Web Service Hosting and Revenue Maximization

M. Mazzucco, I. Mitrani, J. Palmer, M. Fisher, P. McKee.

TECHNICAL REPORT SERIES

No. CS-TR-1047 September, 2007

No. CS-TR-1047 September, 2007

Web Service Hosting and Revenue Maximization

Michele Mazzucco, Isi Mitrani, Jennie Palmer, Mike Fisher, Paul McKee.

Abstract

An architecture of a hosting system is presented, where a number of servers are used to provide different types of web services to paying customers. There are charges for running jobs and penalties for failing to meet agreed Quality-of-Service requirements. The objective is to maximize the total average revenue per unit time. Dynamic policies for making server allocation and job admission decisions are introduced and evaluated. The results of several experiments with a real implementation of the architecture are described.

Bibliographical details

MAZZUCCO, M., MITRANI, I., PALMER, J., FISHER, M., MCKEE, P.

Web Service Hosting and Revenue Maximization
[By] M. Mazzucco, I. Mitrani, J. Palmer, M. Fisher, P. McKee.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2007.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1047)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1047

Abstract

An architecture of a hosting system is presented, where a number of servers are used to provide different types of web services to paying customers. There are charges for running jobs and penalties for failing to meet agreed Quality-of-Service requirements. The objective is to maximize the total average revenue per unit time. Dynamic policies for making server allocation and job admission decisions are introduced and evaluated. The results of several experiments with a real implementation of the architecture are described.

About the author

Michele Mazzucco is a Research Associate and PhD student at the School of Computing Science, Newcastle University.

Isi Mitrani studied Mathematics at the Universities of Sofia and Moscow (Diploma, 1965), and Operations Research at the Technion, Haifa (MSc, 1967). He joined the University of Newcastle in 1968 as a Programming Advisor, became a Lecturer in 1969 (PhD, 1973), Reader in 1986 and Professor in 1991. He has held several visiting positions, including sabbatical years at INRIA (Le Chesnay) and Bell Laboratories (Murray Hill). His research interests are in the areas of Probabilistic Modelling, Performance Evaluation and Optimization. Publications include 4 authored books, 3 edited books, more than 100 journal and conference papers and one patent.

Jennie Palmer studied Mathematics at the University of Cambridge (BA, 1992) and continued her studies at Cambridge in Mathematics Education (PGCE, 1993). After lecturing in Mathematics in Further Education from 1993-2000, she studied for an MSc and a PhD in Computing Science at Newcastle University, awarded in 2001 and 2006 respectively. Her research interests are in the areas of probabilistic modelling, performance evaluation and optimization.Jennie is currently a member of the Recruitment and Admissions Team within the School of Computing Science, with responsibility for School Liaison.

Suggested keywords

WEB SERVICE HOSTING,
QUALITY OF SERVICE,
REVENUE MAXIMIZATION,
SERVER ALLOCATION,
ADMISSION POLICIES,
M/M/N/K QUEUE

Web Service Hosting and Revenue Maximization

Michele Mazzucco, Isi Mitrani, Jennie Palmer
School of Computing Science, Newcastle University, NE1 7RU, UK

Mike Fisher, Paul McKee
BT Group, Adastral Park, Ipswich, IP5 3RE, UK

Abstract

An architecture of a hosting system is presented, where a number of servers are used to provide different types of web services to paying customers. There are charges for running jobs and penalties for failing to meet agreed Quality-of-Service requirements. The objective is to maximize the total average revenue per unit time. Dynamic policies for making server allocation and job admission decisions are introduced and evaluated. The results of several experiments with a real implementation of the architecture are described.

Keywords: Web Service Hosting, Quality of service, Revenue maximization, Server allocation, Admission policies, $M/M/N/K$ queue.

1 INTRODUCTION

This work deals with the topic of web service hosting in a commercial environment. A service provider employs a cluster of servers in order to offer a number of different services to a community of users. The users pay for having their jobs run, but demand in turn a certain quality of service. We assume that the latter is expressed in terms of either waiting time or response time, although other metrics might also be adopted. Thus, with each service type is associated a *Service Level Agreement* (SLA), formalizing the obligations of the users and the provider. In particular, a user agrees to pay a specified amount for each accepted and completed job, while the provider agrees to pay a penalty whenever the response time (or waiting time) of a job exceeds a certain bound. It is then the provider's responsibility to decide how to allocate the available resources, and when to accept jobs, in order to make the system as profitable as possible. Efficient policies that avoid over-provisioning are clearly desirable. That, in general terms, is the problem that we wish to address.

Our approach has two strands. The first consists of designing and implementing a middleware platform for the deployment and use of web services. That system, called SPIRE (Service Provisioning Infrastructure for Revenue Enhancement), is a self-configurable and self-optimizing middleware platform [9] that dynamically migrates servers among hosted services and enforces SLAs by monitoring income and expenditure. It provides a test bed for experimentation with different operating policies and different patterns of demand.

The second strand involves quantitative modelling. Under suitable assumptions about the nature of user demand, it is possible to evaluate explicitly the effect of particular server allocation and admission policies. Hence, we derive a numerical algorithm for computing the optimal queue length thresholds corresponding to a given partition of the servers among the different service types. That computation is sufficiently fast to be performed on-line as part of a dynamic admission policy. The latter can be implemented in any system, but may of course lose its optimality if the simplifying assumptions are not satisfied.

Dynamic server allocation and job admission policies are evaluated and compared in the real hosting system. The performance measure used as a criterion of comparison is the average revenue earned per unit time, over the observation period of the experiment.

The revenue maximisation problem described here does not appear to have been studied before. Perhaps the most closely related work can be found in the papers by Villela *et al* [15] and Levy *et al* [10]. They consider a similar economic model in systems consisting of one or more shared servers. The emphasis is on allocating server capacity only; admission policies are not considered. Yet we shall see that revenues can be improved very significantly by imposing suitable conditions for accepting jobs.

Huberman *et al* [7] and Salle and Bartolini [13] present two different abstract frameworks involving pricing and contracts in IT systems. Those approaches do not take congestion into account explicitly and are therefore not readily applicable in our context.

Rajkumar *et al* [12] consider a resource allocation model for QoS management, where application needs may include timeliness, reliability, security and other application specific requirements. The model is described in terms of a *utility function* to be maximised. This model is extended in [4] and [6]. Such multi-dimensional QoS is beyond the scope of this paper. However, although those models allow for variation in job computation time and frequency of application requests, once again, congestion and response/waiting times are not considered.

Chandra *et al* [2], Kanodia and Knightly [8] and Bennani and Menascé [1] examine certain aspects of resource allocation and admission control in systems where the QoS criterion is related to response time. Those studies do not consider the economic issues related to income and expenditure.

