On Fully Distributed Adaptive Load Balancing

David Breitgand[1], Rami Cohen[2], Amir Nahir[1], and Danny Raz[2]

[1] IBM Haifa Research Lab, Israel
[2] CS Department, Technion, Haifa

Abstract. Monitoring is an inherent part of the management loop. This paper studies the problem of quantifying utility of monitoring in a fully distributed load balancing setting. We consider a system where job requests arrive to a collection of n identical servers. The goal is to provide the service with the lowest possible average waiting time in a fully distributed manner (to increase scalability and robustness).

We present a novel adaptive load balancing heuristic that maximizes utility of information sharing between the servers. The main idea is to forward the job request to a randomly chosen server and to collect load information on the request packet as it moves on. Each server decides, based on that information, whether to forward the job request packet to another server, or to execute it locally. Our results show that in many practical scenarios this self-adaptive scheme, which does not require dedicated resources for propagating of load information and decision making, performs extremely well with respect to best known practice.

1 Introduction

To maximize value of Information Technology (IT), its low level management policies have to be aligned with the high level business goals, which in many cases impel systematic reduction of management overheads. In this paper, we concern ourselves with the monitoring overhead present in any management loop. Consequently, it is important to maximize the utility of monitoring in order to improve the utility of the overall management process.

Consider for example a service that is being provided by a set of servers over the network. The goal of the service provider is to provide the best service (say, minimizing the response time) given the amount of available resources (*e.g.*, the number of servers). The provider can add a load sharing system (for example as suggested in RFC 2391 [1]) and improve the response time. However, the same resources (budget) can be used to add additional servers to the system and thus provide better service to end customers. The dilemma here is between adding more computational power and adding management abilities, where the goal is to achieve the best improvement in the overall system performance.

Simple load balancing schemes, such as random selection or Round Robin are oblivious to actual server load when making job assignment decisions. This may work well for workloads with low variability. However, load-oblivious algorithms lack adaptiveness and therefore may perform poorly for workloads exhibiting

A. Clemm, L.Z. Granville, and R. Stadler (Eds.): DSOM 2007, LNCS 4785, pp. 74–85, 2007.

medium to high variability. In order to considerably improve expected response time in such cases, load-aware algorithms are required. These algorithms need updated load information from the servers. Handling such load information requests requires small but nonzero resources (*e.g.*, CPU) from each server. Thus, it is not easy to predict the actual amount of improvement expected from preferring a specific configuration. It is thus important to identify just the right amount of resources that should be allocated to management tasks (such as monitoring) in order to maximize the overall system performance.

Typically, load balancing is implemented via a centralized dedicated entity that receives all requests and assigns servers to the requests. This option requires additional resources and limits the system's scalability and robustness. In [2] we extensively studied quantifying of monitoring utility in such environment. As shown in [2], for each service request rate, there exists an optimal number of servers that should be monitored in order to maximize utility of monitoring or reducing the total service time. This is a very generic result, which is applicable to any management scheme that employs explicit monitoring components.

In this paper, we extend these results and develop a very efficient fully distributed and self-adaptive load balancing scheme. The main idea behind the scheme is as follows. When a new job request arrives at an idle server, the server executes it locally. Otherwise, it adds its local load information to the job request packet and forwards the request to a randomly chosen peer in the cluster. A peer server that receives the packet with the senders' load information on it, compares it to its own load and makes a decision whether to execute the job locally, or to further forward it to another peer. This way, the load information on the request packet is collected *en-route*. When the d-th server receives the job request packet, the latter contains information about the load of other $d-1$ servers. As d grows, this information becomes more out of date and the waiting time prior to execution grows linearly with d. Thus, there is a tradeoff between the number of hops a job request may travel before getting executed at the least loaded server and both the delay it spends and the quality of load information that is used to determine the least loaded server.

We study several heuristics for optimizing this tradeoff and evaluate their performance using extensive simulations and an implementation on a real testbed. The primary results are represented by the self-adaptive heuristics, in which the system adapts to the changing environmental conditions (i.e., load, type of requests and their service time, *etc.*) in a fully distributed scheme. It turns out that in many realistic scenarios self-adaptiveness performs extremely well, resulting in significant performance gains.

