well as to best measure and estimate the mean workload. The estimation of the mean workload can be a challenging task as traffic can be highly fluctuating with intermittent unexpected workload spikes. As future research directions, it will be interesting to study the impact of network utilization generated by the co-located VMs and hosted services on guaranteeing the SLO latency. For this, the placement of the minimal number of VMs and LBs within the datacenter becomes critical. Improper placement of a VM within a rack, which hosts highly active VMs, can result in violating the SLO latency.

Acknowledgments We would like to acknowledge the reviewers for their invaluable comments and feedback that tremendously enhanced the quality of our work. Moreover, the experimental work in this paper was supported by a generous research Grant provided by Amazon AWS in Education [56].

References

1. Azeez, A.: Auto-scaling web services on Amazon EC2 (2014). http://people.apache.org/~azeez/autoscaling-web-services-azeez.pdf
2. Amazon Inc.: Amazon web services auto scaling (2014). http://aws.amazon.com/autoscaling
3. Aceto, G., Botta, A., de Donato, W., Pescape, A.: Cloud monitoring: a survey. J. Comput. Netw. **57**(9), 2093–2115 (2013)
4. Amazon Inc.: AWS web services (2014). http://aws.amazon.com/
5. Google Inc.: Google compute engine (2014). https://cloud.google.com/products/compute-engine/
6. Google Inc.: Google App Engine (2014). http://appengine.google.com/
7. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. **12**(4), 559–592 (2014)
8. Lagar-Cavilla, H, Whitney, J., Scannell, A., Patchin, P., Rumble, S., Lara, E., Brudno, M., Satyanarayanan, M., SnowFlock: rapid virtual machine cloning for cloud computing. In: Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys'09, Nuremberg, Germany, March 2009, pp. 1–12
9. Mao, M., Humphrey, M.: A performance study on the MV startup time in the cloud. In: Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD2012), June 2012, pp. 423–430

10. Iqbal, W., Dailey, M., Carrera, D., Janecek, P.: Adaptive resource provisioning for read intensive multi-tier applications in the cloud. J. Future Gener. Comput. Syst. **27**(6), 871–879 (2011)
11. Liu, H., Wee, S.: Web server farm in the cloud: performance evaluation and dynamic architecture. In: Proceedings of the 1st 2009 International Conference on Cloud Computing, Springer, Berlin, pp. 369–380 (2009)
12. Wang, Z., Chen, Y., Gmach, D., Singhal, S., Watson, B., Rivera, W., Zhu, X., Hyser, C.: AppRAISE: application-level performance management in virtualized server environments. IEEE Trans. Netw. Serv. Manag. **6**(4), 240–254 (2008)
13. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., Wood, T.: Agile dynamic provisioning of mult-tier internet applications. ACM Trans. Auton. Adapt. Syst. **3**, 1–39 (2008)
14. Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A.: An analytical model for multi-tier internet services and its applications. In: Proceedings of the 2005 ACM SIGMETRICS International Conference, vol. 33, Alberta, Canada, pp. 291–302
15. Khazaei, H., Misic, J., Misic, V.: Performance analysis of cloud computing centers using M/G/m/m + r queueing systems. IEEE Trans. Parallel Distrib. Syst. **23**(5), 936–943 (2012)
16. Kikuchi, S., Matsumoto, Y.: Performance modeling of concurrent live migration operations in cloud computing systems using PRISM probabilistic model checker. In: Proceedings of the 4th IEEE International Conference on Cloud Computing, Melbourne, Australia, pp. 49–56 (2011)
17. Firdhous, M., Ghazali, O., Hassan, S.: Modeling of cloud system using Erlang formulas. In: Proceedings of the 2011 7th Asia-Pacific Conference on Communications (APCC), Saba, Malaysia, October, pp. 411–416 (2011)
18. Xiong, K., Perros, H.: Service performance and analysis in cloud computing. In: Proceedings of the 2009 IEEE Congress on Services, Los Angeles, Californian, July 2009, pp. 693–700
19. Wuhib, F., Yanggratoke, R., Stadler, R.: Allocating compute and network resources under management objectives in large-scale clouds. J. Netw. Syst. Manag. **23**, 111–136 (2015)
20. Jennings, B., Stadler, R.: Resource management in clouds: survey and research challenges. J. Netw. Syst. Manag. **23**, 567–619 (2015)
21. Chunlin, L., Layuan, L.: Multi-layer resource management in cloud computing. J. Netw. Syst. Manag. **22**(1), 100–120 (2014)
22. Salah, K., Boutaba, R.: Estimating service response time for elastic cloud applications. In: Proceedings of the 1st IEEE International Conference on Cloud Networking (CloudNet 2012), Paris, France, 28–30 November 2012, pp. 12–16
23. Cockcroft, A.: Utilization is virtually useless as a metric. In: Proceedings of CMG 2006 Conference, December 2006
24. Salah, K.: Implementation and experimental evaluation of a simple packet rate estimator. AEU Int. J. Electron. Commun. **63**(11), 977–985 (2009)
25. Salah, K., Haidari, F.: Performance evaluation and comparison of four network packet rate estimators. AEU Int. J. Electron. Commun. **64**(11), 1015–1023 (2010)
26. Salah, K., Haidari, F.: On the performance of a simple packet rate estimator. In: IEEE/ACS International Conference on Computer Systems and Applications, 2008. AICCSA 2008 (2008)
27. Andersson, M., Bengtsson, A., Host, M., Nyberg, C.: Web server traffic in crisis conditions. In: Proceedings of the rd Swedish national computer networking workshop. Nov 2005
28. Leland, W., Taqqu, M., Willinger, W., Wilson, D.: On the self-similar nature of ethernet traffic. IEEE/ACM Trans. Netw. **2**(1), 1–15 (1994)
29. Paxson, V., Floyd, S.: Wide-area traffic: the failure of poisson modeling. IEEE/ACM Trans. Netw. **3**(3), 226–244 (1995)
30. Willinger, W., Taqqu, M., Sherman, R., Wilson, D.: Self-similarity through high-variability: statistical analysis of ethernet LAN traffic at the source level. In: Proceedings of ACM SIGCOMM, Cambridge, Massachusetts, pp. 100–113, Aug 1995
31. Salah, K., Elbadawi, K., Boutaba, R.: Performance modeling and analysis of network firewalls. IEEE Trans. Netw. Serv. Manag. **9**(1), 12–21 (2012)
32. Van Der Mei, R.D., Hariharan, R., Reeser, P.K.: Web server performance modeling. J. Telecommun. Syst. **16**(3–4), 361–378 (2001)
33. Chandy, K.M., Sauer, C.H.: Approximate methods for analyzing queueing network models of computing systems. J. ACM Comput. Surv. **10**(3), 281–317 (1978)
34. Vaquero, L., Rodero-Merino, L., Buyya, R.: Dynamically scaling applications in the cloud. ACM SIGCOMM Comput. Commun. Rev. **41**(1), 45–52 (2011)
35. Gross, D., Harris, C.: Fundamentals of Queueing Theory. Wiley, New York (1998)

