Interpreting Stale Load Information

Michael Dahlin*
Department of Computer Sciences
University of Texas at Austin
dahlin@cs.utexas.edu

Abstract

In this paper we examine the problem of balancing load in a large-scale distributed system when information about server loads may be stale. It is well known that sending each request to the machine with the apparent lowest load can behave badly in such systems, yet this technique is common in practice. Other systems use round-robin or random selection algorithms that entirely ignore load information or that only use a small subset of the load information. Rather than risk extremely bad performance on one hand or ignore the chance to use load information to improve performance on the other, we develop strategies that interpret load information based on its age. Through simulation, we examine several simple algorithms that use such load interpretation strategies under a range of workloads. Our experiments suggest that by properly interpreting load information, systems can (1) match the performance of the most aggressive algorithms when load information is fresh relative to the job arrival rate, (2) outperform the best of the other algorithms we examine by as much as 60% when information is moderately old, (3) significantly outperform random load distribution when information is older still, and (4) avoid pathological behavior even when information is extremely old.

1. Introduction

When balancing load in a distributed system, it is well known that the strategy of sending each request to the least-loaded machine can behave badly if load information is old [12, 20, 23, 27]. In such systems a "herd effect" often develops, and machines that appear to be underutilized quickly become overloaded because everyone sends their requests to those machines until new load information is propagated. To combat this problem, some systems adopt randomized strategies that ignore load information or that only use a small subset of load information, but these systems may give up the opportunity to avoid heavily loaded machines.

Load balancing with stale information is becoming an increasingly important problem for distributed operating systems. Many recent experimental operating systems have included process migration facilities [2, 6, 10, 18, 19, 25, 26, 28, 29, 33] and it is now common for workstation clusters to include production load sharing programs such as LSF [34] or DQS [11]. In addition, many network DNS servers, routers, and switches include the ability to multiplex incoming requests among equivalent servers [1, 5, 9], and several run-time systems for distributed parallel computing on clusters or metacomputers include modules to balance requests among nodes [13, 16]. Server load may also be combined with locality information for wide area network (WAN) information systems such as selecting an HTTP server or cache [14, 24, 31]. As such systems include larger numbers of nodes or the distance between nodes increases, it becomes more expensive to distribute up-to-date load information. Thus, it is important for such systems to make the best use of old information.

This paper attempts to systematically develop algorithms for using old information. The core idea is to use not only each server's last reported load information (L_i), but also to use the age of that information (T) and an estimate of the rate at which new jobs arrive to change that information (λ). For example, under a *periodic update* model of load information [23] that updates server load information every T units of time, clients using our approach calculate the fraction of requests they should send to each server in order

*This work was supported in part by an NSF CISE grant (CDA-9624082) and CAREER grant (9733842) and grants from Novell and Sun.

to equalize the load across servers by the end of the epoch of length T. Then, for each new request during an epoch, clients randomly choose a server according to these probabilities.

In this paper, we devise load interpretation (LI) algorithms by analyzing the relevant queuing systems. We then evaluate these algorithms via simulation under a range of load information models and workloads. For our LI algorithms, if load information is fresh (e.g., T or λ or both are small), then the algorithms tend to send requests to machines that recently reported low load, and the algorithms match the performance of aggressive algorithms while exceeding the performance of algorithms that use random subsets of load information or pure random algorithms that use no load information at all. Conversely, if load information is stale, the LI algorithms tend to distribute jobs uniformly across servers and thus perform as well as randomized algorithms and dramatically better than algorithms that naively use load information. Finally, for load information of modest age, the LI algorithms outperform the best of the other algorithms we examine by as much 60%.

Other algorithms that attempt to cope with stale load information, such as those proposed by Mitzenmacher [23], have the added benefit that by restricting the amount of load information that clients may consider when dispatching jobs, they may reduce the amount of load information that must be sent across the network. We examine variations of the LI algorithms that base their decisions on similarly reduced information. We conclude that even with severely restricted information, the algorithms that use LI can outperform those that do not. Furthermore, modest amounts of load information allow the LI algorithms to achieve nearly their full performance. Thus, LI decouples the question of how much load information should be used from the question of how to interpret that information.

The primary disadvantage of our approach is that it requires clients to estimate or be told the job arrival rate (λ) and the age of load information (T). If this information is not available, or if clients misestimate these values, our algorithms can have poor performance. We note, however, that although other algorithms that make use of stale load information do not explicitly track these factors, those algorithm do implicitly assume that these parameters fall within the range of values for which load information can be considered "fresh;" if the parameters fall outside of this range, those algorithms can perform quite badly. Conversely, because our algorithms explicitly include these parameters, they gracefully degrade as information becomes relatively more stale.

The rest of this paper proceeds as follows. Section 2 describes related work with a particular emphasis on Mitzenmacher's recent study [23], on which we base much of our methodology and several of our system models. Section 3 introduces our models of old information and Section 4 describes the load interpretation algorithms we use. Section 5 contains our experimental evaluation of the algorithms, and Section 6 summarizes our conclusions.

2. Related work

Awerbuch et. al [3] examined load balancing with very limited information. However, their model differs considerably from ours. In particular, they focus on the task of selecting a good server for a job when other jobs are placed by an adversary. In our model, jobs are placed by entities that act in their own best interest but that do not seek to interfere with one another. This difference allows us to more aggressively use past information to predict the future.

A number of theoretical studies [4, 8, 12, 17, 22, 30] have suggested that load balancing algorithms can often be quite effective even if the amount of information examined is severely restricted. We explore how to combine this idea with LI in Section 5.6.

Harchol-Balter et. al [15] have found that when job sizes are highly-variable, the greedy approach of sending jobs to the servers with the least remaining work can behave poorly, even when current loads are known precisely. They develop a job-size based policy with better performance. Extending the LI approach to such workloads remains important future work.

Several systems have used the heuristic of weighing recent information more heavily than old information. For example, the Smart Clients prototype [32] distributed network requests across a group of servers using such a heuristic. Additionally, a common technique in process migration facilities is to use an exponentially decaying average of past load to predict future load on a machine (e.g., $Load_{predict} = Load_{old} * k + Load_{current} * (1-k)$ for some $k < 1$). Unfortunately, the algorithms used by these systems are somewhat *ad hoc* and it is not clear under what circumstances to use these algorithms or how to set their constants. A goal of our study is to construct a systematic framework for using old load information.

Several studies have examined the behavior of load balancing with old information [20, 21, 23, 27]. These studies have used combinations of three basic techniques to cope with stale information: (1) considering only a subset of randomly selected servers and choosing from among them rather than from among all servers; (2) using a threshold to classify machines as either lightly-loaded or heavily-loaded and choosing randomly from the lightly-loaded group; and (3) combining "sender-driven" job placement with "receiver-driven" [27] rebalancing in which lightly-loaded or idle nodes remove jobs from heavily loaded nodes. In this paper, we examine the first two options in detail and compare them with LI. Examining the performance of LI-based algorithms in comparison with and combination with receiver-driven algorithms is important future work.