The rest of this paper is organized as follows. In Section 2 we formally define the model for the framework. In Section 3 we describe two advanced heuristics, that are self-adaptable to the load conditions and compare them to the optimally configured Centralized Monitoring (CM scheme). In Section 4 we describe the implementation of the scheme and present its performance evaluation on a real set of servers. Section 5 describes related work. We conclude in Section 6 with a short discussion of our results.

2 Model

In this section we describe our model and provide intuitive motivation for selecting the main factors that influence the total performance of any distributed load balancing mechanism.

As described in the previous section, we consider a fully distributed server system in which client requests arrive in a set of Poisson streams of traffic intencity (load) λ to a set of n identical servers. The total load in the system is $n \cdot \lambda$. We assume a *non preemptive* load sharing model, in which a job that started executing at a server cannot move to another server. For the sake of simplicity we assume a single FCFS queue for job requests at each server. When such a request arrives at a server, the server has to decide whether to serve the job request locally or forward it to another server. The information available to the server when making this decision includes local information and information attached to the job request. The local information contains the server current queue length and statistics describing the workload parameters computed from the local execution history. These statistic information include job request frequency and average job service time. The information provided by the job request itself contains client based information, an estimation of service time, and information added by other servers in the system if the request was forwarded from another server and not directly received from a client.

In order to make the algorithmic decision of whether to accept the request or forward it and to which server to forward it, the server has to stop serving the current job (if the server is not idle), and to allocate resources (CPU) to the management process. It has to examine local data and data provided with the request, run the decision algorithm, and, if needed, forward the request. This delays the execution of the current job and of all jobs that wait for execution in the local queue. Note that in case of a multi-CPU server, the situation is essentially the same, since dedicating a full separate CPU just to serve control requests is wasteful. Thus, preemption of client requests will be unavoidable under high load conditions. To verify this point, we implemented a CPU intense server as described in Section 4. This multithreaded server was executed on a a blade server containing Dual PowerPC machines, each having a 2.2GHz 64-bit CPU, with 4GB of RAM. Figure 1 depicts the average normalized execution time (from the time the job started to be executed until it terminated) as a function of the number load queries per job. The net service time (measured in a setting without load) was about 165 milliseconds, but as can be seen clearly from the figure, the actual time increases linearly with the number of monitoring requests. This shows that even in an advanced architectures, the monitoring overhead is not negligible.

Therefore, the ratio between the time it takes a server to handle such job request and the expected mean service time is a critical factor that affects the overall performance of the system. This *overhead efficiency ratio*, denoted by C, reflects the amount of impact the distributed management task has on the actual service. In this intense CPU setting, where a user request takes about 165 milliseconds, each load request takes about 0.33-0.5 milliseconds so C is between 0.002 and 0.004.

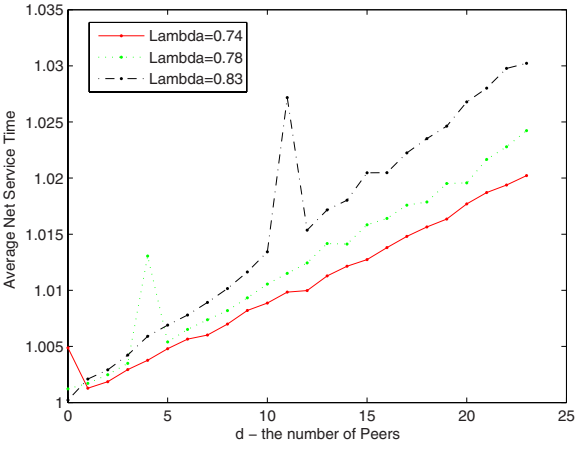

Fig. 1. Execution time as a function of the number peers' load requests

When a job request is forwarded, it takes time until it arrives to the next server, and this time is added to the overall response time. Thus, another important factor is the ratio between the communication time between servers and the mean expected service time of a job, called *communication ratio*. This ratio is denoted by CT. When a server examines a job request, the load information piggybacked in it may be staled due to communication delay. If the delay is small relatively to the service time, the affect of the delay may be negligible, but otherwise it may have a detrimental affect on the total performance. As an example of the former case consider servers communicating over a Gbit Ethernet and an average service time of 300 milliseconds. As an example of the latter case consider servers distributed throughout the Internet and an average service time of about 100 milliseconds; in this case communication time is in the same order as service time and the load information piggybacked on the job request may be totally out of date.