The structure of SPIRE is described in Section 2. The service-level agreements, the mathematical model and the policies for server allocation and job admission are presented in Section 3. Some numerical results and a number of experiments carried out with SPIRE, where the policies are evaluated and compared under different loading conditions, are reported in Section 4. Section 5 contains a summary and conclusions.

2 System architecture

The SPIRE middleware platform was designed and implemented with the aim of providing a real-life environment in which to experiment with various hosting policies. The architecture, illustrated in Figure 1, is message-based and asynchronous. Moreover, it is *dedicated*, rather than *shared*; in other words, the servers in the cluster are dynamically grouped into pools, each pool dealing with demands for a particular service.

Self-management functions are provided by a `Controller` whose main components are a `Dispatcher`, a `Resource Allocator` and a set of `Service Handlers`, one for each offered service[1].

All client requests are sent to the Controller, which acts like a proxy (arrow 1), and are then forwarded to the appropriate Service Handler. The latter implements the admission control policy (notifying users of rejections, arrow 7), the scheduling policy (currently jobs are sent to servers in FIFO order, arrow 2), the collection of statistics through a profiler and the management of the corresponding pool of servers. The results of completed jobs are returned to the Controller, where statistics are collected, and to the relevant user (arrows 5 and 6).

[1]If the same service is offered at different QoS levels (*e.g.* gold, silver and bronze) there will be a different Service Handler for each level.

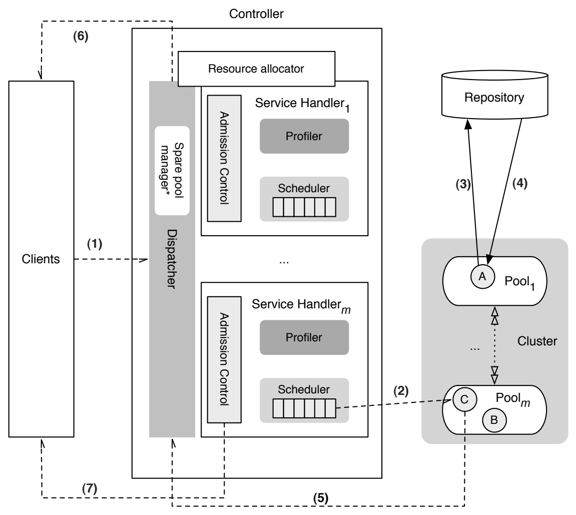

Figure 1. Structure of SPIRE. Dotted lines indicate asynchronous messages.

The dynamic server allocation policy is implemented by the Resource Allocator. This is an asynchronous task which executes every J arrivals, for some J. We decided to use events instead of a time interval because this provides SPIRE with *adaptive windows* [3]. Thus, the Resource Allocator executes more frequently when the system is heavily loaded than when it is lightly loaded. At each configuration point, the Resource Allocator uses the information collected during the current window (which includes statistics on arrivals, completions and queue sizes), to make decisions about server allocations and job admissions during the next window. The actual policies adopted will be described in the next section.

If a deployment is needed, the service is fetched from a (remote) repository (arrows 3 and 4). This incurs a migration time of a few seconds, while a server is switched between pools. SPIRE tries to avoid unnecessary deployments, while still allowing new services to be added at runtime. In such cases, the appropriate Service Handlers are automatically created in the Controller. If there is sufficient space on a server, deployed services could be left in place even when not currently offered on that server. One would eventually reach a situation where all services are deployed on all servers. Then, allocating a server to a particular service pool (handler) does not involve a new deployment; it just means that only jobs of that type would be sent to it. In those circumstances, switching a server from one pool to another does not incur any overhead. This is the case in the present implementation.

3 Contracts and policies

Suppose that the services being offered are numbered $1, 2, \ldots, m$. A request for service i is referred to as a 'job of type i'. We assume that the *quality of service* experienced by an accepted job of type i is measured either in terms of its response time, W (the interval between the job's arrival and completion), or in terms of its waiting time, w (excluding the service time). Whichever the chosen measure, it is mentioned explicitly in a service level agreement between the provider and the users. In particular, each such contract includes the following three clauses:

1. Charge: For each accepted and completed job of type i a user shall pay a charge of c_i (in practice this may be proportional to the average length of type i jobs).

2. Obligation: The response time, W_i (or waiting time, w_i), of an accepted job of type i shall not exceed q_i.

3. Penalty: For each accepted job of type i whose response time (or waiting time) exceeds q_i, the provider shall pay to the user a penalty of r_i.

Thus, service type i is characterized by its 'demand parameters' (these describe the arrival pattern and the run times of type i jobs), and its 'economic parameters', namely the triple

$$(c_i, q_i, r_i) = (charge, obligation, penalty) \tag{1}$$

Within the control of the host are the 'resource allocation' and 'job admission' policies. The first decides how to partition the total number of servers, N, among the m service pools. That is, it assigns n_i servers to jobs of type i ($n_1 + n_2 + \ldots + n_m = N$). The allocation policy may deliberately take the decision to deny service to one or more job types (this will happen if the number of services offered exceeds the number of servers).

The admission policy is defined by a set of thresholds, K_i ($i = 1, 2, \ldots, m$); incoming jobs of type i are rejected if there are K_i of them present in the system. The extreme values $K_i = 0$ and $K_i = \infty$ correspond to accepting none, or all, of the type i jobs.

Of course, both the server allocations n_i and the admission thresholds K_i may, and should, change dynamically in response to changes in demand. The problem is how to do this in a sensible manner.

As far as the host is concerned, the performance of the system is measured by the average revenue, V, obtained per unit time. This is evaluated according to

$$V = \sum_{i=1}^{m} \gamma_i [c_i - r_i P(W_i > q_i)] , \tag{2}$$

where γ_i is the average number of type i jobs *accepted* per unit time, and $P(W_i > q_i)$ is the probability that the response time of an accepted type i jobs exceeds the obligation q_i. If the service level agreements are in terms of waiting times rather than response times, then W_i should be replaced by w_i.

The objective of the resource allocation and job admission policies is to maximize the revenue V.

Note that, although we make no assumptions about the relative magnitudes of the charge and penalty parameters, the more interesting case is where the latter is at least as large as the former: $c_i \le r_i$. Otherwise one could guarantee a positive revenue by accepting all jobs, regardless of loads and obligations.

As well as considering optimal policies for allocating servers, we propose, and experiment with, the following simple heuristics.

