Achieving Performance and Availability Guarantees with Spot Instances

Michele Mazzucco, Marlon Dumas
University of Tartu, Estonia
{*Michele.Mazzucco, Marlon.Dumas*}*@ut.ee*

Abstract—**In the Infrastructure-as-a-Service (IaaS) cloud computing market, spot instances refer to virtual servers that are rented via an auction. Spot instances allow IaaS providers to sell spare capacity while enabling IaaS users to acquire virtual servers at a lower pricer than the regular market price (also called "on demand" price). Users bid for spot instances at their chosen limit price. Based on the bids and the available capacity, the IaaS provider sets a clearing price. A bidder acquires their requested spot instances if their bid is above the clearing price. However, these spot instances may be terminated by the IaaS provider impromptu if the auction's clearing price goes above the user's limit price. In this context, this paper addresses the following question: Can spot instances be used to run paid web services while achieving performance and availability guarantees? The paper examines the problem faced by a Software-as-a-Service (SaaS) provider who rents spot instances from an IaaS provider and uses them to provide a web service on behalf of a paying customer. The SaaS provider incurs a monetary cost for renting computing resources from the IaaS provider, while charging its customer for executing web service transactions and paying penalties to the customer for failing to meet performance and availability objectives. To address this problem, the paper proposes a bidding scheme and server allocation policies designed to optimize the average revenue earned by the SaaS provider per time unit. Experimental results show that the proposed approach delivers higher revenue to the SaaS provider than an alternative approach where the SaaS provider runs the web service using "on demand" instances. The paper also shows that the server allocation policies seamlessly adapt to varying market conditions, traffic conditions, penalty levels and transaction fees.**

I. Introduction

This paper is motivated by the challenges arising when a Software-as-a-Service (SaaS) provider runs a web service on behalf of a paying customer using (virtual) servers provided by an Infrastructure-as-a-Service (IaaS) provider. In line with contemporary pay-per-use practices, the customer pays a fixed charge (fee) to the SaaS provider per web service transaction. The service delivery is governed by a Service Level Agreement (SLAs) that includes performance and availability objectives. Failure to achieve these objectives over a period of time binds the SaaS provider to pay a penalty to the customer.

In this context, the SaaS provider seeks to maximize its net revenue, that is, the total fees charged by the SaaS provider to its customer minus the cost for renting servers from the IaaS provider and the penalties for violating SLA objectives. To maximize its net revenue, the provider seeks to reduce costs by renting servers at the lowest possible price, while still meeting the SLA objectives in order to avoid those penalties.

IaaS providers generally provide virtual servers on a per-hour basis at an "on demand" price. This price is such that the IaaS provider has enough capacity to meet all requests. In the Amazon's Elastic Cloud Computing (EC2) platform[1], customers can also reserve instances in advance (modulo a reservation fee) and pay a lower price per-hour than the "on demand" price. This option is attractive when utilization can be planned in advance. [1] and [2] have addressed the question of whether to acquire "on demand" or "reserved" instances.

In order to improve its data centers' utilization, Amazon has introduced an additional pricing model, namely spot pricing. Spot instances are virtual servers sold per hour (or fraction thereof) via an auction. Users bid for one or multiple virtual servers at a limit price (the maximum price the bidder is willing to pay per hour). Amazon gathers the bids and determines a clearing price (a.k.a. *spot price*) based on the bids and the available capacity. A bidder gets the required instances if his/her limit price is above the clearing price. In this case, the bidder pays the clearing price (not his/her limit price). The clearing price is updated as new bids arrive. If the clearing price goes above the bidder's limit price, the bidder's running spot instances are terminated.

In this paper, we consider the case where the SaaS provider uses spot instances to deliver a web service. One issue is that in case of spot instance termination due to the clearing price crossing above the limit price, the web service is unavailable until the SaaS provider acquires new servers. This may result in the provider having to pay an unavailability penalty and losing revenue from unfulfilled web service transactions. In parallel, the SaaS provider faces the decision of how many spot instances to acquire. The more instances it acquires, the better performance it can offer, but at a higher cost.

To manage these tradeoffs, we propose a revenue maximization scheme to optimize the average net revenue per time unit on behalf of the SaaS provider. The scheme relies on two components: (i) a price prediction model aimed at determining the lowest limit price to bid in order to achieve a given level of availability; and (ii) a policy for server allocation and admission control based on dynamic estimates of traffic parameters and models of system behavior. Two alternative policies are proposed. The first one makes assumptions about the nature of the user demand in order to evaluate analytically the effect of particular decisions on the maximum achievable

[1] http://aws.amazon.com/ec2/

revenues. The second one, instead, emphasizes generality rather than analytical tractability, allowing any kind of traffic.

The above revenue maximization problem does not appear to have been studied before. Perhaps the most similar related work is by Mazzucco *et al* [3], [4], but those studies do not consider the variable cost of acquiring the servers nor do they take into account the penalties arising for rejecting customers in case of outage due to spot instance termination. Andrzejak *et al* [5] present a probabilistic model aiming at minimizing the budget needed to meet the performance and reliability requirements of applications running on the Cloud, while [6] introduces an autonomic solution that given a set of goals to optimize (*e.g.*, monetary cost or execution time) selects the resources to best meet the specified target. Hu *et al* [7] investigate how to deliver response time guarantees in a multi-server and multi-class setting hosted on the Cloud by means of allocation policies only. Similarly, [8], [9] and [10] investigate how to deliver acceptable performance levels while minimizing monetary cost or electricity consumption. Stokely *et al* [11] address the resource provisioning problem in a cluster in which users bid for the available resources. Similarly, Mattess *et al* [12] consider the economics of purchasing resources on the spot market to deal with unexpected load spikes, while Chohan *et al* [13] study how to best use spot instances for speeding up MapReduce workflows and investigate how the bid price affects the likelihood of premature instance termination.