Our study most closely resembles Mitzenmacher's work [23]. Mitzenmacher examined a system in which arriving jobs are sent to one of several servers based on stale information about the servers' loads. The goal in such a system is to minimize average response time. He examined a family of algorithms that make each server choice from small random subsets of the servers to avoid the "herd effect" that can cause systems to exhibit poor behavior when clients chase the apparently least loaded server. Under Mitzenmacher's algorithm, if there are n servers, instead of sending a request to the least loaded of the n servers, a client randomly selects a subset of size k of the servers, and sends its request to the least loaded server from that subset. Note that when $k = 1$, this algorithm is equivalent to uniform random selection without load information and that when $k = n$ it is equivalent to sending each request to the apparently least loaded server. In addition to the formulating these k-subset algorithms as a solution to this problem, Mitzenmacher uses a fluid limit approach to develop analytic models for these systems for the case when ($n \to \infty$); however, the primary results in the study come from simulating the queuing systems, and we follow a similar simulation methodology here.

Mitzenmacher concludes that the $k = 2$ version of the algorithm is a good choice in most situations. He finds that it seldom performs significantly worse and generally performs significantly better than the more aggressive algorithms (e.g., $k = n$ or even the modestly aggressive $k = 3$ algorithm) and that $k = 2$ outperforms the uniform random ($k = 1$) algorithm for a wide range of update frequencies.

We believe, however, that this approach still has drawbacks. In particular, we note that as T—the update frequency of load information—changes, the optimal value of k also changes. For example, under Mitzenmacher's periodic update model and one sample workload he examines, $k = 100$ outperforms $k = 2$ by more than 70% when $T < 0.1$[1] but $k = 2$ quickly becomes much better than $k = 100$ for larger values of T. Similarly, although $k = 2$ outperforms $k = 1$ when $T < 36$ for such a workload, the reverse is true for larger values of T. For example, when $T = 100$, the $k = 1$ algorithm is a factor of 2 better than the $k = 2$ variation.

3. Models of old information

There are several reasonable ways to model a delay from when load information is sampled to when a decision is made to when the job under consideration arrives at its server. Different models will be appropriate for different practical systems, and Mitzenmacher found significant differences in system behavior among models [23]. We therefore examine performance under three models so that we can understand our results under a wide range of situations and so that we can compare our results directly to those in the literature. We take the first two models, *periodic update* and *continuous update*, from Mitzenmacher's study.[2] Our third model, *update-on-access*, abstracts some additional systems of practical interest. We describe these models in more detail below.

3.1. Periodic and continuous update

Mitzenmacher's periodic update and continuous update models can be visualized as variations of a bulletin board model. Under the *periodic update* model, we imagine that every T units of time a bulletin board that is visible to all arriving jobs is updated to reflect the current load of all servers. The period between bulletin board updates is termed a phase. Load information will thus be accurate at the beginning of a phase and may grow increasingly inaccurate as the phase progresses.

Under the *continuous update* model, the bulletin board is constantly updated with load information, but on average the board state is T units of time behind the true system state. Each request thus bases its decisions on the state of the system on average T units of time earlier. Mitzenmacher finds that the probability distribution of T had a significant impact on the effectiveness of different algorithms. For a given average delay T, distributions with high variance in which some requests see newer information and others see older information outperform distributions with less variance where all jobs see data that are about T units of time old.

[1] Throughout this paper we express T in units of the average job service time. Similarly, because λ represents the ratio of the service time to the per-node inter-arrival time, the inter-arrival time, $\frac{1}{\lambda}$ is also expressed in units of the job service time.

[2] Mitzenmacher found a third model, *individual updates* to have similar behavior to the periodic update model, so we omit analysis of this model for compactness.

Note that the real systems abstracted by these models would typically not include a centralized bulletin board. The periodic model could represent, for instance, a system that periodically gathers load information from all servers and then multicasts it to clients. The continuous update model could represent a system where an arriving job probes the servers for load information and then chooses a server but where there is a delay T due to network latency and transfer time from when the servers send their load information to when the client's job arrives at its destination server.

3.2. Update-on-access

The final model we examine was not examined by Mitzenmacher. In our *update-on-access* model, we explicitly model separate clients sending requests to the servers, and different clients may have different views of the system load. In particular, when a client sends a request to a server, we assume that the server replies with a message that contains the system's current load values, and that snapshot of system load may be used by the client's next request. In such a system, the average load update delay, T, is equal to a client's inter-request time. Thus, the update-on-access model assumes that jobs sent by active clients will have a fresher picture of load than jobs sent by inactive clients.

We consider this model because it may be applicable for problems such as the server selection problem on the Internet [14, 24] where it may be prohibitively expensive to maintain load information at clients that are not actively using a service, but where it may be possible for clients to maintain good pictures of server load while they are actively using a service. Furthermore, we hypothesize that if a system exhibits bursty access patterns, it may be able to perform good load balancing even though a node's load information is, on average, quite stale.

4. Algorithms

In this section we describe our algorithms for load balancing, which work by interpreting load information in the context of its age. We first describe the basic algorithm under the periodic update model and then describe a more aggressive algorithm under the same model. Finally, we describe minor variations of the algorithms to adapt them for the continuous update and update-on-access models.

In general, our algorithms for interpreting load information follow two principles that distinguish them from previous algorithms. First, we consider the magnitude of imbalance between nodes, not just the nodes' ranks. Second, we modify our interpretation of information based on its age and the arrival rate of requests in the system to account for expected changes to system state over time.

In the descriptions below, we use the following notation:

T	Average age of the load information in units of per-job service time
n	Number of servers
λ	Average per-server arrival rate as a fraction of a server's maximum throughput
L_i	Reported load (queue length) on server i
L_{tot}	$\sum_{i=0}^{n-1} L_i$
$arrive_T$	Expected number of request arrivals per phase
P_i	Probability that an arriving request will be sent to server i

4.1. Algorithms for periodic update model

The during a phase of length T, $arrive_T = (T * \lambda * n)$ requests will arrive in the system. The goal of the *Basic Load Interpretation* (Basic LI) algorithm is to determine what fraction of those requests should be sent to each server in order to balance load (represented as server queue length) so that the sum of the jobs at the servers at the start of the phase plus the jobs that arrive during the phase are equal across all servers.[3] So, if we begin with L_{tot} jobs at the servers and L_i jobs at server i, the probability P_i that we should send an arriving job to server i is

$$P_i = \frac{\frac{L_{tot}+arrive_T}{n} - L_i}{arrive_T} \quad (\text{assumes } \forall i (\frac{L_{tot}+arrive_T}{n} > L_i)) \tag{1}$$

The first term in the numerator is the number of jobs that should end up at each server to evenly divide the incoming jobs plus the current jobs. The second term in the numerator is the jobs already at server i. So the numerator is the number of jobs that should be sent to server i during this phase. The denominator is the total number of jobs that are expected to arrive during the phase.