Both the communication ratio and the overhead efficiency ratio depend on physical aspects of the system such as the communication delay and the CPU and memory abilities, and on the service provided that determines the distribution of jobs' service time.

Let d be the number of hops (the number of servers) an incoming job request travels before it is being admitted to a local FCFS queue of some server. Effectively, this is the number of servers being implicitly monitored by this job request. Let $\frac{1}{\mu}$ denote the mean service time. Then the expected mean response time in the system $E(R)$ is given by Equation 1, where $\bar{L}$ is the average queue length.

$$E(R) = \frac{1}{\mu}(\bar{L}(1 + d \cdot C) + d \cdot CT) \tag{1}$$

The first factor is due to the queue and the fact that on the average d job requests arrive during the service of a job, each consumes C fraction of the job

service time. The second factor is due to the time it takes to move from one server to another.

3 Self Adaptive Heuristics

In Basic heuristic that is used for baselining, if the server is idle or the server receives a job request that traveled more than d hops, the server serves the request. Otherwise, if the job request traveled less than d hops, the server forwards the request to a random new server. If the job request traveled exactly d hops, the server finds a server with the shortest queue among these d, according to information on the request, and forwards the job to this server. This heuristic basically says that as long as we did not examine d servers we stop only if we find an idle server. After examining d servers, we execute the job on the least loaded (the one with the shortest queue) server among these d.

The important factor is, of course the choice of d. The dependency of the average queue length (and thus the average time in the system) on d is rather complex. On the one hand, as d increases we have a higher probability of finding an idle server (or a server with a small queue) and thus reducing the average queue length. On the other hand, as d increases more and more servers have to forward the same job request thus "paying" a portion of C resources and slowing down the actual service.

The optimal value of d depends on the system parameters such as the load, the overhead efficiency ratio C, and the communication ratio CT. While C and CT are almost an invariant for a given system, the load of the system may change often. Thus monitoring a constant number of servers d (explicitly or via piggybacking) is not optimal. In this section, we present two self adaptive heuristics in which the number of examined servers dynamically changes with the system load.

In the centralized model, in which one dispatcher receives all the jobs, the mean time between jobs arrival together with the mean service time and the number of servers, determine the load in the system. Updating the load can be done dynamically by considering the mean time between job arrivals measured so far and the new time between consecutive job arrivals, every time a new job arrives. Thus, if two consecutive jobs arrive to the dispatcher at t^+ and t^- ($t^+ > t^-$), the new mean time between job arrivals, calculated at time t^+ is:

$$mtba(t^+) = \alpha \cdot mtba(t^-) + (1 - \alpha) \cdot (t^+ - t^-), \tag{2}$$

where $0 < \alpha < 1$ is a parameter that defines the speed in which the system adapts to the new settings. When the distributed model is considered, each server receives only a small part of all the jobs. Thus, the load measured by computing the mean time between job arrivals locally at each server can be biased and may not reflect the actual load of the entire system. To overcome this problem, each server computes the load by considering the mean time between job arrivals using only the population of jobs that served by this server, but not those forwarded to other servers. This estimation is much more robust since the

served jobs are distributed more evenly due to the load balancing mechanism that is used, and the mean time between local job assignments approximates the actual mean inter-arrival time of the jobs in the system.

The theoretical analysis in [2] provides a way to determine an optimal number of servers that should be monitored for each set of values of load and C. These values can be used to estimate the optimal value of d. In our first self adaptive heuristic, each server maintains a lookup table that contains an optimal value of d for different values of load and C. When a new job arrives to a server it can locally determine (based on its local estimation of the load) what the optimal value of d is. Then, using *Basic* heuristic, it decides whether to accept the job or to forward it to a different random server.

Forwarding a job from one server to another can reduce the waiting time of the job if the queue length in the next server is shorter than the current queue length. However, there is also a cost associated with this process. This cost includes the communication cost and the processing time it takes to monitor the next server. In the second self adaptive heuristic we present, every server that receives a job, evaluates the cost associated with forwarding the job to another server and compares it with the expected benefit. If the benefit is greater than the cost, the server forwards the job, otherwise, the server assigns the job to the server with the shortest queue among all servers that were visited by this job request so far.