The rest of this paper is organized as follows. Section II discusses the general approach to exploit spot instances on Amazon EC2 for delivering a service with guarantees. Next, the system model and the associated QoS contract are described in Section III. The mathematical analysis and the resulting policies for server allocation and admission control are presented in Section IV. A number of experiments are discussed in Section V, while Section VI concludes the paper.

II. Spot Price Prediction

The first pillar of the proposal is a model to determine an optimal limit price for the SaaS provider to bid on the spot market. Importantly, in a (multi-unit) Vickrey auction such as that used in the Amazon spot market, bidders have an incentive to bid truthfully (*i.e.*, bid according to their value) rather than over- or under-bidding. In our context, the objective is to bid in such a way as to achieve a desired level of availability.

As displayed in Figure II the spot price variation in Amazon EC2 over time does not seem to follow any particular law. On the other hand, in order to predict what future spot prices will be, we have to gain some understanding of how they change over the time. To this end, we use the autocorrelation function (ACF), which measures the correlation of a random variable with itself at different points in time, as a function of the two times or of the time difference (lag, l) [14]

$$ACF(l) = \frac{\sum_{t=1}^{N-l}(x_t - \bar{x})(x_{t+l} - \bar{x})}{\sum_{t=1}^{N}(x_t - \bar{x})^2}, \tag{1}$$

where N is the number of observations, $\bar{x}$ is the average value, x_t is the value at time t, and x_{t+l} is the value at time $(t + l)$. The value of the ACF lies in the range $[-1, 1]$, with 1 indicating perfect correlation, -1 indicating perfect anti-correlation and 0 indicating that there is no correlation between values at different points in time. Hence, we analyzed the ACF of the spot prices for small, large and extra large Linux instances over a two months period for different lags. Figure II shows that there is almost no correlation between prices, even at lag 1. In other words, past values of the spot prices give no information about future prices. Hence, standard predictive tools such as ARIMA or Double Exponential Smoothing can not be employed to predict the spot market's behavior.

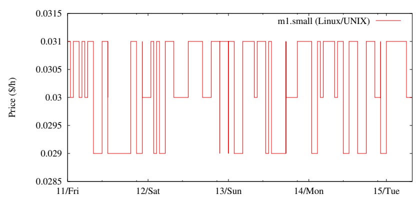
Having failed to establish a relationship between spot prices at different times, we try to exploit properties of the price distribution in order to predict how much the provider should bid. Previous studies have shown that prices in similar markets follow a normal distribution [15], [16]. Hence, we employ a normal approximation to model the spot price distribution.

In Table I we report statistics of the spot prices over the period December 25, 2010 – February 23, 2011 (*us-east-1* region) as well as the approximated values estimated from a normal distribution with the same mean and variance over 10,000 samples. According to our results the normal approximation is adequate but not perfect, as the distribution of the spot prices is more heavily-tailed: even though the first three quantiles are a good match, the minimum and maximum values in the historical data differ substantially from those of the approximation. This was confirmed also by the Q-Q plots as well as by the Shapiro-Wilk test.Similarly, the third and fourth central moment of the distributions (skewness, a measure of symmetry of the distribution, and kurtosis, a measure of the peakedness of the distribution) differ from those of the approximation.

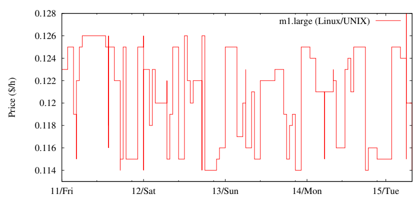
In light of the above, we propose the following algorithm:

1) Collect the prices over a period of time, in order to estimate their mean and variance.
2) Use the normal approximation described above, *i.e.*, assume that spot prices are normally distributed with the same mean and variance.
3) If $ACF(l) > 0.4$, then try to predict the prices for the next l hours using linear regression [14], $y = a + bx$.
4) If $ACF(l) \leq 0.4$ use the quantile function (inverse cumulative distribution function) of the Normal distribution to predict the prices for the next l hours. The quantile function returns the value of x such that $P(X \leq x)$ with the desired probability p.
5) Use the maximum value returned at point 3 (or 4) as a bid price.
6) If the bid price is smaller than the spot price, thus causing a failure (*out-of-bid* event [5]), increase the bid by 40% for the next interval.

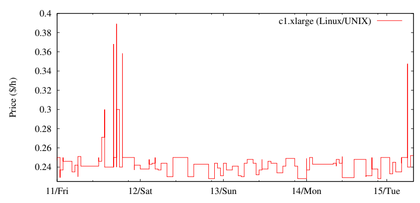
In Table II we report the performance of the prediction algorithm for different instance types, prediction horizons and availability targets over the two months period under consideration (the first 48 intervals were employed to train the predictive scheme). The algorithm performs well, especially for predicting the price of small and large instances, while the