1. *Measured Loads heuristic*. From the statistics collected during a window, estimate the arrival rate, λ_i, and average service time, $1/\mu_i$, for each job type. For the duration of the next window, allocate the servers roughly in proportion to the offered loads, $\rho_i = \lambda_i/\mu_i$, and to a set of coefficients, α_i, reflecting the economic importance of the different job types. In other words, start by setting

$$n_i = \left\lfloor N \frac{\rho_i \alpha_i}{\sum_{j=1}^{m} \rho_j \alpha_j} + 0.5 \right\rfloor \tag{3}$$

(adding 0.5 and truncating is the round-off operation). Then, if the sum of the resulting allocations is either less than N or greater than N, adjust the numbers so that they add up to N.

2. *Measured Queues heuristic*. From the statistics collected during a window, estimate the average queue sizes, L_i, for each job type. For the duration of the next window, allocate the servers roughly in proportion to those queue sizes and the coefficients α_i. That is, start by setting

$$n_i = \left\lfloor N \frac{L_i \alpha_i}{\sum_{j=1}^{m} L_j \alpha_j} + 0.5 \right\rfloor , \tag{4}$$

and then adjust the numbers so that they add up to N.

The intuition behind both of these heuristics is that more servers should be allocated to services which are (a) more heavily loaded and (b) more important economically. The difference is that the first estimates the offered loads directly, while the second does so indirectly, via the observed average queue sizes. One might also decide that it is undesirable to starve existing demand completely. Thus, if the heuristic suggests setting $n_i = 0$ for some i, but there have been arrivals of that type, an attempt is made to adjust the allocations and set $n_i = 1$. Of course, that is not always possible.

One could use different expressions for the coefficients α_i. Experiments have suggested that a good choice is to set $\alpha_i = r_i/c_i$ (the supporting intuition is that the higher the penalty, relative to the charge, the more important the job type becomes). If $r_i = c_i$, then $\alpha_i = 1$ and servers are allocated in proportion to offered loads (or queues).

An allocation policy decision to switch a server from one pool to another does not necessarily take effect immediately. If a service is in progress at the time, it is allowed to complete before the server is switched (i.e., switching is non-preemptive).

Under both allocation policies it may happen, either at the beginning of a window, or during a window, that the number of servers allocated to a pool is greater than the number of jobs of the corresponding type present in the system. In that case, one could decide to return the extra servers to a 'spare pool' and to assign them temporarily to other job types. If that is done, the heuristic will be called 'greedy'. The original policy, where no reallocation takes place until the end of the current window, will be called 'conservative'. Thus, we have conservative and greedy versions of both the Measured Loads heuristic and the Measured Queues heuristic.

3.1 Admission policies

Having decided how many servers to allocate to each pool, one should choose appropriate queue size thresholds for purposes of job admission. Unfortunately, the optimal threshold K_i depends not only on n_i and on the average interarrival and service times, but also on the distributions of those intervals. Moreover, those dependencies are very hard to evaluate, in general. Therefore, we make simplifying assumptions about the arrival and service processes, and choose the 'best' thresholds under those assumptions. Such choices may be sub-optimal when the assumptions are not satisfied, but should nevertheless lead to reasonable admission policies.

The simplification consists of assuming that jobs of type i arrive according to an independent Poisson process with rate λ_i, and their required service times are distributed exponentially with mean $1/\mu_i$. Thus, for any fixed admission threshold K_i, queue i is modelled as an $M/M/n_i/K_i$ queue in the steady state (see, for example, [11]).

Denote by $p_{i,j}$ the stationary probability that there are j jobs of type i in the $M/M/n_i/K_i$ queue, and by $W_{i,j}$ the response time of a type i job which finds, on arrival, j other type i jobs present. The average number of type i jobs accepted into the system per unit time, γ_i, is equal to

$$\gamma_i = \lambda_i(1 - p_{i,K_i}) \ . \tag{5}$$

Similarly, the probability $P(W_i > q_i)$ that the response time of an accepted type i jobs exceeds the obligation q_i, is given by

$$P(W_i > q_i) = \frac{1}{1 - p_{i,K_i}} \sum_{j=0}^{K_i - 1} p_{i,j} P(W_{i,j} > q_i) \ . \tag{6}$$

The average revenue, V_i, gained from type i jobs per unit time, is

$$V_i = \gamma_i [c_i - r_i P(W_i > q_i)] \ . \tag{7}$$

If the QoS measure is the waiting time rather than the response time, then $P(W_{i,j} > q_i)$ is replaced by $P(w_{i,j} > q_i)$ (where $w_{i,j}$ is the waiting time of a type i job which finds, on arrival, j other type i jobs present).

The stationary distribution of the number of type i jobs present may be found by solving the balance equations

$$p_{i,j} = \begin{cases} \rho_i p_{i,j-1}/j & \text{if } j \le n_i \\ \rho_i p_{i,j-1}/n_i & \text{if } j > n_i \end{cases} \ , \tag{8}$$

where $\rho_i = \lambda_i/\mu_i$ is the offered load for service type i. These equations are best solved iteratively, starting with $p_{i,0} = 1$, and then dividing each probability by their sum, in order to ensure that

$$\sum_{j=0}^{K_i} p_{i,j} = 1 \ . \tag{9}$$

The conditional response time distribution, $P(W_{i,j} > q_i)$, is obtained as follows.

Case 1: $j < n_i$. There is a free server when the job arrives, hence the response time is distributed exponentially with parameter μ_i.

$$P(W_{i,j} > q_i) = e^{-\mu_i q_i} \ . \tag{10}$$

Case 2: $j \ge n_i$. If all servers are busy on arrival, the job must wait for $j - n_i + 1$ departures to take place (these occur at exponentially distributed intervals with parameter $n_i\mu_i$), before starting its own service. The conditional response time is now distributed as the convolution of an Erlang distribution with parameters $(j - n_i + 1, n_i\mu_i)$, and an exponential distribution with parameter μ_i. The Erlang density function with parameters (k, a) has the form (see [11])

$$f_{k,a}(x) = \frac{a(ax)^{k-1}e^{-ax}}{(k-1)!} \ .$$

Hence, we can write

$$P(W_{i,j} > q_i) = \int_0^{q_i} \frac{n_i\mu_i(n_i\mu_i x)^{j-n_i}}{(j-n_i)!} e^{-n_i\mu_i x} e^{-\mu_i(q_i - x)} dx + \int_{q_i}^{\infty} \frac{n_i\mu_i(n_i\mu_i x)^{j-n_i}}{(j-n_i)!} e^{-n_i\mu_i x} dx \ . \tag{11}$$

After some manipulation, using existing expressions for the Gamma integral (e.g., see [5]) we obtain

$$P(W_{i,j} > q_i) = \frac{e^{-\mu_i q_i} n_i^{j-n_i+1}}{(n_i-1)^{j-n_i+1}}$$
$$+ e^{-n_i \mu_i q_i} \sum_{k=0}^{j-n_i} \left[\frac{(n_i \mu_i q_i)^k}{k!} - \frac{n_i^{j-n_i+1} (\mu_i q_i)^k}{k!(n_i-1)^{j-n_i+1-k}} \right] . \tag{12}$$