[3] Notice that we make the simplifying assumption that the departure rate is the same at all servers so that we can ignore the effect of departing jobs on the relative server queue lengths. This assumption will be correct if all servers are always busy, but it will be incorrect if some servers are idle at any time during the phase. This assumption can be justified because we are primarily concerned that our algorithms work well when the system is heavily loaded, and in that case queues will seldom be empty. The impact of this simplification is that for lightly-loaded systems, we will overestimate the queue length at lightly-loaded machines and send too few requests to them. In such a case, our probability distribution will be somewhat more uniform across servers than would be ideal, and our algorithms will not be as aggressive as they could be. Our experiments suggest that this simplification has little performance impact.

Note that if $\frac{L_{tot}+arrive_T}{n} < L_i$ for any i, then the phase is too short to completely equalize the load. In that case, we want to place the $arrive_T$ requests in the l least loaded buckets to even things out as well as we can. To generalize Equation 1 to include this case, we first calculate l, the number of least-loaded servers to which jobs should be sent during the phase. Without loss of generality, assume that the servers are sorted by load and numbered $0 \ldots n-1$ with 0 being the least-loaded server. Then l is the maximum value between 1 and n, inclusive, such that:

$\sum_{i=0}^{l-1}(L_{l-1} - L_i) < arrive_T$

Now, we wish to send enough requests to servers $(0 \ldots l-1)$ to bring them up to the load of server $l-1$, and then evenly distribute the remaining requests from the phase evenly across the l least-loaded servers. Thus,

$$P_i = \begin{cases} \frac{(L_{l-1}-L_i)+\frac{arrive_T - \sum_{j=0}^{l-1}(L_{l-1}-L_j)}{L}}{arrive_T} & \text{if } i < l \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Note that this equation reduces to Equation 1 for $l = n$.

Thus the Basic LI algorithm is to send each arriving request to server i with probability P_i as calculated above for the current phase.

4.1.1. More aggressive algorithm

The above algorithm seems sub-optimal in the following sense: it tries to equalize the load across servers by the end of a phase. Thus, if the phase is long, the system may spend a long time with significantly unbalanced server load. A more aggressive algorithm might attempt to subdivide the phase and use the first part of the phase to bring all machines to an even state and then distribute requests uniformly across servers for the rest of the phase.

Our aggressive algorithm works as follows: without loss of generality assume that the servers have been sorted by L_i so that machine i is the ith least loaded server and set $L_n = \infty$ as a sentinel value. Then, subdivide the phase into n intervals. During interval j (with $j = 0 \ldots n-1$), evenly distribute arrivals across machines $0 \ldots j$ to bring their loads up to L_{j+1}. Thus, subinterval j is of length $T_j = \frac{j*(L_{j+1}-L_j)}{\lambda n}$, and P_{ij}, the probability that an arriving request should be sent to machine i during subinterval j, is:

$$P_{ij} = \begin{cases} \frac{1}{j+1} & \text{if } i \le j \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

4.2. Algorithms for other update models

Adapting the Basic LI algorithm to the continuous update or update-on-access model is simple. We use Equation 2 to calculate the probabilities P_i for sending incoming requests to each server. The only difference is that for the periodic update model this calculation is based on the L_i estimates that hold during the entire phase, but under the new models the L_i estimates may change with each request. P_i can now be thought of as a current estimate of the instantaneous rate at which requests should be sent to each server.

Adapting the Aggressive LI algorithm is more problematic. We use Equation 3 to calculate the P_{ij} values based on the current L_i array. However, under the continuous update model, we are effectively always "at the end of a phase" in that the information is T units of time old. And, although the aggressive algorithm is more aggressive than the basic algorithm during the early subintervals of a phase (e.g., j near 0), it is less aggressive during later subintervals (e.g., j near n). Thus, the "aggressive" algorithm may actually be less aggressive than the basic algorithm under these update models when T is large.

5. Evaluation

In this section we evaluate the algorithms under a range of update models and workloads. Our primary methodology is to simulate the queuing systems. We model task arrivals as a Poisson stream of rate λn for a collection of n servers. When a task arrives, we send it to one of the server queues based on the algorithm under study. Server queues follow a first-in-first-out (FIFO) discipline. We select default system parameters to match those used in Mitzenmacher's study [23] to facilitate direct comparison of the algorithms. In particular, unless otherwise noted, $\lambda = 0.9$ and $n = 100$, and we assume that each server has a service rate of 1 and the service time for each task is exponentially distributed with a mean time of 1.

We run each simulation for 500,000 job arrivals unless otherwise noted, and we use the first $\frac{1}{4}$ of the jobs to bring the system to a steady-state. We then measure average job response time for jobs $\frac{njobs}{4} \ldots \frac{3njobs}{4}$. We run each experiment at least 10 times using different random seeds, and we show the 90% confidence interval for each data point. We validated the simulator by comparing our k-subset results to those published by Mitzenmacher [23]. In addition, the results for the Aggressive LI algorithm under the bulletin board model have been verified by Christenson [7].

We initially examine the Basic LI and Aggressive LI algorithms under the periodic update, continuous update, and update-on-access models and compare their performance to the k-subset algorithms examined by

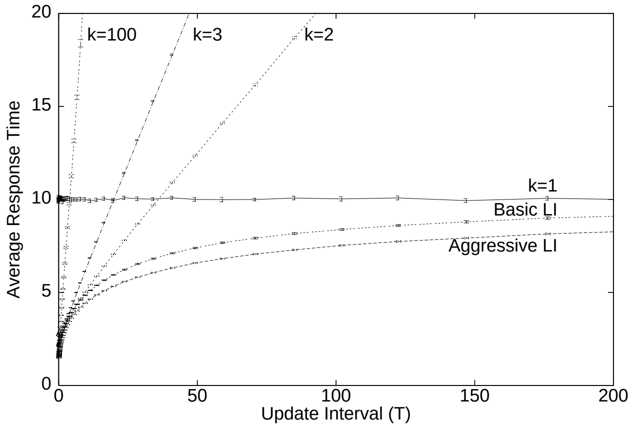


Figure 1. Service time v. update delay for periodic update model ($\lambda = 0.9$, $n = 100$.)