(a) Small

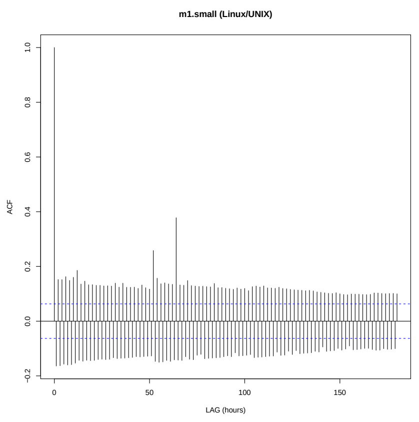


(b) Large

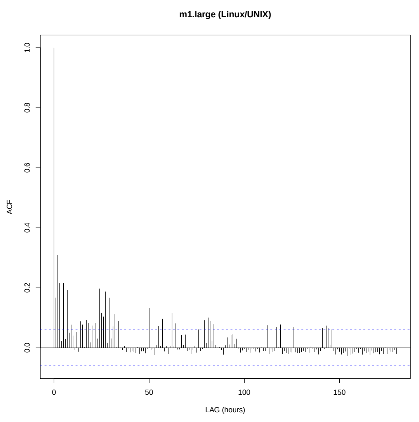


(c) Extra Large

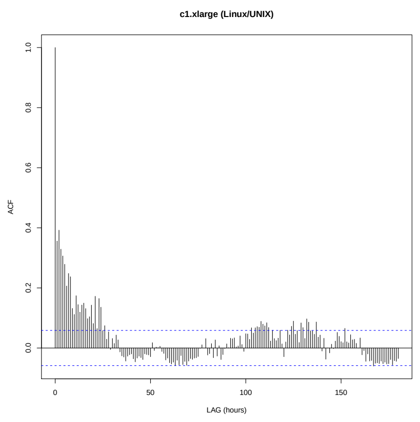
Fig. 1. Price history for (a) small and (b) large and (c) extra large Linux instances during the period 11 to 15 February, 2011 (*us-east-1* region).



(a) Small



(b) Large



(c) Extra Large

Fig. 2. Autocorrelation function for the interval December 25, 2010 – February 23, 2011 of the spot price for (a) small and (b) large and (c) extra large Linux instances (*us-east-1* region).

	m1.small (Linux/Unix)		m1.large (Linux/Unix)		c1.xlarge (Linux/Unix)	
	Data	Normal Approx.	Data	Normal Approx.	Data	Normal Approx.
Min.	0.029	0.02098	0.1140	0.05558	0.228	0.0571
1st Quartile	0.0290	0.02877	0.1170	0.10780	0.234	0.2274
Median	0.031	0.03020	0.1210	0.12190	0.243	0.2572
3rd Quartile	0.031	0.03163	0.1240	0.13550	0.249	0.2984
Max.	0.085	0.04010	0.3400	0.20670	0.800	0.4059
Mean	0.0302	0.03025	0.1226	0.12180	0.256	0.2585
Variance	4.503941×10^{-6}	4.515323×10^{-6}	4.469088×10^{-4}	4.611706×10^{-4}	4.55561×10^{-3}	3.647205×10^{-3}
Skewness	1.877202×10	1.659259×10^{-2}	9.256405	-7.901398×10^{-3}	5.323107	-0.1564130
Kurtosis	4.700735×10^{2}	3.004374	9.127482×10	3.218801	3.370251×10	3.567116

TABLE I
CHARACTERISTICS OF THE SPOT PRICE DISTRIBUTION FOR DIFFERENT INSTANCE TYPES (*us-east-1* REGION).

monetary saving compared to the scenario where "on demand" instances are employed[2] is evident. According to Table II, the highest bid for spot instances ($/hour) is:

- 0.03463, achieving 99.673% uptime, compared to 0.085 for "on demand" instances (small instances).
- 0.21196, achieving 98.928 uptime, compared to 0.34 for "on demand" instances (large instances).
- 0.51995, achieving 97.858% uptime, compared to 0.68 for "on demand" instances (extra large instances).

It is worth noting that the "best" price to bid heavily depends on the desired level of availability. The bid price can be reduced significantly by accepting a slightly smaller availability level. This confirms the findings reported in [5].

III. Model and SLA

The provider has a cluster of S processors/cores or virtual machines (*servers*, from now on) which are used to provide a service to paying customers. In this study we assume that the available resources are homogeneous: if that is not the case, a simple normalization function such as that suggested in [17] should be employed. Also, we assume that a maximum of m web service transactions (herewith called *jobs*) can be processed in parallel on each server without significant interference. Such a limit is dictated by the number of available threads or processes[3]. We model this by assuming that there m parallel servers on each machine/core. Consequently, there are $n = Sm$ total servers available. A job finding all n servers busy is temporarily parked in a FIFO queue, which is shared between all servers [18], [3]. As soon as a job completes its service and leaves the system, the server which executed it retrieves and runs another job from the head of the queue, if any, or begins to idle if the queue is empty.

Service Level Agreements (SLAs) can express the *Quality of Service* (QoS) requirements in different terms such as the average (or the percentile) of the response time, the

[2]"On demand" instances guarantee 99.95% availability, see http://aws.amazon.com/ec2-sla/.

[3]http://httpd.apache.org/docs/2.0/mod/prefork.html

m1.small (Linux/Unix)

Prediction	Target	Achieved	Avg. bid ($)
6	0.9	0.99673	0.03170
6	0.95	0.99673	0.03212
6	0.99	0.99673	0.03270
6	0.999	0.99673	0.03355
6	0.99999	0.99673	0.03463
12	0.9	0.99675	0.03207
12	0.95	0.99675	0.03250
12	0.99	0.99675	0.03307
12	0.999	0.99675	0.03385
12	0.99999	0.99675	0.03499
24	0.9	0.99679	0.03253
24	0.95	0.99679	0.03266
24	0.99	0.99679	0.03350
24	0.999	0.99679	0.03411
24	0.99999	0.99679	0.03533

m1.large (Linux/Unix)

Prediction	Target	Achieved	Avg. bid ($)
6	0.9	0.98051	0.15215
6	0.95	0.98051	0.15949
6	0.99	0.98928	0.17273
6	0.999	0.98928	0.18817
6	0.99999	0.98928	0.21196
12	0.9	0.98062	0.13293
12	0.95	0.98062	0.13476
12	0.99	0.98837	0.13683
12	0.999	0.98837	0.14079
12	0.99999	0.98934	0.14592
24	0.9	0.97868	0.13420
24	0.95	0.97868	0.13582
24	0.99	0.98837	0.13751
24	0.999	0.98837	0.14121
24	0.99999	0.98837	0.14642

c1.xlarge (Linux/Unix)