Note that the right-hand side of (12) is not defined for $n_i = 1$. However, in that case the conditional response time $W_{i,j}$ has a simple Erlang distribution with parameters $(j+1, \mu_i)$:

$$P(W_{i,j} > q_i) = e^{-\mu_i q_i} \sum_{k=0}^{j} \frac{(\mu_i q_i)^k}{k!} . \tag{13}$$

If the QoS measure is the waiting time rather than the response time, then the situation is more straightforward. The conditional waiting time, $w_{i,j}$, of a type i job which finds j other type i jobs on arrival, is equal to 0 if $j < n_i$ and has the Erlang distribution with parameters $(j - n_i + 1, n_i \mu_i)$ if $j \geq n_i$:

$$P(w_{i,j} > q_i) = e^{-n_i \mu_i q_i} \sum_{k=0}^{j-n_i} \frac{(n_i \mu_i q_i)^k}{k!} . \tag{14}$$

The above expressions, together with (7), enable the average revenue V_i to be computed efficiently and quickly. When that is done for different sets of parameter values, it becomes clear that V_i is a unimodal function of K_i. That is, it has a single maximum, which may be at $K_i = \infty$ for lightly loaded systems. We do not have a mathematical proof of this proposition, but have verified it in numerous numerical experiments. That observation implies that one can search for the optimal admission threshold by evaluating V_i for consecutive values of K_i, stopping either when V_i starts decreasing or, if that does not happen, when the increase becomes smaller than some ϵ. Such searches are typically very fast.

Thus, having chosen a dynamic server allocation policy, we obtain a dynamic job admission policy as follows:

- For each job type i, whenever its server allocation n_i changes, compute the 'best' admission threshold, K_i, by carrying out the search described above.

We put quotation marks around the word 'best', because that threshold may not be optimal if the exponential assumptions are violated. This admission policy can be applied in any system but is, in general, a heuristic.

We have chosen to introduce one exception to the above rule: if the allocation policy is greedy, and n_i changes because a server has been temporarily allocated to pool i as a result of not being needed elsewhere, then the threshold K_i is not recalculated. The rationale is that such a server may have to return to its original pool soon, so type i admissions should not be relaxed.

Note that, whenever $n_i = 0$ for some job type, then $K_i = 0$, i.e. jobs of that type are not accepted. If that is undesirable for reasons other than revenue, then either the allocations or the thresholds can be adjusted appropriately. In our implementation, if there has been at least one arrival of type i during the current window, the resource allocator tries to assign at least one server to pool i for the next window.

3.2 Optimal server allocation

Rather than using a heuristic server allocation policy, one might wish to determine the 'optimal' server allocation at the end of each window. The problem can be formulated as follows.

For a given set of demand and economic parameters (the former are estimated from statistics collected during the window), find a server allocation vector, $(n_1, n_2, \ldots, n_m)$, $n_1 + n_2 + \ldots + n_m = N$, which, when used together with the corresponding optimal admission threshold vector $(K_1, K_2, \ldots, K_m)$, maximizes the total average profit (2).

One way of achieving this is to try all possible server allocation vectors. For each of them, compute the corresponding optimal thresholds, evaluate the total expected revenue V, and choose the best. This exhaustive search method is feasible only when N and m are small. The complexity of determining the optimal vector $(K_1, K_2, \ldots, K_m)$ for a given vector $(n_1, n_2, \ldots, n_m)$ is on the order of the number of services, $O(m)$. Hence, the complexity of the exhaustive search is on the order of $O(sm)$, where s is the number of different ways that N servers can be partitioned among m pools (e.g., see [14]):

$$s = \left(\begin{array}{c} N+m-1 \\ m-1 \end{array} \right) . \tag{15}$$

That number grows quite rapidly with N and m.

When the exhaustive search is too expensive to be performed on-line, we propose a fast method for minimizing the cost. This can be justified by arguing that the revenue is a concave function with respect to its arguments $(n_1, n_2, \ldots, n_M)$. Intuitively, the economic benefits of giving more servers to pool i become less and less significant as n_i increases. On the other hand, the economic penalties of removing servers from pool j become more significant as n_j decreases. Such behaviour is an indication of concavity. One can therefore assume that any local maximum reached is, or is close to, the global maximum.

A fast search algorithm suggested by the above observation works as follows.

1. Start with some allocation, $(n_1, n_2, \dots, n_M)$; a good initial choice is provided by the Observed Loads heuristic.
2. At each iteration, try to increase the revenue by performing 'swaps', i.e. increasing n_i by 1 for some i and decreasing n_j by 1 for some j $(i, j = 1, 2, \dots, m\ ;\ i \neq j)$. Recompute K_i and K_j.
3. If no swap leads to an increase in V, stop the iterations and return the current allocation (and the corresponding optimal admission thresholds).

There are different ways of implementing step 2. One is to evaluate all possible swaps (there are $m(m-1)$ of them) and choose the best, if any. Another is to stop the current iteration as soon as a swap is found that improves the revenue, and go to the next iteration. In the worst case this would still evaluate $m(m-1)$ swaps, but in general the number would be smaller. The trade-off here is between the speed of each iteration and the number of iterations. Experiments have shown that the second variant tends to be faster than the first (but not always).

An algorithm of the above type reduces drastically the number of partitions that are examined. Its worst-case complexity is on the order of $O(zm^2)$, where z is the number of iterations; the latter is typically small. For example, in a system with $N = 50$, $m = 5$, the fast search algorithm found the optimal configuration in less than 0.5 seconds, whereas the exhaustive search took about 0.5 hours!

We have found some examples where the fast search algorithm terminates at a local maximum rather than the global one. However, those examples are rare and can be described as 'pathological': they involve situations where the globally best policy is to allocate 0 servers to a pool and not accept any jobs of that type. Even in those cases, the local maxima found by the algorithm provide acceptable revenues.

The fast search algorithm can be performed dynamically, at the end of each window, instead of applying a heuristic server allocation policy. When N and m are large, it is also possible to apply a 'compromise solution' whereby the number of iterations is bounded (e.g., no more than 5 iterations). The resulting policy may be sub-optimal, but any improvements will be obtained at a small run-time cost.

The dynamic optimization was implemented in one of the experiments described in the next section.

4 Numerical and empirical results

Several experiments were carried out, with the aim of evaluating the effects of the server allocation and job admission policies that have been proposed. To reduce the number of variables, the following features were held fixed:

- The QoS metric is the response time, W.
- The obligations undertaken by the provider are that jobs will complete within twice their average required service times, i.e. $q_i = 2/\mu_i$.
- All penalties are equal to the corresponding charges: $r_i = c_i$ (i.e., if the response time exceeds the obligation, users get their money back).