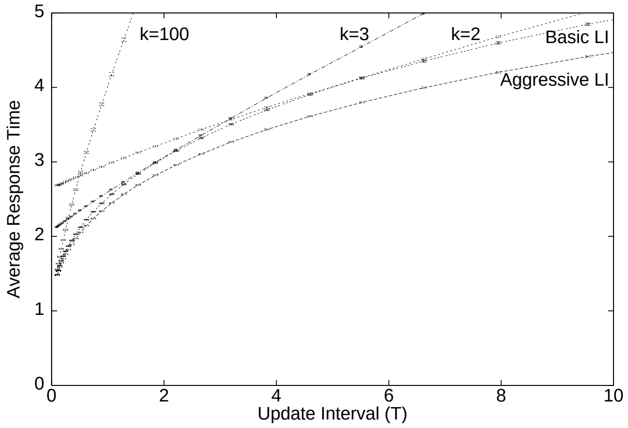


Figure 2. Detail of service time v. update delay for periodic update model ($\lambda = 0.9$, $n = 100$.)

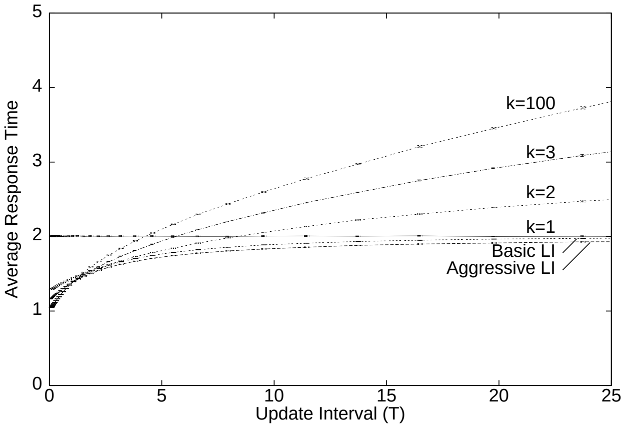


Figure 3. Service time v. update delay for periodic update model ($\lambda = 0.5$, $n = 100$.)

Mitzenmacher. We then explore three key questions for the LI algorithms: (1) What is the impact of bursty arrival patterns? (2) What is the impact of misestimating the system arrival rate? (3) What is the impact of limiting the amount of load information available to the algorithms?

5.1. Periodic update model

Figure 1 shows system performance under the periodic update model for the default parameters. The performance of the LI algorithms is good over a wide range of update intervals. When T is large, the LI algorithms do not suffer the poor performance that the k-subset algorithms encounter for any $k > 1$. In fact, the LI algorithms maintain a measurable advantage over the oblivious random algorithm ($k = 1$) even for large values of T. For example, when $T = 200$, Basic LI outperforms the oblivious algorithm by 9% and Aggressive LI outperforms the oblivious algorithm by 17%. For more modest values of T, the advantages are larger. For example, at $T = 40$, Aggressive LI is 60% faster than any of the k-subset algorithms and Basic LI is 41% faster than any of the k-subset algorithms.

Figure 2 details the performance of the algorithms for small values of T. Aggressive LI outperforms all other algorithms by at least a few percent down to the smallest value of T we examined ($T = 0.1$). Basic LI is generally better than and always at least as good as any k-subset algorithm over this range of T.

Figure 3 shows the performance of the system under a workload with a lighter load ($\lambda = 0.5$) than our default. When load is lighter, the need for load balancing is less pronounced and the gains by any algorithm over random are more modest. When information is fresh, the algorithms can perform nearly a factor of two better than the oblivious algorithm. When information is stale, the performance of the k-subset algorithms is not nearly as bad as it was for the heavier load, although they do exhibit poor behavior compared to the oblivious algorithm for large T. Over the entire range of staleness examined ($0.1 \leq T \leq 200$), the Basic LI and Aggressive LI algorithms perform as well as or better than the best k-subset or oblivious algorithm.

Figure 4 shows the performance of the system with $n = 8$ servers rather than the standard $n = 100$. The results are qualitatively similar to the results for the standard $n = 100$ case.

As noted in Section 2, another common strategy for dealing with old information is to use a *threshold* value to classify machines as either lightly or heavily loaded and then to choose randomly from the lightly-loaded group [12, 20, 21, 27]. Figure 5 shows the performance of this approach for different thresholds. These data suggest that changing the threshold value acts much like changing the k value in the standard k-subset algorithm, allowing the system to vary how aggressively it interprets the information. As with the k-subset algo-

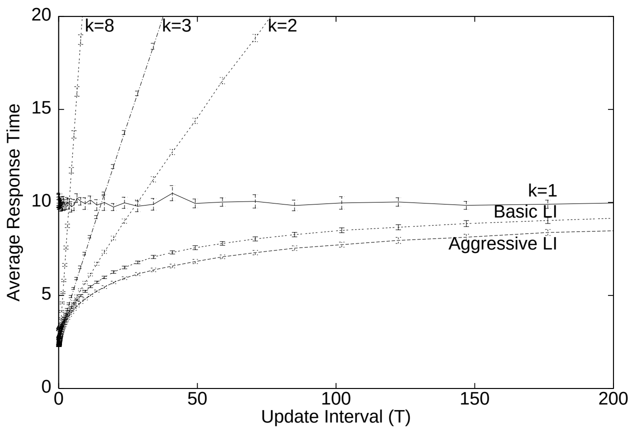


Figure 4. Service time v. update delay for periodic update model ($\lambda = 0.9$, $n = 8$.)

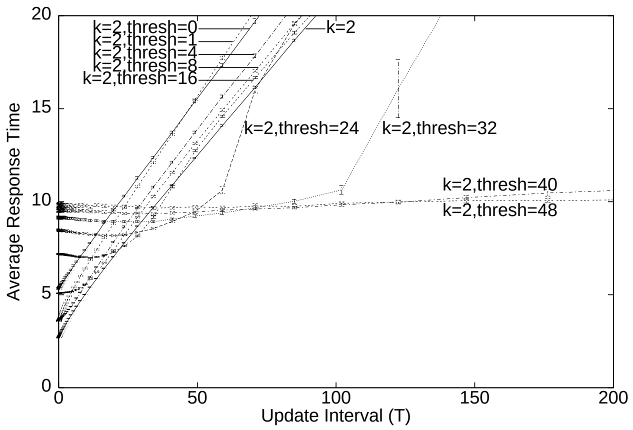


Figure 5. Service time v. update delay for the threshold algorithm.

rithm, the best parameter value depends on the age of the information. Mirchandaney also noted this strong sensitivity of the optimal threshold value on the age of the load information [20]. Through the rest of this paper we will focus on comparing the LI algorithms with the k-subset algorithms.

5.2. Continuous update model

Figure 6 shows the performance of the algorithms under the continuous update model. Because system behavior depends on the distribution of the delay parameter, we show results for four distributions of delay, all with average value T. In order of increasing variation, they are: constant(T), uniform($\frac{T}{2}$ to $\frac{3T}{2}$), uniform(0 to $2T$), and exponential(T). As the earlier discussion suggests, the Aggressive LI algorithm is actually less aggressive than the Basic LI algorithm, and Basic generally outperforms Aggressive for this model. We will therefore focus on the Basic LI algorithm.