Prediction	Target	Achieved	Avg. bid ($)
6	0.9	0.94599	0.35023
6	0.95	0.95810	0.36785
6	0.99	0.96741	0.40748
6	0.999	0.97392	0.45258
6	0.99999	0.97858	0.51995
12	0.9	0.93240	0.31604
12	0.95	0.94444	0.32256
12	0.99	0.95000	0.34232
12	0.999	0.95370	0.36329
12	0.99999	0.96018	0.39443
24	0.9	0.93425	0.31441
24	0.95	0.93796	0.32034
24	0.99	0.94166	0.33544
24	0.999	0.94907	0.34726
24	0.99999	0.95555	0.36986

TABLE II
PERFORMANCE OF THE PRICE PREDICTION ALGORITHM FOR DIFFERENT PREDICTION HORIZONS AND AVAILABILITY TARGETS. THE AVERAGE PRICE/HOUR IN THE PERIOD DECEMBER 25, 2011 – FEBRUARY 23, 2011 WAS 0.03020 $ FOR M1.SMALL INSTANCES, 0.12284 $ FOR M1.LARGE INSTANCES AND 0.25620 $ FOR C1.XLARGE INSTANCES.

throughput, or the availability (see, for example, [19], [20]). In this paper we are concerned with guaranteeing the average response time, β, over intervals of a given length t (*e.g.*, t = one minute), where the response time of a job is defined as the interval between the job's arrival and its completion. Thus:

$$\beta = \frac{1}{t\gamma} \sum_{i=1}^{t\gamma} \delta_i, \tag{2}$$

where δ_i is the response time of the i-th job in the interval, and γ is average number of jobs accepted into the system per unit time .

The contract regulating the provisioning contract states, among the other *Service Level Objectives* (SLOs), that for each accepted and completed job the user pays a charge of c $. On the other hand, the provider must fulfill the following:

1) **Performance**: the average response time, β, over an agreed interval of length t should not exceed a certain threshold q. If the average response time exceeds the threshold, the provider is liable to pay a penalty of r_1 $ for each job executed in that interval.
2) **Availability**: the provider is liable to pay a penalty of r_2 $ for every job which is rejected due to resources unavailability.

Also, the system is subject to disastrous failures at which times all customers in the system (both waiting and being served) as well as arrivals occurring while the system undergoes repair are lost:

3) **Disaster**: the provider is liable to pay a penalty of r_3 $ for every job which, after being accepted, is lost due to resources becoming unavailable.

Given that renting servers from the Cloud costs r_4 $/h, the service provider tries to optimize is profits by means of a *resource allocation* policy which controls how many servers to run. The extreme value $n = 0$ correspond to rent 0 servers, thus denying the service to all potential customers. Hence, as far the provider is concerned, the performance of the system is measured by the average revenue, R, earned per unit time. That value can be estimated as

$$R = \gamma\left[c - r_1 P(\beta > q)\right] - r_2(\lambda - \gamma) - r_3 P(l_p < s_p)L - r_4 n, \tag{3}$$

where $P(\beta > q)$ is the probability that the average response time during the interval under consideration exceeds the predetermined threshold, L is the average number of jobs inside the system, and $P(l_p < s_p)$ represents the probability that the spot price exceeds the limit price, thus causing a disaster.

N.B. The relative magnitude of charge and penalties is irrelevant to the model.

IV. POLICIES

QoS can be guaranteed by means of different techniques (see [21] and the references cited within). In this paper, we adopt a general approach based on two complementary policies. The first one is a resource allocation policy that, based on traffic estimates, determines the number of servers to rent. This policy is invoked periodically. The intervals between consecutive allocation policy invocations are called *observation windows* or *epochs*. During an epoch, the number of running servers is constant, unless the acquired servers are terminated due to the clearing price crossing above the provider's limit price (cf. Section II). Also, during an epoch, the revenue optimization system collects traffic and service statistics (*e.g.*, mean arrival rate and service time), which are used by the allocation policy at the start of the next epoch.

The second policy is an *admission control* policy, *i.e.*, a mechanism which may deliberately decide to reject some jobs. The admission policy is defined by means of a threshold K (the *buffer size*). Incoming jobs are rejected if K other requests are in the system, without influencing future arrivals. The extreme values $K = 0$ and $K = \infty$ correspond to accepting none or all jobs. The problem of determining the optimal K in multi-server systems with finite waiting room has a long history. However the simultaneous optimization of K and of the number of servers (n) is known to be a significantly more complex problem [22]. Therefore, we approach this optimization problem by first choosing the number of servers (n), and then choosing an appropriate K for the chosen number of servers. Unfortunately, the optimal value of K depends not only on the number of available servers and average load, but also on the distribution of the load (service times and interarrival intervals). While some approximations exist (*e.g.*, [23]) these approximations either suffer from high computational complexity or they are only applicable in very restricted cases. Hence, we make simplifying assumptions about the arrival and service time processes. Specifically, we assume that jobs enter the system according to an independent Poisson process with rate λ while service times are exponentially distributed with mean b, and make the "best" possible choice under these assumptions. In other words, for a certain value K, the system behaves like a multi-server queueing system with n servers and a queue whose maximum size is $(K-n)$, *i.e.*, an $M/M/n/K$ queue. The admission policy might be sub-optimal if these simplifying assumptions are violated, but even in such cases it is likely to lead to reasonable performance levels.