Forwarding a job to another server increases the response time of that job by CT. Moreover, the new server has to invest CPU in order to handle this job request. This increases the service time of all jobs waiting to be served at this new server by C times the mean service time. Thus, the cost of forwarding a job in terms of the mean service time is $CT + \bar{L} \cdot C$, where $\bar{L}$ is the average queue length at the new server.

If the queue length of the new server is shorter than the minimum queue length found so far, then it is beneficial to forward the job to the new server. Denote by $p(i)$ the probability that the queue length in the new server is equal to i and by $\mathcal{Q}(min)$ the minimum queue length among servers visited so far. The expected benefit from forwarding a job to a new server in terms of the mean service time is:

$$\mathcal{B} = \sum_{i=0}^{\mathcal{Q}(min)} (\mathcal{Q}(min) - i) \cdot p(i). \tag{3}$$

In [3] the author shows that the probability that a queue length is greater than or equal to i is $s(i) = \lambda^{\frac{d^i-1}{d-1}}$. Considering the overhead efficiency ratio C, the effective load is increased by $\frac{1}{1-a \cdot C \cdot d}$, where a is some constant (see [2]), therefore we use $\lambda = \lambda' \cdot \frac{1}{1-2C \cdot d}$ for the system load, where λ' is the incoming load. Thus,

$$p(i) = s(i) - s(i+1) = \lambda^{\frac{d^i-1}{d-1}} - \lambda^{\frac{d^{i+1}-1}{d-1}}. \tag{4}$$

The current load in the system λ can be estimated using the same technique discussed in the first heuristic, and d the number of peers is the average number

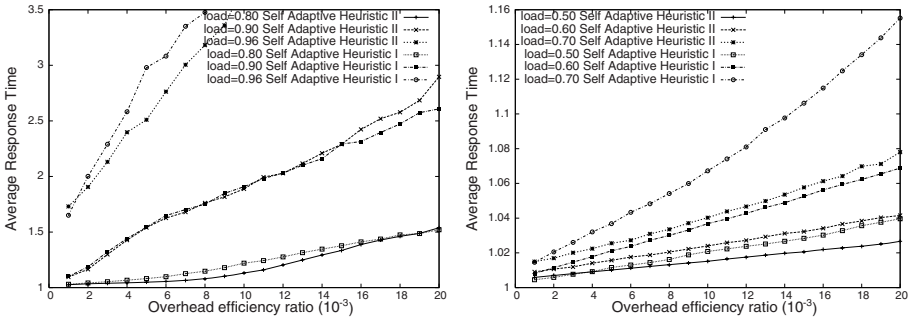

Fig. 2. Average service time as a function of the overhead efficiency ratio

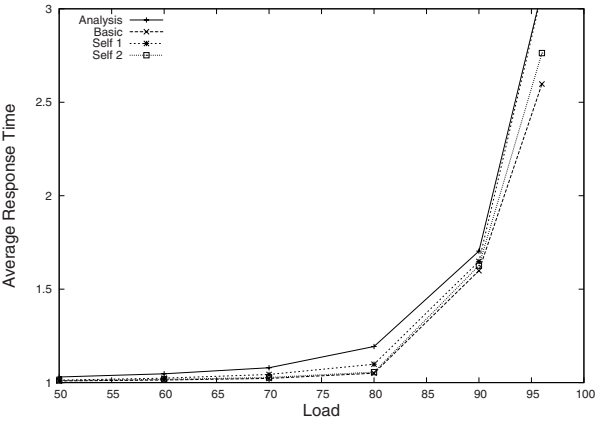

Fig. 3. Performance of the self adaptive heuristic as a function of the system load - Simulation results

of peers visited so far. Using the above formulae and these estimations, each server can compute the expected cost and benefit and forwards the job only if the expected benefit exceeds the cost.

Figure 2 depicts the average response time vs. the overhead efficiency ratio C for a different values of load, obtained by simulating the two self adaptive heuristics over a LAN environment. In most cases, especially when the load is relatively low, the second heuristic achieves better results compared to the first heuristic, while in high load their performance is similar. This is due to the fact that while the first heuristic uses a general model to derive the optimal d, the second heuristic is more sensitive to the actual queue lengths found so far.