Mitzenmacher notes that for a given T, the k-subset algorithms' performance improves for distributions that contain a mix of more recent and older information. This relationship seems present but less pronounced for the LI algorithms. As a result, as the distribution's variability increases, the advantage of LI over the k-subset algorithms shrinks. Thus, Basic LI seems a clear choice for the constant and uniform T distributions: for any value of T, its performance is as good as any of the k-subset algorithms and for any given k-subset algorithm there is some range of T where Basic LI's performance is significantly better.

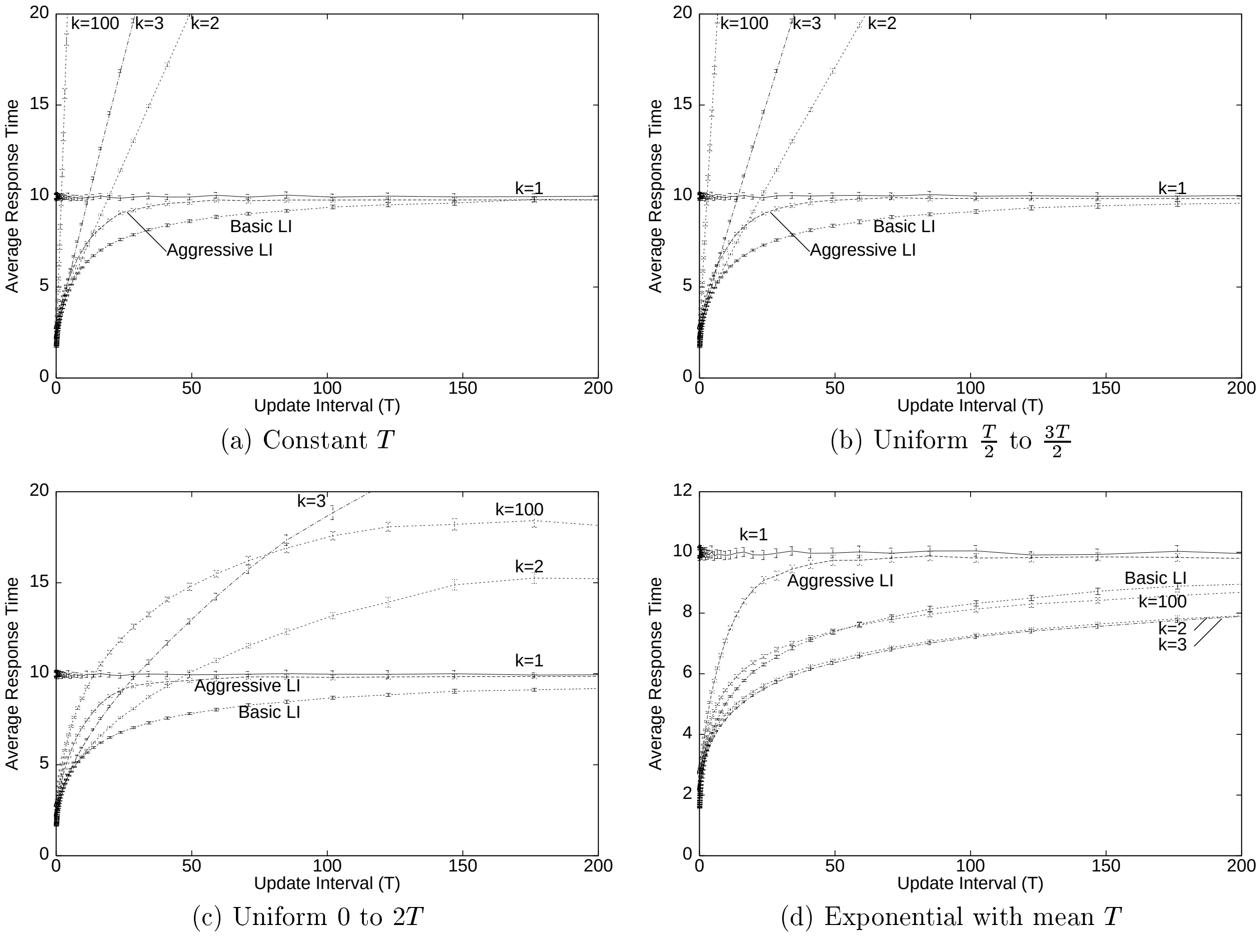
For the exponential distribution of T, however, the k-subset algorithms enjoy an advantage of up to 16% over Basic LI. Figure 7 tests the hypothesis that the relatively poor performance of Basic LI in this situation is because the algorithm calculates P_i using the expected value of T whereas each individual request may see significantly different values of T. In this figure, T still varies according to the specified distribution, but rather than knowing the average value of T, each request knows the value of T that holds for that request, and the algorithm calculates its P_i vector using this more certain information. Compared to the performance in Figure 6, this extra information improves performance for each distribution of T, and the improvement becomes more pronounced for distributions with more variation. From this we conclude that good estimates of the delay between when load information is gathered and when a request will arrive at a server are important for getting the best performance from the LI algorithms.

5.3. Update-on-access model

Figure 8 shows performance for the update-on-access model. In this model each load-generating client uses the load information gathered after sending one request to decide where to send the next request. Thus, T equals the average per-client inter-request time. To vary T for a fixed total arrival rate, we simply vary the number of clients from which the fixed aggregate rate of requests are issued: $nclients = \frac{T}{\lambda nservers}$. To ensure that the realized load information age is close to the desired age for each client, we increase the number of simulated arrivals for experiments that use large numbers of clients so that each client launches at least 1,000 jobs.

For this model, all of the algorithms perform reasonably well. It appears that the per-client updates desynchronize the clients enough to reduce the herd effect. The Basic LI algorithm outperforms all of the others and provides a modest speedup over a wide range of update intervals.