Denote by p_j the stationary probability that there are j jobs in the $M/M/n/K$ queue. We observe that the average number of jobs being accepted into the system per unit time is

$$\gamma = \lambda(1 - p_k), \tag{4}$$

where p_k is the probability that the queue is full. The stationary distribution of the number of jobs into the system may be found by solving the balance equations

$$p_j = \begin{cases} \dfrac{(n\rho)^j}{j!} p_0 & j = 1, \dots, n-1 \\ \dfrac{(n\rho)^n}{n!} \rho^{j-n} p_0 & j = n, \dots, K \end{cases}, \tag{5}$$

where $\rho = \lambda b$ is the offered load. Steady state for this Birth-and-Death process always exists, as the queue is not allowed to grow unbound. In other words $\sum_{j=0}^{K} p_j = 1$.

Having computed the stationary distribution of the number of jobs present, by means of Little's law it is easy to estimate the average response time, W. That value is given by the relation $W = L/\gamma$, where γ is the effective arrival rate, see Equation (4), while L is the average number of jobs present (both waiting and being served)

$$L = \sum_{j=0}^{K} j p_j. \tag{6}$$

It is perhaps worth stressing that, even though the problem we are tackling here looks similar to that discussed in [8], the two are actually very different. In fact, while both are trying to optimize the average response time, in this paper we have constraints as well (costs and penalties) which make the problem much more challenging. Therefore, the search for the minimum amount of servers capable of satisfying $(\beta < q)$ would not optimize Equation (3). Similarly, Xiong and Perros [9] approximate a multi-server system by means of an $M/M/1$ queue, for which all the performance measures (including the distribution of the response times) are well known and easy to derive. On the other hand, in multi-server queueing systems with limited waiting room, the distribution of the response times is not exponential, in spite of making Markovian assumptions [3]. However, we can exploit the fact that the observed average response time over an interval, which according to Equation (2) involves the sum of $(t\gamma)$ response times, can be treated as being approximately normally distributed with mean W and variance $VAR(W)/(t\gamma)$, with $VAR(W)$ being the variance of the response times. That approximation appeals to the central limit theorem and ignores the dependencies between individual response times.

Based on the normal approximation, the probability that the observed average response time exceeds a given value, q, can be estimated as

$$P(\beta > q) = 1 - \Phi\left(\frac{q - W}{\sqrt{VAR(W)/(t\gamma)}}\right), \tag{7}$$

where $\Phi(x) = P(N(0,1) \leq x)$ is the cumulative distribution function (CDF) of the standard normal distribution having mean 0 and variance 1. That function can be computed very accurately by means of a rational approximation (see [24]).

In order for the approximation described in Equation (7) to perform well we require a large number of arrivals for each interval, *e.g.*, $t \gg \lambda$ and $t \gg b$. That condition also guarantees that any dependency between individual response times can be neglected. Finally, instead of estimating the variance of the response times at runtime, we introduce a further approximation by exploiting the observation that quite often waiting times are negligible compared to service times. Hence the variance of the response times can be approximated as the variance of the service times, $VAR(W) \approx VAR(b)$. Several numerical experiments confirmed that the choice of n nor that of K are affected when this approximation is employed. The above expressions, together with Equation (3), enable the average revenue R to be computed efficiently and quickly. Having fixed n, R becomes a unimodal function of K, *i.e.*, it has a single maximum, which may be at $K = \infty$ for over-provisioned systems. We do not have a mathematical proof of this statement, but have verified it in several numerical experiments. The above observation implies that the following Hill Climbing heuristics can be employed:

1) Choose an initial value of n; a good candidate is $n = \lceil \rho \rceil$ [3]. Set K to $n+1$.
2) At each iteration, try to increase the revenue by increasing the value of K by 1.
3) If the increase of K does not lead to a revenue increase, increase n by 1 and go to 2.
4) Stop when the revenue stops increasing, or when the revenue increase is smaller than a fixed value ϵ.

Note that when n is increased in Step 3, K is not set back to n (as it happens in Step 1). The reason is that it is unlikely that a higher revenue is produced by increasing n and decreasing K. Hence, a bulk of candidate solutions are discarded every time step 3 is executed. Also, due to the plurality of cost terms in Equation (3) the above algorithm is not guaranteed to find the global maximum. Possible improvements include using a tabu-list or trying to "jump" plateaus by using random restarts (Stochastic Hill Climbing). However, those techniques are not guaranteed to converge to a global maximum either. On the other hand we have experimentally verified that the local maxima found by the above Hill Climbing heuristics are sufficient to achieve good revenue levels (*i.e.*, the local maxima are global maxima, or very close to it).

A. *Simpler Heuristic*

The policy we have described in the previous section is rather complicated, as it requires not only the optimal number of servers n to perform well, but also the optimal threshold, K (see Figure 3(a)). It may therefore be desirable to design a heuristic that is simpler but performs reasonably well.

The model we will introduce in this section is based on the observation that in multi-server queueing systems the response time is often dominated by the service time, while the waiting time decreases if the load is scaled up with n, *e.g.*, a 20 servers system with a normalized load of 0.90 is less congested than a 2 servers system with a normalized load of 0.70 [25]. Hence, a simple approximation is to assume that the buffer size is unbounded, thus allowing all the traffic into the system. In other words Equation (3) simplifies to

$$R = \lambda\left[c - r_1 P(\beta > q)\right] - r_3 P(l_p < s_p)L - r_4 n, \qquad (8)$$

under the assumption that the system is stable, *i.e.*, $\rho < n$. If that is the case the throughput is equal to the arrival rate, otherwise the system is unstable, *e.g.* the queue grows unbound. For the following, we will relax the assumptions about the nature of the user demand, thus allowing for general distributions of inter-arrival intervals as well as service times. Since the heuristic policy does not reject any job we can model the system as an $GI/G/n$ queue in steady-state, for which there is no exact solution (see, for example, [25]). In order to estimated the average response time we are not required to compute the stationary distribution of the number of jobs present as in Section IV. Instead, we employ the two-moments approximation proposed by Allen and Cunneen [26]