36. Salah, K.: To coalesce or not to coalesce. Int. J. Electron. Commun. **61**(4), 215–225 (2007)
37. Jain, R.: The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling. Wiley, New York (1991)
38. Amazon Inc.: Amazon Elastic Load Balancing (2014). http://aws.amazon.com/elasticloadbalancing/
39. Kleinrock, L.: Power and deterministic rules of thump for probabilistic problems in computer communications. In: Proceeding of the IEEE ICC'79, Boston, Massachusetts, June 1979
40. Law, A., Kelton, W.: Simulation Modeling and Analysis, 2nd edn. McGraw-Hill, New York (1991)
41. White, J.: An effective truncation heuristic for bias reduction in simulation output. Simul. J. **69**(6), 323–334 (1997)
42. Amazon Inc.: Amazon EC2 instances (2014). https://aws.amazon.com//ec2/instance-types/
43. Apache JMeter: Apache.org. http://jmeter.apache.org/
44. Custom Plugins for Apache JMeter: JMeter-Plugins.org. http://jmeter-plugins.org/
45. HAProxy: 2014. http://haproxy.1wt.eu/
46. AWS Documents: HAProxy layer (2014). http://docs.aws.amazon.com/opsworks/latest/userguide/workinglayers-load.html
47. Amazon Web Services: Amazon Virtual Private Cloud Route Tables. http://aws.amazon.com/documentation/vpc/
48. Botta, A., Dainotti, A., Pescapè, A.: A tool for the generation of realistic network workload for emerging networking scenarios. Comput. Netw. **56**(15), 3531–3547 (2012)
49. Distributed Internet Traffic Generator (2014). http://traffic.comics.unina.it/software/ITG/
50. Dainotti, A., Pescape, A., Ventre, G.: A packet-level characterization of network traffic. Proceedings of the 11th IEEE Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks, pp. 38–45 (2006)
51. Salah, K., Hamawi, M.: Comparative packet-forwarding measurement of three popular operating systems. Int. J. Netw. Comput. Appl. **32**(4), 1039–1048 (2009)
52. Dejun, J., Pierre, G., Chi, C.-H.: EC2 performance analysis for resource provisioning of service-oriented applications. In: Proceedings of the 3rd Workshop on Non-functional Properties and SLA Management in Service-Oriented Computing, Nov 2009
53. Islam, S., Lee, K., Fekete, A., Liu, A.: How a consumer can measure elasticity for cloud platforms. In: Proceedings of the 3rd International Conference on Performance Engineering, Boston, MA, 22–25 April 2012
54. Mello, J.P.: Netflix rates broadband provided by bandwidth. In: PCWorld Magazine. 27 Jan 2011
55. Ward, N.: How to improve Netflix streaming (2014). http://www.helium.com/items/2067366-how-to-improve-netflix-streaming
56. Amazon Inc.: Amazon AWS Education Grants (2014). http://aws.amazon.com/education

Khaled Salah is an associate professor at the ECE Department, Khalifa University, UAE. He received the B.S. degree in Computer Engineering with a minor in Computer Science from Iowa State University, USA, in 1990, the M.S. degree in Computer Systems Engineering from Illinois Institute of Technology, USA, in 1994, and the Ph.D. degree in Computer Science from the same institution in 2000. His primary research interests are in the areas of cloud computing, cyber security, and queueing systems.

Khalid Elbadawi received his the Ph.D. degree from the School of Computing, College of Computing and Digital Media, DePaul University, USA. He received his BS degree in Mathematics and Computer Science from University of Khartoum, Sudan, in 1994. In 2001, he joined King Fahd University of Petroleum and Minerals and obtained his MS degree in 2003. His research interests are in performance analysis, cloud computing, and network resource management.

Raouf Boutaba is a University of Waterloo computer science professor. He is the founding EiC of the IEEE Transactions on Network and Service Management (2007–2010). He received several recognitions including the Fred Ellersick Prize, the Dan Stokesbury award, and the McNaughton Gold Medal. He is IEEE, EIC, and CAE fellow.