5.4. Bursty arrivals



(a) Constant T (b) Uniform $\frac{T}{2}$ to $\frac{3T}{2}$

(c) Uniform 0 to $2T$ (d) Exponential with mean T

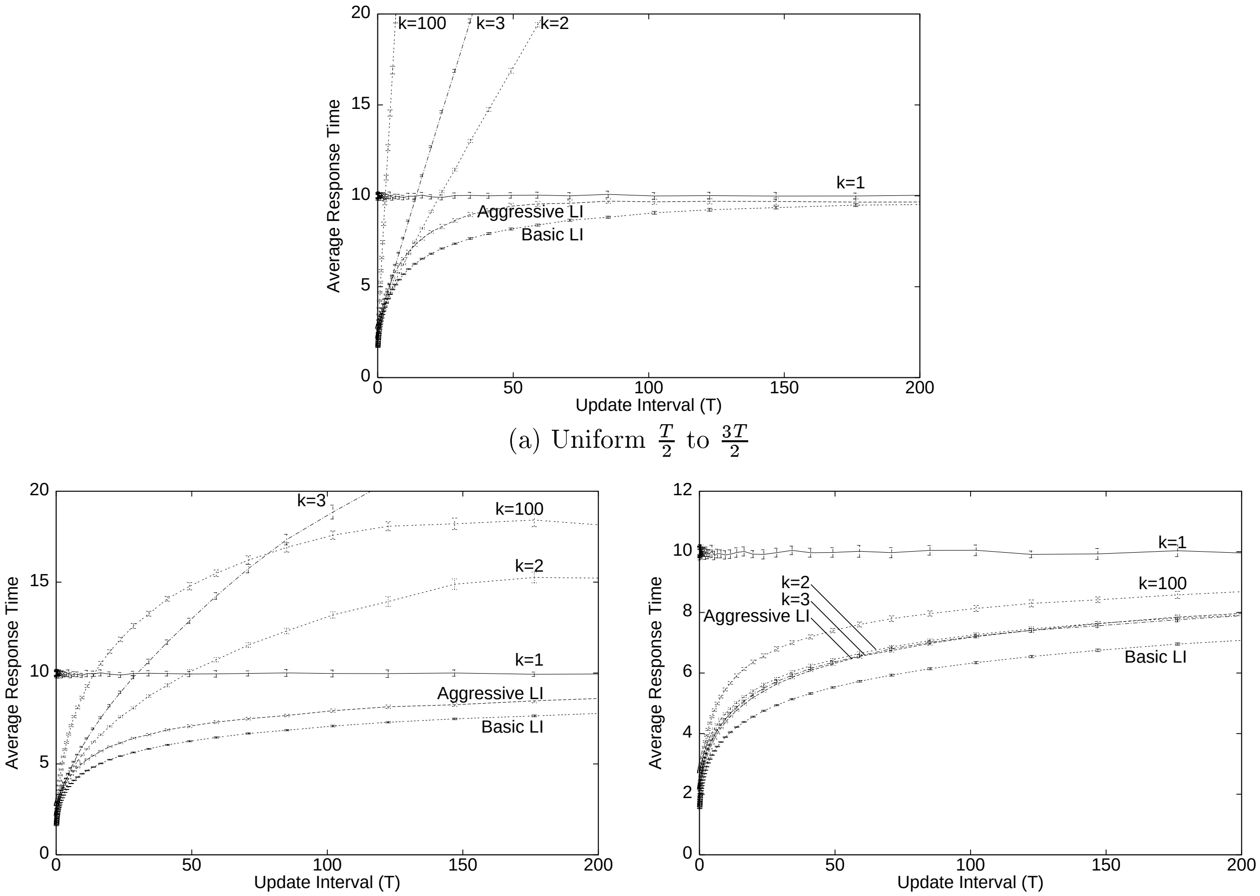
Figure 6. Service time v. update delay for continuous update model when clients only know T, the average delay. (a) – (d) show result for different distributions of delay around the average T.

Figure 9 shows performance under a bursty-arrival version of the update-on-access model. As with the standard update-on-access model, each client uses the server loads discovered during one request to route the next one. To generate our bursty-arrivals workload, rather than assume that each client produces exponentially-distributed arrivals, we assume that a client whose average inter-request time is T produces a burst of b requests separated by exponential($\frac{T}{b}$) units of time, with the bursts separated by exponential(Tb) units of time. For Figure 9, $b = 10$.

The bursty workload significantly increases the performance of all of the algorithms that use server load compared to the oblivious algorithm. Although over time, a client's picture of server load is on average T units of time old, an average request sees a much fresher picture of the L_i vector. This suggests that it may often be possible to significantly outperform the oblivious strategy even for challenging workloads such as internet server selection [14, 24] where information will likely be old on average, but where a client's requests to a service are bursty. Once again, the Basic LI algorithm is the best or tied for the best over the entire range of T examined ($0.1 \leq T \leq 200$).

5.5. Impact of imprecise information

The primary drawback to the LI algorithms is that they require good estimates of T and λ. Subsection 5.2 examined the impact of uncertainty about T. In this subsection, we examine what happens when the estimate of λ is incorrect. We believe that servers supporting the LI algorithms would be equipped to inform clients both of their current load and of the arrival rate of requests they anticipate. For example, a server might report the arrival rate it had seen over some recent period of time, or it might report the maximum request rate it anticipates encountering. However, it may be difficult for some systems to accurately predict future request patterns based on history.



(a) Uniform $\frac{T}{2}$ to $\frac{3T}{2}$

(b) Uniform 0 to $2T$

(c) Exponential with mean T

Figure 7. Service time v. update delay for continuous update model when clients know the age of information actually encountered for each request. (a) – (c) show result for different distributions of delay around the average T.

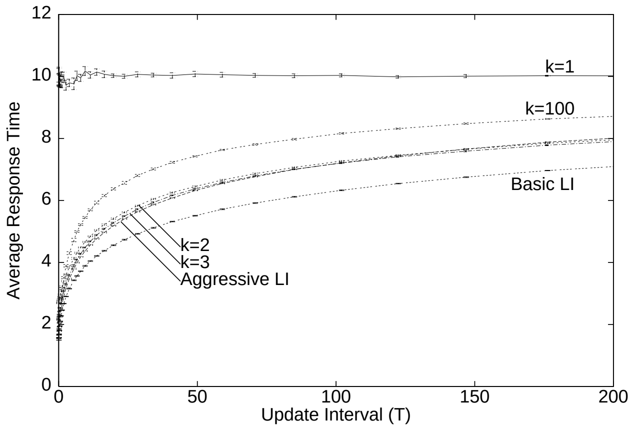


Figure 8. Service time v. update delay for update-on-access model.

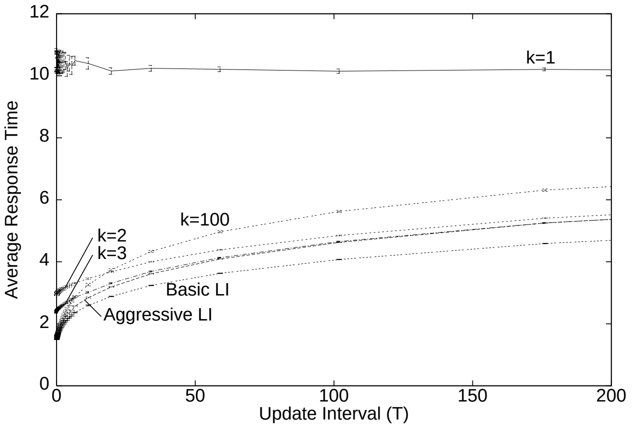


Figure 9. Service time v. update delay for update-on-access model under bursty workload ($b = 10$).