$$W \approx b + \left[P(j \geq n) \cdot \frac{b}{n-\rho} \cdot \left(\frac{{c_a}^2 + {c_s}^2}{2}\right)\right], \qquad (9)$$

where $P(j \geq n)$ is the probability that an incoming job will have to wait (Erlang-C formula), while ${c_a}^2$ and ${c_2}^2$ are the squared coefficients of variation (the variance divided by the square of the mean) of the inter-arrival intervals and services times respectively. The Erlang-C formula includes factorial and power elements, hence its evaluation requires special care when n and ρ are large. However Halfin and Whitt [27] derived a closed-form approximation which performs well

$$P(j \geq n) \approx [1 + \sqrt{2\pi}\eta\Phi(\eta)\ e^{(\eta^2/2)}]^{-1}, \qquad (10)$$

where $\eta = (1 - \rho/n)\sqrt{n}$ and $\Phi(\cdot)$ is the CDF of the standard normal distribution (see also Equation (7)).

A search algorithm similar to that introduced in Section IV can be employed for finding the values of n that optimizes Equation (8).

V. Performance Evaluation

We carried out experiments with the aim of evaluating how the price prediction model and the policies proposed in Sections II and IV affect the maximum achievable revenues. Since the number of variables is high, we adopted default values for some of them (cf. Table III).

b	0.5 sec.	Service time
q	1 sec.	Performance threshold
c	2.951×10^{-5} \$	Charge per job
r_1	$1.5 \times c$	Penalty for performance
r_2	$2 \times c$	Penalty for availability
r_3	$3 \times c$	Penalty for disaster
r_4	0.085 \$/h	Rental cost
t	1 min.	Interval length (SLA)

TABLE III
SETTINGS.

The first experiment examines via numerical simulation the extent to which the use of a sensible admission threshold can improve revenues, for five different allocations. Figure 3(a) shows that (a) in each case there is an optimal queue length, (b) the heavier the load, the more important is to operate at, or very close to, the optimum threshold, and (c) due to the several cost factors in Equation (3), it is not always true that the queue threshold should decrease in response to a load increase. Thus, when $n = 35$ (*i.e.*, the normalized load is 0.857) the queue threshold producing the best revenue (3.4 \$/h) is $K = 188$. However the same revenue is obtained for $K = \infty$, and an almost equal revenue is obtained for $K = 100$. If n is reduced to 31, resulting in a normalized load of 0.967, the optimal revenue (3.74 \$/h) is achieved when $K = 500$ (not shown in order not to clutter the graphs). However, choosing $K = 170$ or $K = \infty$ would not make a lot of difference. When $n = 30$ (saturated system) the best revenue (3.54 \$/h) is produced by setting $K = 91$, but it drops very sharply if K exceeds the optimal value (R is a discontinuous function). For example the revenue drops to 2.12 \$/h when $K = 94$, while if $K = 95$ the system produces a revenue of -1 \$/h, or -4.24 \$/h when $K = 96$, hence losing money. That behavior is further emphasized when the system becomes over-loaded (*e.g.*, smaller amount of servers). Figures 3(b) shows a similar experiment, with the difference that now the disaster probability is 0.25. As a result, not only the maximum achievable revenues are lower (*e.g.*, 0.873 \$/h when $n = 35$), but the admission thresholds producing the highest possible revenues are also smaller (*e.g.*, 71 instead of 188 when $n = 35$, or 28 instead of 47 when $n = 23$). If the probability of disaster is further increased to 0.5, the best thresholds get even smaller. However the situation worsens to the point that no solution is capable of producing a positive revenue (see Figure 3(c)).

The next experiment aims at measuring to which extent the value of the penalties affect the system's behavior. Hence we vary the values of the penalty for QoS violation, r_1, and the penalty for rejecting jobs, r_2 for a fixed load, $\rho = 10$, and number of servers, $n = 10$. Servers are "on demand" instances, hence they never fail. The system is saturated on purpose so that the policy has to carefully choose between a long queue, hence increasing the chance of failing to meet agreed

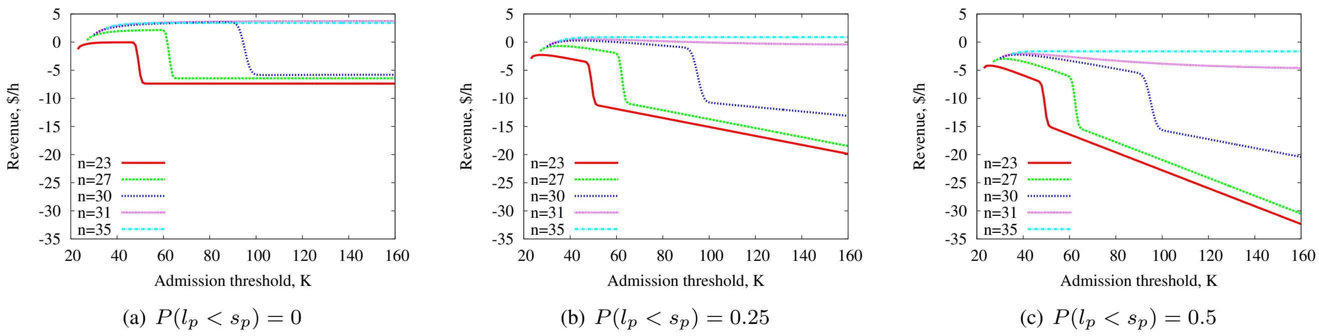


Fig. 3. Revenue as a function of the admission threshold, K, and number of servers, n. $\lambda = 300$, $r_4 = 0.085$ \$/h. Other settings as in Table III.

performance objectives, and a short queue, thus rejecting many jobs. When the penalty for failing do deliver performance, r_1, is 0, the best revenue is achieved when accepting all jobs, *i.e.*, by setting $K = \infty$. In the remaining five scenarios the best admission threshold is always $K = 30$, however the produced revenues differ, see Figure 4(a).