Figure 3 demonstrates average response time of the self-adaptive heuristics, for $C = 5 \cdot 10^{-3}$ and a very small communication ratio. For comparison we also plot the average service time of the Basic heuristic, where for each run k is set to be the optimal value for this load value, and the expected average service time in the centralized model of [2]. As one can see, the adaptive methods (without

any tuning) perform as good as the algorithms optimally configured for the specific load level and the basic distributed heuristic.

4 Practical Implementation and Evaluation

In order to study the practical performance of a distributed load sharing system as described in this paper and the actual usefulness of the theoretical results, we implemented such a server based system and tested its performance on a testbed network.

Each server is comprised of two main components:

1. **Service Component:** this is the component which performs user requests. When no user requests exist, this component is idle. When a new user request arrives (queued by the main component see below), the Service Component is interrupted, it deques the user request and processes it. Processing of a request is implemented as a busy-wait loop emulating CPU intensive service, where the service duration depends on a tunable parameter and the servers CPU power. Upon completion, a response is sent to the originating user, and if additional user requests exist, the Service Component proceeds with the processing of the next request.
2. **Main Component:** this component listens for job requests (received directly from the user or forwarded from another server). Whenever a job request is received, this component determines whether to process the request locally, or to forward it to another server. If the server is idle the request is processed locally. Otherwise, the specific logic implementing the heuristic is deployed and the decision is made.

The load estimation mechanism required for both adaptive heuristics is implemented based on Equation 2. Whenever a server queues a job request, it calculates the length of the time interval between the previous job request's

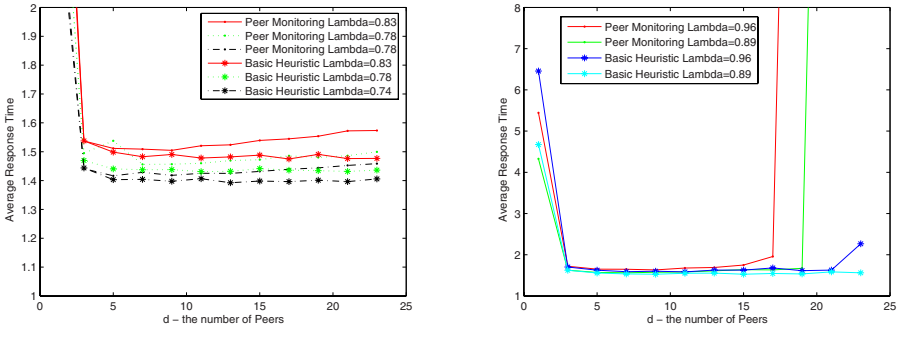

Fig. 4. Performance of the scheme on a testbed

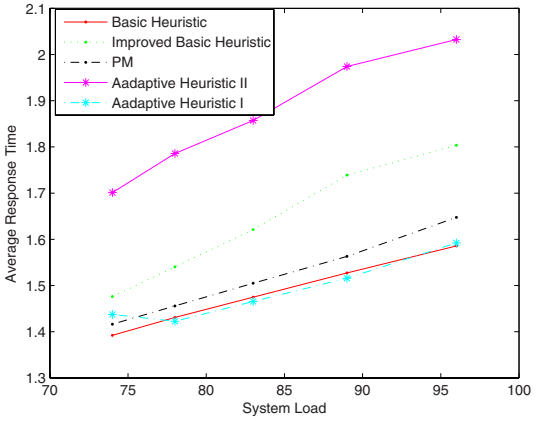

Fig. 5. Performance of the self adaptive heuristic as a function of the system load

arrival time and the current one's, and updates its load estimation. For the purpose of testing, $\alpha = 0.9$ was used. The average queue length, $\bar{L}$, needed for the implementing the scond heuristic is estimated locally in a similar manner.

The entire server is implemented in Java. Each component is implemented as a Java thread. The Main Component acts as the producer in a consumer-producer relationship between the Main and Service Components. Jobs received by the Main Component which are queued locally are consumed by the Service Component. To simplify the implementation, service requests are implemented using UDP.