Unless otherwise stated, the arrival processes are Poisson and the service times are distributed exponentially. However, there will be experiments with clustered arrivals and with non-exponential service times.

The first experiment is purely numerical. It examines the effect of the admission threshold on the achievable revenue. A single service is offered on a cluster of 10 servers ($N = 10,\ m = 1$). The average job length and the charge per job are $1/\mu = 1.0$ and $c = 100.0$, respectively. The results are shown in Figure 2, where the revenue earned, V, is plotted against the admission threshold, K, for three different arrival rates.

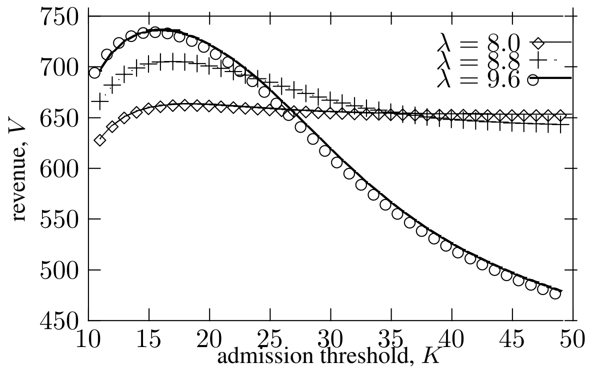

Figure 2. Revenue as function of admission threshold

$$N = 10, m = 1, \mu = 1.0, c = r = 100$$

The figure illustrates the following points, of which the last is perhaps less intuitive than the others.

(a) In each case there is an optimal admission threshold.

(b) The heavier the load, the lower the optimal threshold but the higher the maximum achievable revenue.

(c) The heavier the load, the more important it is to operate at or near the optimum threshold.

Thus, when $\lambda = 8.0$, the optimal admission threshold is $K = 18$; however, much the same revenue would be earned by setting $K = 10$ or $K = \infty$. When the arrival rate is $\lambda = 8.8$, about 10% higher revenue is obtained by using the optimal threshold of $K = 17$, compared with the worst one of $K = \infty$. When the arrival rate increases further to $\lambda = 9.6$, the revenue drops very sharply if the optimal admission threshold of $K = 16$ is exceeded significantly.

In the next experiment, a 20-server system with 2 types of service was subjected to fluctuating demand controlled by a single parameter, λ. During a period of time of length 1000, jobs of type 1 and 2 arrive at rates $\lambda_1 = \lambda$ and $\lambda_2 = 10\lambda$, respectively. Then, during the next period of length 1000, the arrival rates are $\lambda_1 = 10\lambda$ and $\lambda_2 = \lambda$, respectively; and so on. The average service times for the two types are equal, $1/\mu_1 = 1/\mu_2 = 0.8$, as are the charges, $c_1 = c_2 = 100$.

The aim is to compare the revenues that would be earned by three operating policies, assuming that the demand parameters did not have to be estimated. The first is the optimal policy, obtained by exhaustive search of all possible server allocations and the corresponding admission thresholds. The second is, essentially, the Measured Loads heuristic (except that no windows are necessary because the loads are given). The third policy uses the same server allocations as the heuristic, but does not restrict admissions (i.e., $K_1 = K_2 = \infty$). In all cases, it is assumed that, at the beginning of every new period, the demand parameters become known instantaneously, so that the server allocations and admission thresholds can be computed and applied during that period. This experiment was carried out by simulation.

In Figure 3, the total revenue earned per unit time by the three policies is plotted against the offered load (which is equal to $11\lambda/\mu_1$), expressed as a percentage of the saturation load. The near-optimality of the heuristic is rather remarkable. In contrast, the revenues earned by the unrestricted admission policy increase more slowly, and then drop sharply as the load becomes heavy. This example demonstrates that, by itself, a sensible server allocation is not enough; to yield good results, it should be accompanied by a sensible admission policy.

All subsequent experiments were carried out on the SPIRE system. Jobs were generated, queued and executed on real servers; messages were sent and delivered by a real network. Random interarrival intervals were generated by client processes and random service times were generated at the server nodes.

SPIRE has been implemented in Java and deployed on Apache Axis2 1.1.1. Nodes exchange SOAP messages over HTTP using WS-Addressing [16]. The cluster is composed of machines running Linux version 2.6.14 and interconnected via a Gbps ethernet. The load generator is a process running on a separate computer. The connection between that client and the controller is provided by a 100 MBps Ethernet network. The average round trip time between nodes and the controller is 0.258 ms, while the one

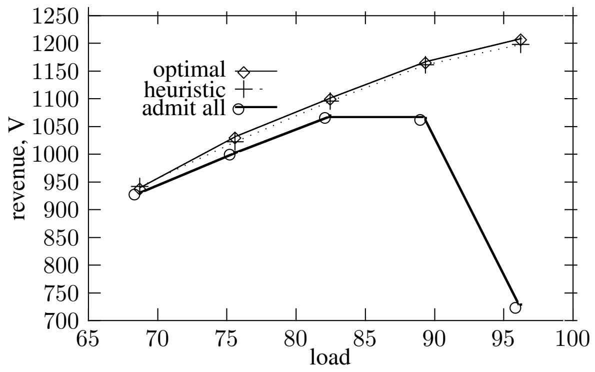

Figure 3. Policy comparisons: revenue as function of load

$$N = 20, \mu_i = 0.8, c_i = r_i = 100$$

between the client and the controller is 0.558 ms (nevertheless, since both the servers and the network are shared, unpredictable delays due to other users are possible).

Except where indicated explicitly, the 'conservative' versions (see Section 3) of the server allocation policies are used.

In the following two examples, a 20-server system provides two types of services. The arrival rate for type 1 is kept fixed, at $\lambda_1 = 0.2$, while λ_2 takes different values ranging from 0.4 to 1.8. The average service times for the two types are $1/\mu_1 = 50$ and $1/\mu_2 = 5$ respectively (i.e., type 1 jobs are on the average 10 times longer than type 2). Thus, the offered load of type 1 is 10, while that of type 2 varies between 2 and 9; the total system load varies between 60% and 95%. The charges for the two types are equal: $c_1 = c_2 = 100$.

Three operating policies are compared: (a) A 'Static Oracle' policy which somehow knows the values of all parameters. At time 0 of each run, it uses the Measured Loads heuristic to compute the server allocations n_1 and n_2, and the admission thresholds K_1 and K_2; thereafter, the configuration remains unchanged. (b) The dynamic Measured Loads heuristic. The windows which separate consecutive reconfiguration decisions are defined so that a total of J jobs (of all types) arrive during a window. That parameter may take different values. (c) The dynamic Measured Queues heuristic. Same definition of a window.