In the following set of experiments we evaluate by means of discrete event simulation how the proposed policies perform under different loading conditions, traffic characteristics and instance type ("on demand" and "spot" instances of type Linux/Unix *m1.small*). The load varies between 7.5 and 25 by increasing the arrival rate from 15 to 50 requests/second. The average revenues obtained per hour are plotted against the arrival rate. Each point in the figure represents one run lasting 28 days, and include the corresponding 95% confidence interval. That value is calculated using the Student's t-distribution. Allocation and admission decisions are taken every six hours. In other words in the scenarios where the SaaS employs "spot instances", they are replaced every six hours. Spot prices vary every hour as in Figure 1(a), while the algorithm used to predict the prices (see Section II) limits the probability of disaster to 0.001. Finally, servers' bootstrap as well as recovery from disasters take 5 minutes. For all the policies, at the start of each run, before any statistic has been collected, servers are allocated in the measure of 2ρ, while the admission threshold is set to $K = \infty$ (*e.g.*, all jobs are accepted).

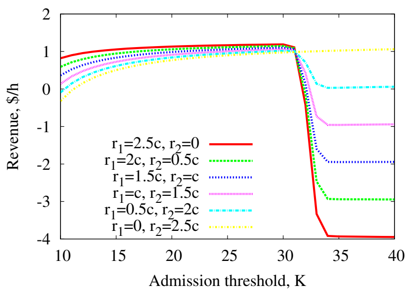
Apart from the revenues increase as a consequence of the load increase, the most notable feature of the graph plotted in Figure 4(b) is that the system is much more profitable when using "spot instances" compared to the case where "on demand" instances are employed, hence showing that the price prediction scheme introduced in Section II performs well (please note that in case of disaster the provider should pay a penalty of $3cL$ \$ due to the jobs in the system being lost plus $600c\lambda$ \$ due to the penalties arising from rejecting all the jobs during the recovery). Another interesting finding is that the heuristic policy performs very close to the algorithm using admission thresholds, with most of the points being within each other's confidence intervals.

In the next experiment we depart from the assumption that the traffic is Markovian in order to evaluate the effects of inter-arrival and service time variability on performance. The average values are kept the same as before. However now both the inter-arrival intervals and service times are generated according to a Log-Normal distribution. The corresponding squared coefficient of variation are $c_a^2 = 2$ and $c_s^2 = 5$. It is legitimate to expect the performance to deteriorate when the traffic variability increases, since the system becomes less predictable, and therefore it becomes more difficult to choose the best n and K (for the threshold policy). Figure 4(c) shows that the revenues achieved when the Markovian assumptions are violated are slightly lower. However the difference is not very large. It is perhaps worth mentioning that the relative insensitivity of the system performance with respect to the nature of the user demand is, from a practical point of view, a very good feature. As in the previous experiment, the difference between using "on demand' and "spot" instances is evident, while the threshold and heuristic policies perform almost the same. This is probably due to the fact that the heuristic policy uses approximate algorithms capable of dealing with any kind of traffic.

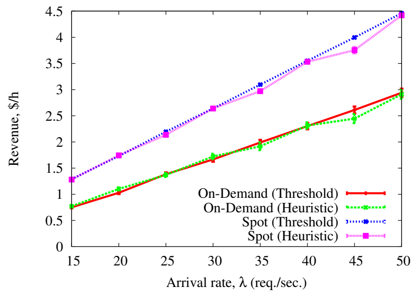
VI. Conclusions

We presented an approach to maximize the net revenue per time unit earned by a SaaS provider who hosts a paid web service using spot instances. Since the revenue is affected by the number of successfully served customers, paid penalties and server rental cost, the proposed approach aims at maximizing the first while minimizing the last two factors by deciding: (a) how much to bid for resources on the spot market?, and (b) how many servers to allocate for a given time period, and how many jobs to accept? To address the first question, we outlined an algorithm to determine the optimal (truthful) limit price to bid on the spot market to achieve a desired availability level. The second question is addressed by means of dynamic policies for server allocation and admission control.

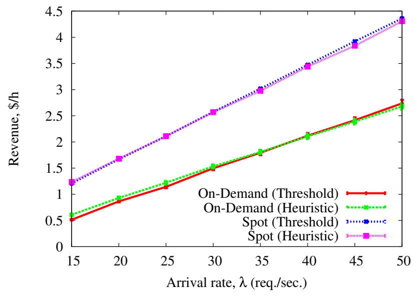
The experimental evaluation put into evidence the trade-offs involved in this revenue optimization problem. In particular, the optimal queue length (maximum number of jobs admitted before rejecting further jobs) is highly dependent on the availability level. When the likelihood of premature termination of the running instances is low, the threshold can be increased. Vice versa, if the chance of early instance termination is high, low thresholds are better. The experiments also showed that the proposed policies are robust against traffic variability.



(a) $n = 10,\ \rho = 10,\ P(l_p < s_p) = 0$



(b) $c_s^2 = 1,\ c_a^2 = 1,\ P(l_p < s_p) = 0.001$



(c) $c_s^2 = 5,\ c_a^2 = 2,\ P(l_p < s_p) = 0.001$

Fig. 4. (a) Revenue as function of r_1, r_2 and admission threshold, K, and (b, c) revenue as a function of the arrival rate, λ. "On demand" instances cost 0.085 $/h, while "spot instances" are rented on the spot market. The bid price is determined using the algorithm proposed in Section II. Other settings as in Table III.