In order to test the performances of our scheme, a system containing distributed copies of the above described server component, 24 copies of this server were deployed on 12 Dual PowerPC machines, each of which has a 2.2GHz 64-bit CPU, with 4GB of RAM. These machines are part of a Blade Center. Therefore CT in our setting is negligible and implicitly accounted by C. A client was created to generate all requests, to collect all answers from the servers and to produce logs that were processed to create the system statistics. The average service time in all test runs was set up to be around 165 milliseconds.

In order to compare our method to distributed load balancing alternatives, which employ dedicated monitoring, we implemented a scheme called Peer Monitoring (PM). This scheme is depicted in Figure 6. In PM, the server actively queries k peer servers for their load (where k is a parameter of the scheme), using a dedicated monitoring algorithm, and assigns the job to the least loaded server among these peers. Essentially, PM is a straightforward distributed version of CM, in which each server acts in two capacities: a regular server and a centralized dedicated monitor for a neighborhood of size k. From testing the service time for different values of d (number of peers to query) we conclude that the overhead efficiency ratio C was approximately 0.003 in our setting.

For each d value and each load parameter (load is achieved by tuning the average inter arrival time at the client) we performed several runs of over 20000

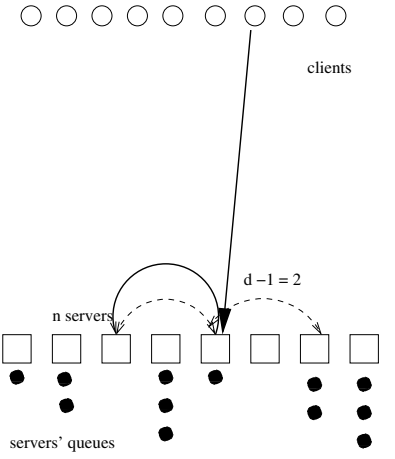

Fig. 6. A basic distributed scheme with explicit monitoring: Peer Monitoring

job requests each, and computed the average response time. The average response time for five load values as a function of d is depicted in Figure 4. One can see that the basic hop count scheme performs better than the PM scheme. This is because the Basic heuristic incurs less management overhead on the server, since the data needed to perform management decision is conveyed via the job request packet, and not by direct load queries. It is worth noting that the results of the Basic heuristic are quite similar in the low and medium load (when $d \geq 3$), this is due to the fact that in such loads, the probability of finding an idle server within a few hops is high.

Figure 5 depicts the performance of the two adaptive heuristics with respect to Basic PM, where each is assumed to use the *best* parameters for that load. One can see that Self Adaptive Heuristic I outperforms both the PM scheme as well as the basic hop count scheme. The second Adaptive Heuristic does not perform as well. This indicates that this heuristic is more sensitive to parameter tuning. Note again that neither of the adaptive heuristics uses any tuning, and the process is self adaptable to the current load. Also note that even for a relatively small set of servers, the load balancing performs very well for all load values.

5 Related Work

Load balancing mechanisms were extensively studied in a variety of contexts over the last twenty years. Proliferation of highly dynamic research and industrial efforts caused the "load balancing" term to become a somewhat overloaded concept. Load balancing (sometimes referred to as "load sharing") may have slightly different meanings when different computer science communities are concerned. Interested reader are encouraged to read [4] for the load balancing taxonomy and fully specified positioning of our monitoring overhead-aware approach to load balancing.

The cost of monitoring is an integral part of the overall cost of the load balancing algorithm. Clearly, a trade-off exists between the quality of monitoring and the cost of acquiring the monitoring data. On the one hand, the more updated the monitoring data is, the higher is the total quality of load balancing [5,6,7,2]. On the other hand, since monitoring takes small but non-zero amount of computational and communication resources, its inherent cost becomes a limiting factor for scalability. Although this trade-off has been long noticed, no formal study of it was performed until recently [2]. Some insightful empiric studies have been carried out, though [7,5].

Our approach is fully distributed and is close in spirit to *Direct Neighbor Repeated (DNR)* policy [8] used in *diffusive*. Our solution is different from the DNR-like strategies since it does not use explicit monitoring to evaluate the load distribution in the neighborhood, but rather collect the monitoring information *en-route*, while a new job travels a few hops before settling for execution. In addition, we use a randomly selected dynamic logical neighborhood, while the DNR-like solutions use a physical static neighborhood.