For both dynamic heuristics, at the start of a run, before any statistics have been collected, servers are allocated on demand as jobs arrive; the initial admission policy is to accept all jobs (i.e., $K_i = \infty$).

Clearly, the Static Oracle policy could not be used in practice, since the demand parameters are not usually known in advance; it is included here only for purposes of comparison.

The average revenues obtained per unit time are plotted against the arrival rate for type 2, λ_2. Each point represents a separate system run, during which at least 1000 jobs of type 1 are processed (the number of type 2 jobs is larger, and increases with λ_2). In the case of the Observed Queues heuristic, the corresponding 95% confidence intervals are also shown. Those for the other policies are omitted in order not to clutter the graphs; they confirm the well-known observation that the confidence intervals tend to grow with the load.

Figures 4 and 5 illustrate the behaviour of the system for two different window sizes, $J = 30$ and $J = 150$ jobs, respectively.

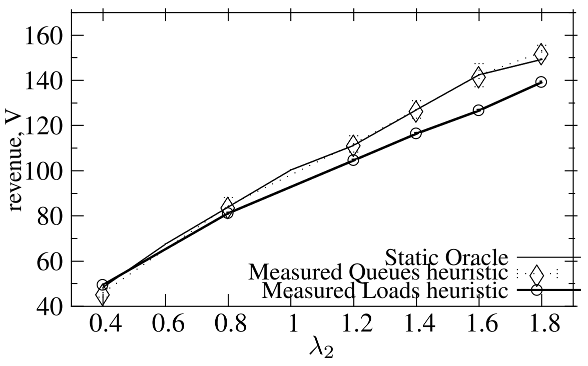

Figure 4. Dynamic policies in SPIRE: window size 30 jobs

$N = 20, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$

When the window size is small, the dynamic Measured Loads heuristic performs slightly worse than the other two policies. This is probably due to the fact that the arrival and service rate estimates obtained during each window are not sufficiently accurate. The Measured Queues heuristic appears to react better to random changes in load.

The effect of increasing the window size to 150 jobs is not very pronounced. The Measured Loads heuristic is now, if anything, slightly better than the Measured Queues heuristic, and both sometimes outperform the Static Oracle policy. However, the differences are not statistically significant. Corresponding points are mostly within each other's confidence intervals.

It is perhaps worth mentioning that the relative insensitivity of the system performance with respect to the window size is, from a practical point of view, a very good feature. It means that the decision of what window size to choose is not too critical.

In the next several experiments, the arrival process of type 2 is no longer Poisson, but consists of bursts. More precisely, the observation period is divided into alternating periods of lengths 180 seconds and 60 seconds, respectively.

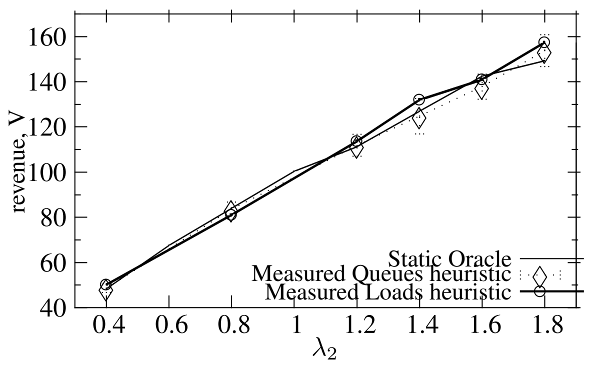

Figure 5. Dynamic policies in SPIRE: window size 150 jobs

$N = 20, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$

During the longer periods, $\lambda_2 = 0.4$ jobs/sec, while during the shorter periods it is about 12 times higher, $\lambda_2 = 5$ jobs/sec. The other demand parameters are the same as before. Note that if the higher arrival rate was maintained throughout, the system would be saturated.

Figure 6 illustrates the revenues obtained by the Measured Loads and Measured Queues heuristics, for three different window sizes. For purposes of comparison, an Oracle policy which knows the values of all parameters and makes optimal allocations at the beginning of each period was also evaluated. Its performance is of course independent of the window size.

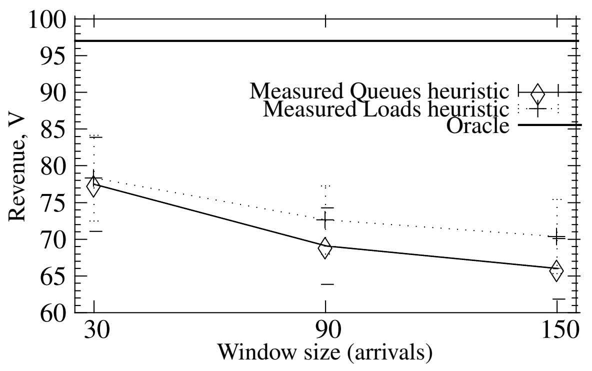

Figure 6. Bursty arrivals: different window sizes; equal charges

$N = 20, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$

As might be expected in view of the highly variable demand, shorter windows are now significantly better than long ones. We also note that the the Measured Loads heuristic consistently outperforms the Measured Queues one. However, the differences are not large. The largest difference between the two heuristics is about 5%; the largest loss of revenue due to using a window whose size is too large is about 20%. Moreover, the revenues achieved by both heuristics are within about 75% of the ideal (and unrealizable) revenue of the Oracle policy.

To examine the effect of applying different charges to different services, the experiment was re-run with the same demand parameters, but doubling the charge (and the penalty) for type 1 jobs: $c_1 = 200$, $c_2 = 100$. Note that this change does not affect the allocation heuristics (since $\alpha_i = r_i/c_i = 1$), but it may affect the admission policies. The results are shown in Figure 7.

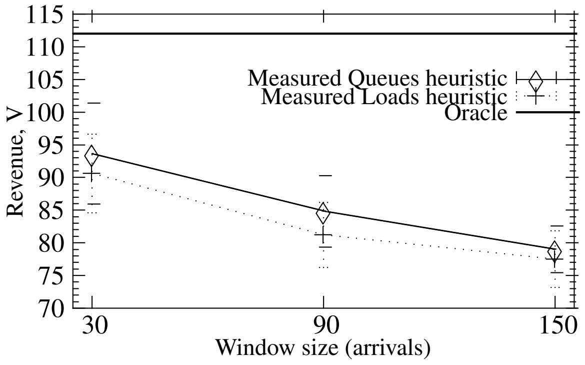

Figure 7. Bursty arrivals: different window sizes; unequal charges

$N = 20, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_1 = 200, c_2 = 100$

The behaviour we observe is similar to that in Figure 6. Revenues are higher, in line with the increased charges for type 1 jobs. Again, shorter windows are better than longer ones, but not by much. It is a little surprising that the Measured Queues heuristic now appears to outperform the Measured Loads one. However, the differences are rather small; the corresponding points are well within each other's confidence intervals. Both policies achieve at least 70% of the Oracle's revenues.