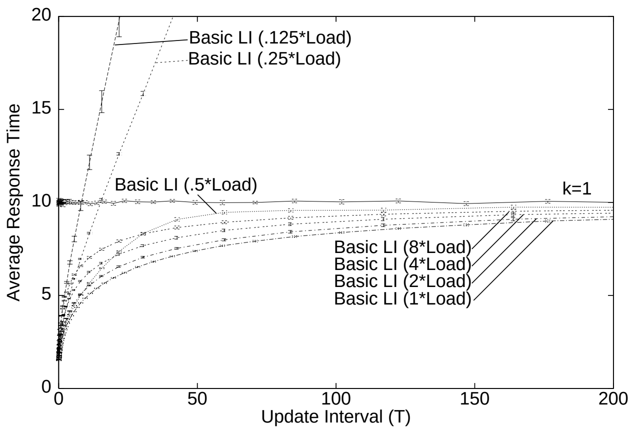


Figure 10. Service time v. update delay for periodic update model when clients mis-estimate the arrival rate.

Figure 10 shows performance under the periodic update model when the LI algorithm uses an incorrect estimate of λ. Each line shows performance when the λ used for calculating P_i is multiplied by an error factor e between $\frac{1}{8}$ and 8. If we overestimate the load, the algorithm is more conservative than it should be and performance suffers a bit. If we underestimate the load, the algorithm sends too many requests to the apparently-least-loaded servers, and performance is very poor.

From this, we conclude that systems should err on the side of caution when estimating λ. From Figure 10 note that if $\lambda_{estimated} = 2\lambda_{actual}$, performance is only marginally worse than if $\lambda_{estimated} = \lambda_{actual}$. Also note that for these experiments, $\lambda_{actual} = 0.9$ and the the system would be unstable if $\lambda_{actual} \geq 1.0$. In other words, to overestimate λ by a factor of two, one would have to predict a service rate 1.8 times larger than could ever be sustained by the system.

We suggest the following strategy for estimating λ: if the system's maximum achievable throughput is known, use that throughput as an estimate of λ for purposes of the LI algorithms. When the system is heavily loaded, that estimate will be only a little bit higher than the actual arrival rate; when it is lightly loaded, the estimate will be far too high. But, as we have seen, the algorithm is relatively insensitive to overestimates of arrival rate. Furthermore, overestimating the arrival rate does little harm when the system is lightly loaded. In that case, the conservative estimate of λ tends to make the LI algorithm distribute requests uniformly across the servers, which is an acceptable strategy when load is low. Figure 11 illustrates the effect of assuming $\lambda_{estimated} = 1.0$ as we vary λ_{actual} for a system with $T = 20$. The two Basic LI lines—one with exact and the other with conservative estimates of λ—are al-

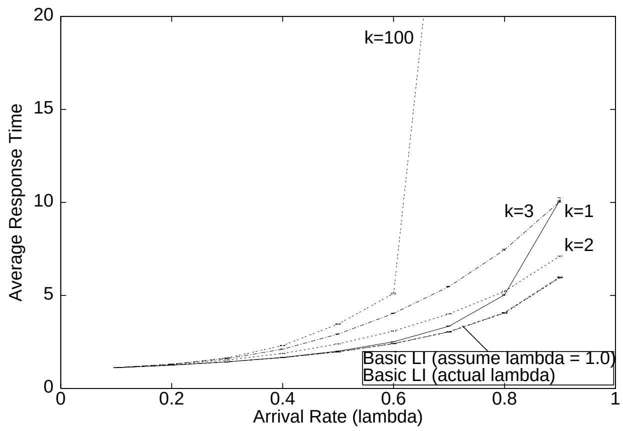


Figure 11. Service time v. arrival rate (λ) for periodic update model with $T = 20$. The graph compares the standard algorithms as well as a variation of the Basic LI algorithm that overestimates λ as the maximum achievable system throughput ($\lambda = 1.0$).

most indistinguishable. For all points, the difference between the two results is less than 1%.

5.6. Impact of reduced information

The k-subset algorithms have an additional purpose beyond attempting to cope with stale load information: by restricting the amount of load information that clients may consider when dispatching jobs, they may reduce the amount of load information sent across the network. A number of theoretical [4, 8, 17, 22, 30] and empirical [12, 20] studies have suggested that load balancing algorithms can often be quite effective even if the amount of information they have is severely restricted.

The Basic LI algorithm can also be adapted to use a subset of server load information rather than requiring a vector of all servers' loads. In the k-subset version of the Basic LI algorithm (*Basic LI-k*), we select a random subset of k servers and use the algorithm to determine how to bias requests among those k nodes. In particular, we modify Equation 2 to use P'_i and L'_i arrays of size k rather than n, to compute L'_{tot} from the smaller L'_i array, to replace n with k, and to calculate $arrive'_T = T * \lambda * k$. Note that, as for the standard k-subset algorithms, we select a different subset for each incoming request.

Figure 12 examines the impact of restricting the information available to the Basic LI algorithm. This experiment suggests that the Basic LI-k algorithm can achieve good performance. Under the update-on-access model, the original k-subset algorithms perform well, and the LI-2 algorithm's performance is similar to that of the standard $k = 2$ and $k = 3$ algorithms. Unlike the

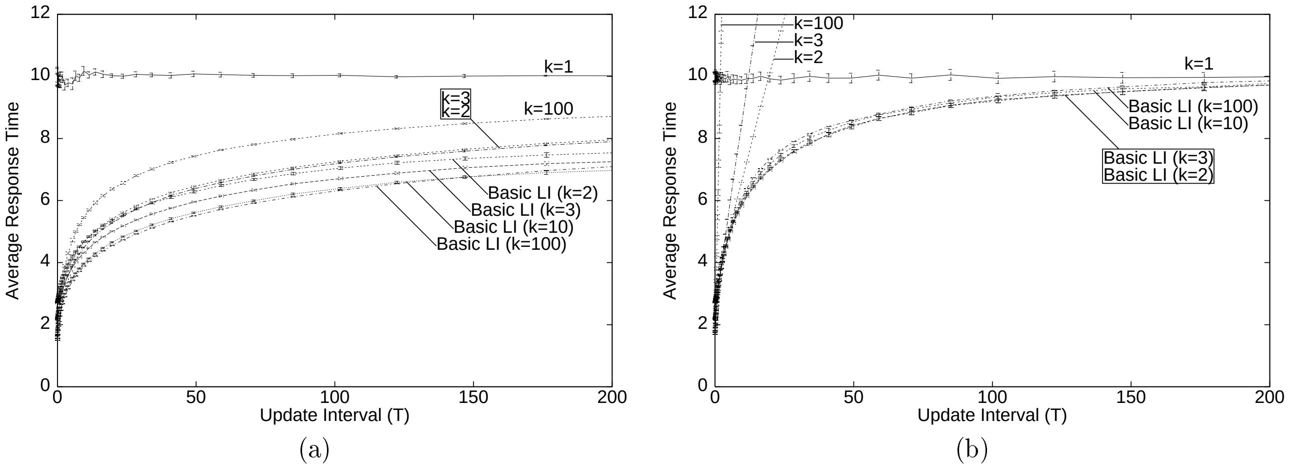


Figure 12. Service time v. update delay for k-subset version of Basic LI algorithm for (a) update-on-access model and (b) continuous update with fixed delay model.

standard k-subset algorithms, as the LI-k algorithm is given more information, its performance becomes better. The LI-3 algorithm outperforms all of the standard k-subset algorithms by a noticeable amount, and the full Basic LI algorithm (LI-100) widens the margin.