A possible direction for future research is to consider Service Level Objectives (SLOs) expressed in terms of "percentile of requests served below a given response time", as opposed to SLOs expressed in terms of averages over a certain interval.

Acknowledgements

The authors would like to thank Dmytro Dyachuk, Jim MacGregor Smith and Peeter Oja for the useful comments and discussions. This work was partly funded by the EU Regional Development Funds under EUREKA Project 4989 (SITIO).

References

[1] G. Singer, I. Livenson, M. Dumas, S. N. Srirama, and U. Norbisrath, "Towards a model for cloud computing cost estimation with reserved resources," in *Proceedings of 2nd ICST International Conference on Cloud Computing (CloudComp 2010), Barcelona, Spain.* Springer, October 2010.

[2] M. Mazzucco and M. Dumas, "Reserved or On-Demand Instances? A Revenue Maximization Model for Cloud Providers," in *Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD 2011).* IEEE, July 2011. [Online]. Available: http://math.ut.ee/~mazzucco/papers/cloud_2011.pdf

[3] M. Mazzucco, I. Mitrani, J. Palmer, M. Fisher, and P. McKee, "Web Service Hosting and Revenue Maximization," in *Proceedings of the Fifth European Conference on Web Services (ECOWS'07).* IEEE Computer Society, November 2007, pp. 45–54.

[4] M. Mazzucco, I. Mitrani, M. Fisher, and P. McKee, "Allocation and Admission Policies for Service Streams," in *In Proocedings of 16th IEEE MASCOTS*, September 2008, pp. 155–162.

[5] A. Andrzejak, D. Kondo, and S. Yi, "Decision model for cloud computing under sla constraints," in *Proceedings of the 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2010).* IEEE, August 2010, pp. 257–266.

[6] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues, "Conductor: orchestrating the clouds," in *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS '10).* ACM, 2010, pp. 44–48.

[7] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu, "Resource provisioning for cloud computing," in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '09).* ACM, 2009, pp. 101–111.

[8] F. Ahmad and T. N. Vijaykumar, "Joint optimization of idle and cooling power in data centers while maintaining response time," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS '10).* ACM, 2010, pp. 243–256.

[9] K. Xiong and H. Perros, "Resource Optimization Subject to a Percentile Response Time SLA for Enterprise Computing," in *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM '06)*, December 2006, pp. 1 –6.

[10] Y.-J. Hong, M. Thottethodi, and J. Xue, "Dynamic Server Provisioning to Minimize Cost in an IaaS Cloud," School of Electrical and Computer Engineering, Purdue University, Tech. Rep. 414, March 2011. [Online]. Available: http://docs.lib.purdue.edu/ecetr/414/

[11] M. Stokely, J. Winget, E. Keyes, C. Grimes, and B. Yolken, "Using a market economy to provision compute resources across planet-wide clusters," in *Proceedings for the International Parallel and Distributed Processing Symposium (IPDPS 2009).* IEEE, 2009.

[12] M. Mattess, C. Vecchiola, and R. Buyya, "Managing peak loads by leasing cloud infrastructure services from a spot market," in *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC 2010).* IEEE, 2010, pp. 180–188.

[13] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz, "See spot run: using spot instances for mapreduce workflows," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10).* USENIX Association, 2010, pp. 7–7.

[14] C. Chatfield, *The Analysis of Time Series – An Introduction*, 6th ed. Chapman & Hall/Crc, 2004.

[15] M. Jofre-Bonet and M. Pesendorfer, "Bidding behavior in a repeated procurement auction: A summary," *European Economic Review*, vol. 44, no. 4-6, pp. 1006 – 1020, 2000.

[16] L. Brannman, J. Klein Douglass, and L. W. Weiss, "The Price Effects of Increased Competition in Auction Markets," *The Review of Economics and Statistics*, vol. 69, no. 1, February 1987.

[17] V. Rykov and D. Efrosinin, "Optimal control of queueing systems with heterogeneous servers," *Queueing Systems*, vol. 46, no. 3-4, pp. 389–407, 2004.

[18] D. Menascé, "Tradeoffs in Designing Web Clusters," *IEEE Internet Computing*, vol. 6, pp. 76–80, September 2002.

[19] E. Wustenhoff, "Service Level Agreement in the Data Center," April 2002. [Online]. Available: http://www.met.reading.ac.uk/~swsellis/tech/solaris/performance/doc/blueprints/0402/sla.pdf

[20] D. A. Menascé, "QoS Issues in Web Services," *IEEE Internet Computing*, vol. 6, pp. 72–75, November 2002.

[21] M. Mazzucco, "Revenue maximization problems in commercial data centers," Ph.D. dissertation, Newcastle University, 2009. [Online]. Available: http://hdl.handle.net/10443/457

[22] J. MacGregor Smith, "Multi-server, Finite Waiting Room, M/G/c/K Optimization Models," *INFOR: Information Systems and Operational Research*, vol. 45, 2007.

[23] W. Whitt, "A Diffusion Approximation for the G/GI/n/m Queue," *Operations Research*, vol. 52, pp. 922–941, November 2004.

[24] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover Publications, 1972.

[25] W. Whitt, "Approximations for the GI/G/m Queue," *Production and Operations Management*, vol. 2, no. 2, pp. 114–161, 1993.

[26] A. Allen, "Queueing models of computer systems," *Computer*, vol. 13, pp. 13–24, 1980.

[27] S. Halfin and W. Whitt, "Heavy-Traffic Limits for Queues with Many Exponential Servers," *Operations Research*, vol. 29, May-June 1981.