It is interesting to observe the effect of replacing the conservative versions of the allocation policies with the greedy ones (i.e., unused servers are temporarily reallocated to other pools before the end of the current window).

Figure 8 shows that there is a marked improvement in the performance of both heuristics. The revenues they achieve are now within about 90% of those of the Oracle policy. The Measured Queues heuristic still outperforms Measured Loads, but only slightly. More importantly, both policies are now even less sensitive with respect to the window size. The relative differences between the best and the worst average revenues are less than 5% (the confidence intervals are quite large, but that is explained by the variability of demand).

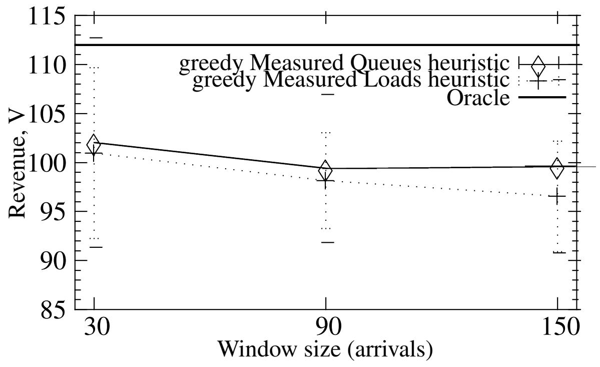

Figure 8. Greedy policies, bursty arrivals: unequal charges

$N = 20, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_1 = 200, c_2 = 100$

To get a better idea of why conservative policies are affected to a greater extent by window sizes, consider Figure 9, which traces the behaviour of a system similar to the one in Figure 6 (the bursts are less pronounced here; $\lambda_2 = 1.8$ during burst periods). The top part of the figure shows the arrivals (each point represents the average number of type 2 arrivals per second during a 10-second interval). The two plots in the second part describe how many servers were allocated to type 2 by the Observed Queues heuristic, using windows of 30 and 150 jobs, respectively ($n_1 = 20 - n_2$). The third part gives a similar description for the Observed Loads heuristic. It is very clear from the figure that the smaller window size enables both heuristics to react better to the bursts in load.

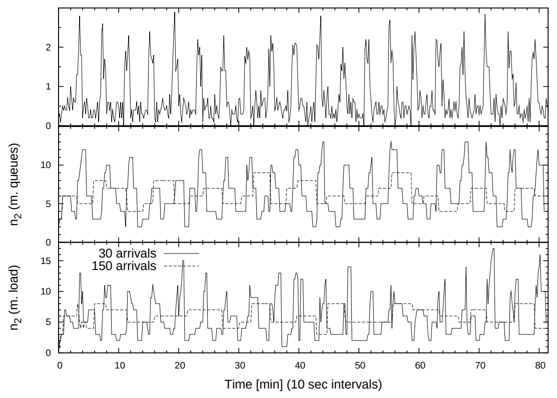

Figure 9. Dynamic server allocation
$N = 20, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_1 = c_2 = 100$

The reason why the greedy versions of the heuristics are less affected by the window size is that a 'wrong' server allocation can be adjusted as soon as its symptoms appear, without waiting for the end of the window.

Another aspect of system performance is the rejection rate, i.e. the total average number of jobs that are not admitted into the system, per unit time. We have concentrated on revenue, but this metric may be of interest in some circumstances. Some of the traces of rejections caused by the conservative and greedy versions of the Measured Loads heuristic are illustrated in Figure 10, for the same parameters as in Figure 6.

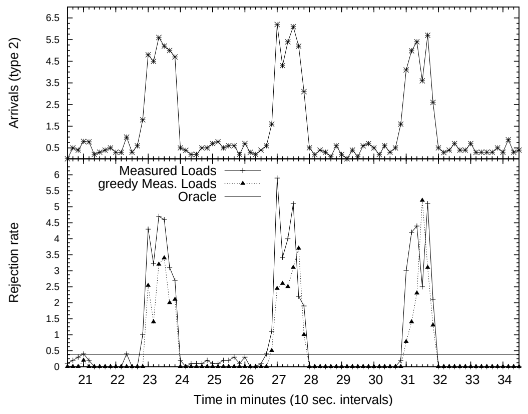

Figure 10. Rates of rejection: conservative and greedy policies

$$N = 20, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_1 = c_2 = 100$$

The top part of Figure 10 shows the bursts of type 2 arrivals, while the bottom part displays the numbers of rejections per second under the two versions of the policy. It is very noticeable that the greedy policy rejects fewer jobs than the conservative one. The 'ideal' rejection rate of the Oracle policy is also shown for comparison; that rate is not achievable.

The next experiment departs from the assumption that all service times are distributed exponentially. Now the service time distributions are two-phase hyperexponential. Type 1 jobs have mean 32.9 with probability 0.7 and mean 90 with probability 0.3; the overall average service time is 50 seconds, as before. The corresponding parameters for type 2 are mean 3.71 with probability 0.7, and mean 8 with probability 0.3; the overall average service time is 5 seconds. These changes increase the variances of the service times, while preserving the averages. A window size of 150 jobs was used. Note that the Measured Loads heuristic for allocating servers is not affected significantly, since it is based on averages only. However, one might expect that the admission decisions will now be 'wrong' more often (since the thresholds are still calculated assuming exponentially distributed service times), and that the revenues will drop as a consequence.

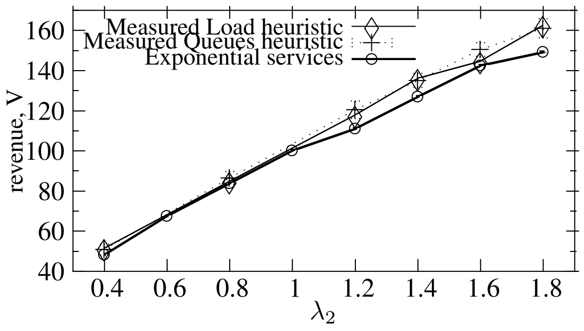

Figure 11. Hyperexponential service time distribution

$$N = 20, \lambda_1 = 0.2, c_1 = c_2 = 100$$

In fact, Figure 11 shows that this is not the case. Not only have the revenues not decreased, compared with those obtained when the distribution is exponential, but they have increased slightly. In other words, the procedures for making admission decisions under both heuristics are sufficiently robust to cope with violations of the distributional assumptions. The slight increases in revenue are probably explained by a reduction in the penalties paid (because most jobs are shorter).