Under the continuous update with fixed delay model (Figure 12-b), the performance of the LI-k algorithms is also good. In this case, the original k-subset algorithms behave badly, but the LI-k versions behave nearly identically with the Basic LI system. In this experiment, the $k = 2$, $k = 3$, and $k = 10$ versions of the LI algorithm are slightly better than the $k = 100$ version, with smaller k consistently giving slightly better performance. We do not have an explanation for the improving behavior with reduced information in this experiment.

From these experiments, we conclude that LI can be an effective technique in environments where we wish to restrict how much load information is distributed through the system. Modest amounts of load information allow the LI algorithms to achieve nearly their full performance. Thus, LI decouples the question of how much load information should be used from the question of how to interpret that information.

6. Conclusions and future work

This work introduces the LI framework and provides an evaluation of that approach under simple, synthetically generated workloads. Future work is needed to evaluate and adapt the LI principles to workloads with burstier arrival patterns, workloads with increased job-size variation, and trace or synthetic workloads that model specific applications. Christenson [7], for example, has made preliminary efforts to study LI algorithms under a web-server workload. In addition, in this paper we assume that all servers have equal capacity and leave the more general heterogeneous-server case for future work.

The primary contribution of this paper is to present a simple strategy for interpreting stale load information. This approach resolves the paradox that under some algorithms, using additional information can result in worse performance than using less information or none at all. The Load Interpretation (LI) strategy we propose interprets load information based on its age so that a system is essentially always better off when it has and uses more information. When information is fresh, the algorithm aggressively addresses imbalances; when the information is stale, the algorithm is more conservative.

We believe that this approach may open the door to safely using load information to attempt to outperform random request distribution in environments where it is difficult to maintain fresh information or where the system designer does not know the age of the information *a priori*. Our experiments suggest that by interpreting load information, systems can (1) match the performance of the most aggressive algorithms when load information is fresh, (2) outperform current server-driven load balancing algorithms by as much as 60% when information is moderately old, (3) significantly outperform random load distribution when information is older still, and (4) avoid pathological behavior even when information is extremely old.

Acknowledgements

We thank Lorenzo Alvisi and the anonymous reviewers for their helpful comments on earlier drafts of this

paper.

References

[1] The Next Step in Server Load Balancing. http:// www.alteon.com/ products/ white_papers/ slbwp.html, 1998.

[2] Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, 22(9):47–56, September 1989.

[3] B. Awerbuch, Y. Azar, A. Fiat, and T. Leighton. Making Commitments in the Face of Uncertainty: How to Pick a Winner Almost Every Time. In *Proceedings of the Twenty-eighth ACM Symposium on Theory of Computing*, pages 519–30, 1996.

[4] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced Allocations. In *Proceedings of the Twenty-sixth ACM Symposium on Theory of Computing*, pages 593–602, 1994.

[5] T. Brisco. DNS Support for Load Balancing. Technical Report RFC 1794, Network Working Group, April 1995.

[6] D. Butterfield and G. Popek. Network Tasking in the Locus Distributed Unix System. In *Proceedings of the Summer 1984 USENIX Conference*, pages 62–71, June 1984.

[7] K. Christensen. Dangers of the Herd Effect in Selection Methods for Replicated Web Servers. http://www.csee.usf.edu/ christen/, Feb 1999.

[8] T. Decker, R. Diekmann, R. Lüling, and B. Monien. Towards Developing Universal Dynamic Mapping Algorithms. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 456–459, 1995.

[9] K. Delgadillo. Cisco Distributed Director. whitepaper, Cisco, Inc., 1997.

[10] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(7):757–785, July 1991.

[11] D. Duke, T. Green, and J. Pasko. Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queuing System Version 3.0. http:// www.scri.fsu.edu/ p̃asko/ dqs/ dqs.html, January 1996.

[12] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[13] J. Gehring and A. Reinefeld. MARS – A Framework for Minimizing the Job Execution Time in a Metacomputing Environment. *Future Generation Computer Systems (FGCS)*, 12(1):87–99, 1996. Elsevier Science B.V.

[14] J. Guyton and M. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *Proceedings of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1995.

[15] M. Harchol-Balter, M. Crovella, and C. Murta. Task Assignment in a Distributed Server. In *10th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*. Lecture Notes in Computer Science, No. 1469, September 1998.

[16] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. http:// www.bsp-worldwide.org/standard/standard.htm, May 1997.

[17] R. Karp, M. Luby, and F. auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. In *Proceedings of the Twenty-fourth ACM Symposium on Theory of Computing*, pages 318–326, 1992.

[18] M. Litzkow, M. Livny, and M. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, 1988.

[19] D. Milojicic. *Load distribution: Implementation for the Mach Microkernel*. PhD thesis, University of Kaiserslautern, Kaiserslautern Germany, 1993.

[20] R. Mirchandaney, D. Towsley, and J. Stankovic. Analysis of the Effects of Delays on Load Sharing. *IEEE Transactions on Computers*, 38:1513–1525, 1989.

[21] R. Mirchandaney, D. Towsley, and J. Stankovic. Adaptive Load Sharing in Heterogeneous Distributed Systems. *Journal of Parallel and Distributed Computing*, 9:331–346, 1990.

[22] M. Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California, Berkeley, September 1996.

[23] M. Mitzenmacher. How Useful is Old Information. In *Proceedings of the Sixteenth Symposium on the Principles of Distributed Computing*, 1997.

[24] A. Myers, P. Dinda, and H. Zhang. Performance Characteristics of Mirror Servers on the Internet. Technical Report CMU-CS-98-157, Carnegie Mellon University, July 1998.

[25] D. Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12, October 1987.

[26] M. Powell and B. Miller. Process Migration in DEMOS/MP. *Operating Systems Review*, 17(5):110–119, 1983.

[27] N. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, pages 33–44, December 1992.

[28] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[29] M. M. Theimer, K. A. L., and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[30] N. Vvedenskaya, R. Dobrushin, and F. Karpelevich. Queuing Systems with Selection of the Shortest of Two Queues: an Asymptotic Approach. *Problems of Information Transmission*, 32:15–27, 1996.

[31] D. Wessels. Squid Internet Object Cache. http://squid.nlanr.net/Squid/, August 1998.

[32] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, January 1997.

[33] E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 13–24, October 1987.

[34] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software - Practice and Experience*, 23(12):1305–1336, December 1993.