A novel approach to Load Evaluation and Task Migration sub-policies was presented in [9]. In this solution a unified cost model for heterogenous resources was presented. Using this cost model, the "marginal cost" of adding a task to a given destination is computed. A task is dispatched to the destination which minimizes the total marginal cost. In a sense, it is a variation of a greedy algorithm. However, thanks to its sophisticated cost model, it outperforms simple greedy algorithms.

Another innovative approach that combines a threshold-based approach with the greedy strategy in an interesting way was presented in [10]. The primary goal of this work is to achieve an autonomic adaptable strategy for peer-to-peer overlay maintenance with QoS guarantees in spite of possible server crashes.

In contrast to other load balancing mechanisms, our solution explicitly takes monitoring costs – which is an integral part of any load balancing solution – into account following the footsteps of [2]. In contrast to [2], in this work we concentrate on a fully distributed setting. The distribution, however, brings about a problem of staleness of the load data acquired at different points in time as new job requests travel among different randomly selected servers. It turns out that closed form solution of the average response time in presence of staleness is unlikely even for the Poisson traffic model [5,2]. We, therefore, use mainly simulations and emulation to study the behavior of our proposed heuristics.

6 Conclusions and Future Work

The ability to quantify the benefits of a system management tool and the overhead associated with it is an important step toward developing cost effective self enabled systems. This paper provides one building block in the strive to rigorously quantify the effectiveness of management systems. We consider a distributed service setting where the goal is to minimize the total average time required to provide the service to the customers. Much of the overhead associated

with load balancing systems in such a setting is due to the need to monitor the load on the different servers in order to assign job requests to sub-utilized servers.

In order to understand the exact benefit of this explicit monitoring, we compare the benefit of explicit monitoring systems with the best possible "no monitoring" solution. Note that a simple random assignments of servers to jobs in our setting results in an average waiting time of $\frac{1}{1-\lambda}$, which yields a total time of 2.5 times the service time in 80% load. Our schemes reduces this factor considerably (to $1.2 - 1.5$).

These results indicate the importance of identifying the exact cost and benefit associated with system and network management. The same methods could be used to understand this tradeoff in different networking settings (such as routing) that involve dissemination of local information through the network.

References

1. Srisuresh, P., Gan, D.: Load Sharing using IP Network Address Translation (LSNAT) (August 1998)
2. Breitgand, D., Cohen, R., Nahir, A., Raz, D.: Cost aware adaptive load sharing. In: IWSOS 2007. The 2nd International Workshop on Self-Organizing Systems, English Lake District, UK (September 2007)
3. Mitzenmacher, M.: The power of two choices in randomized load balancing. IEEE Transactions on Parallel and Distributed Systems 12(10), 1094–1104 (2001)
4. Breitgand, D., Nahir, A., Raz, D.: To know or not to know: on the needed amount of management information, Tech. Rep. H-0242, IBM T.J. Watson Research Center (2006)
5. Mitzenmacher, M.: How useful is old information? IEEE Transactions on Parallel and Distributed Systems 11(1), 6–20 (2000)
6. Hui, C.-C., Chanson, S.T.: Improved Strategies for Dynamic Load Balancing. IEEE Concurrency 7(3), 58–67 (1999)
7. Othman, O., Balasubramanian, J., Schmidt, D.C.: Performance Evaluation of an Adaptive Middleware Load Balancing and Monitoring Service. In: 24th IEEE International Conference on Distributed Computing Systems (ICDCS), Tokyo, Japan (May 2004)
8. Corradi, A., Leonardi, L., Zambonelli, F.: On the Effectiveness of Different Diffusive Load Balancing Policies in Dynamic Applications. IEEE Concurrency 7(1), 22–31 (1999)
9. Amir, Y., Awerbuch, B., Barak, A., Borgstrom, R.S., Keren, A.: An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster. IEEE Transactions on Parallel and Distributed Systems 11(7), 760–768 (2000)
10. Adam, C., Stadler, R.: Adaptable Server Clusters with QoS Objectives. In: IM 2005. 9th IFIP/IEEE International Symposium on Integrated Network Management, Nice, France (May 2005)