The aim of final experiment is to evaluate the extent to which revenues can be improved by carrying out the dynamic optimization described in Section 3.2, rather than using a heuristic allocation policy. The usual 20-server cluster now offers 3 types of services, with average service times $1/\mu_1 = 40$, $1/\mu_2 = 10$ and $1/\mu_3 = 4$, respectively. The arrival rates for types 1 and 2 are fixed, at $\lambda_1 = 0.175$ and $\lambda_2 = 0.6$, while that of type 3 increases from $\lambda_3 = 0.25$ to $\lambda_3 = 1.5$. Thus, the total offered load is increased from moderate, 14, to heavy, 19. The charge/penalty values differ between the three types: $c_1 = r_1 = 500$, $c_2 = r_2 = 250$, $c_3 = r_3 = 100$.

The 'Optimal' policy estimates the demand parameters during each window and uses the fast search algorithm of Section 3.2 to find the optimal allocations and thresholds that will apply during the next window. The resulting revenues are compared, in Figure 12, with those of the Measured Loads heuristic which in this case allocates servers in proportion to the observed loads λ_i/μ_i (and then computes the best thresholds).

This experiment confirms that it is feasible to search for the optimal configuration on-line: the fast search algorithm took about 30 milliseconds at the end of each window.

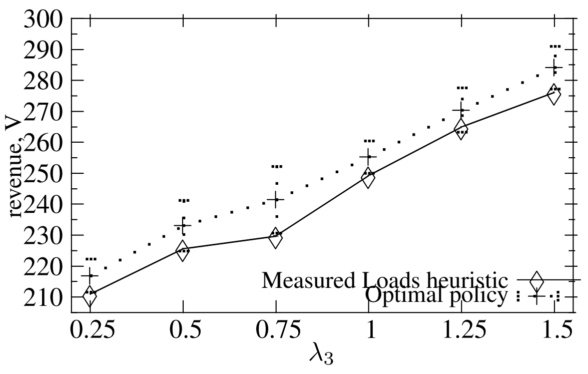

Figure 12. Three job types: optimal and heuristic policies

$$N = 20, \rho_1 = 7, \rho_2 = 6, \rho_3 \in [1, 6]$$

That policy does yield consistently higher revenues than the heuristic. On the other hand, we also note that the relative improvement is not very great; it is on the order of 5% or less. One could therefore interpret these results also as a confirmation that the Measured Loads heuristic is a pretty good policy for the control of a service provisioning system.

5 CONCLUSIONS

This paper examines the market in computer services. It introduces (a) a quantitative framework where performance is measured by the average revenue earned per unit time, and (b) a middleware platform that provides an appropriate infrastructure. We have demonstrated that policy decisions such as server allocations and admission thresholds can have a significant effect on the revenue. Moreover, those decisions are affected by the contractual obligations between clients and provider in relation to the quality of service.

Dynamic heuristic policies for server allocation and job admission have been proposed and shown to perform well. Indeed, there is some evidence that they are close to optimal. An important feature from a practical point of view is that these heuristics are not very sensitive with respect to the window size that is used in their implementation. Moreover, they are able to cope with bursty arrivals and non-exponential service times.

The Measured Queues heuristic appears to offer no significant performance advantage, compared to the Measured Loads heuristic. Since the estimation of arrival rates and average service times is in any case necessary for the computation of admission thresholds, one could ignore the queue sizes and operate just the Measured Loads heuristic.

The following are some directions for future research.

1. Relax the Markovian assumptions used in computing the admission thresholds. This would probably imply abandoning the exact solutions and seeking approximations.

2. Rather than operating an admission policy, one might have a contract specifying the maximum rate at which users may submit jobs of a given type, and then be committed to accepting all submissions. The question would then arise as to what that maximum rate should be.

3. It may be possible to share a server among several types of services. Then one would have to consider different job scheduling strategies, e.g., preemptive and non-preemptive priorities, Round-Robin, etc.

4. System reconfigurations, such as switching a server from one type of service to another, may incur non-negligible costs in either money or time. Taking those costs into account would mean dealing with a much more complex dynamic optimization problem.

All of the above avenues, and possibly others, are worth pursuing. In addition, if this methodology is to be applied in practice, it may be necessary to carry out some market research. It would be useful to discover what kind of response time or waiting time obligations the users might ask for, and how much they would be willing to pay for them.

Acknowledgements

This work was carried out as part of the research project QOSP (Quality Of Service Provisioning), funded by British Telecom. It was also supported by the European Union Network of Excellence EuroFGI (Future Generation Internet).

References

[1] M. N. Bennani and D. A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *Procs. 2nd IEEE Int. Conf. on Autonomic Computing*, pages 229–240, 2005.

[2] A. Chandraa, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Procs. 11th ACM/IEEE Int. Workshop on Quality of Service*, pages 381–400, June 2003.

[3] M. N. B. Daniel A. Menascé and H. Ruan. On the Use of Online Analytic Performance Models, in Self-Managing and Self-Organizing Computer Systems. In *Self-* Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 128–142. 2005.

[4] S. Ghosh, R. Rajkumar, J. Hansen, and J. Lehoczky. Scalable Resource Allocation for Multi-Processor QoS Optimization. In *Procs. 23rd Int. Conf. on Distributed Computing Systems*, pages 174–183, 2003.

[5] I. Gradshtein and I. Ryzhik. *Table of Integrals, Series, and Products*. Academic Press, 1980.

[6] J. P. Hansen, S. Ghosh, R. Rajkumar, and J. Lehoczky. Resource Management of Highly Configurable Tasks. In *Procs. 18th Int. Parallel and Distributed Processing Symposium*, pages 116–123, 2004.

[7] B. A. Huberman, F. Wu, and L. Zhang. Ensuring Trust in One Time Exchanges: Solving the QoS Problem. *NETNOMICS*, 7(1):27–37, April 2005.

[8] V. Kanodia and E. W. Knightly. Multi-Class Latency-Bounded Web Services. In *Eighth Int. Workshop on Quality of Service*, pages 231–239, 2000.

[9] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[10] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance Management for Cluster Based Web Services. In *IFIP/IEEE 8th Int. Symposium on Integrated Network Management*, pages 247–261, 2003.

[11] I. Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1998.

[12] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Procs. 18th IEEE Real-Time Systems Symposium*, pages 298–307, December 1997.

[13] M. Sallé and C. Bartolini. Management by Contract. In *Procs. 2004 IEEE/IFIP Network Operations and Management Symposium*, April 2004.

[14] N. Y. Vilenkin. *Combinatorics*. Academic Press, 1971.

[15] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning Servers in the Application Tier for E-Commerce Systems. *ACM Trans. on Internet Technology*, 7(1), February 2007.

[16] W3C. *http://www.w3.org/Submission/ws-addressing/*.