Distributed Oblivious Load Balancing Using Prioritized Job Replication

Amir Nahir
Department of Computer Science
Technion, Israel Institute of Technology
Haifa 32000, Israel
Email: nahira@cs.technion.ac.il

Ariel Orda
Department of Electrical Engineering
Technion, Israel Institute of Technology
Haifa 32000, Israel
Email: ariel@ee.technion.ac.il

Danny Raz
Department of Computer Science
Technion, Israel Institute of Technology
Haifa 32000, Israel
Email: danny@cs.technion.ac.il

***Abstract*—Load balancing in large distributed server systems is a complex optimization problem of critical importance in cloud systems and data centers. However, any full (i.e., optimal) solution incurs significant, often prohibitive, overhead due to the need to collect state-dependent information.**

We propose a novel scheme that incurs *no communication overhead between the users and the servers upon job arrivals*, thus removing any scheduling overhead from the job execution's critical path. Furthermore, our scheme is *oblivious*, that is, it does not use any state information.

Our approach is based on creating, in addition to the regular job requests that are assigned to randomly chosen servers, also replicas that are sent to different servers; these replicas are served in low priority, such that they do not add any real burden on the servers. Through analysis and simulations we show that the expected system performance improves up to a factor of 2 (even under high load conditions), if job lengths are exponentially distributed, and over a factor of 5, when job lengths adhere to heavy-tailed distributions. We implemented a load balancing system based on our approach and deployed it on the Amazon Elastic Compute Cloud (EC2). Realistic load tests on that system indicate that the actual performance is as predicted.

I. Introduction

Efficient management of large distributed computation systems is a long standing challenge both from the academic and the practical point of view. Recently, due to the growing interest in large data centers, mainly in the context of cloud Computing [13], this problem is receiving renewed attention.

In order to provide a global service and to cope with the high load, many services are provided simultaneity from several data centers placed all around the world. Each such data center may contain ten of thousands of servers [14], many of them executing the same service. This is the case, for example, with the Google search services - at each given point in time, there are several front-end servers and several back-end search engines active in various locations over the globe. An incoming user search-request is directed to one of the front-end servers (using DNS) and then redirected into one of the back-end search engines.

This research was supported in part by the Israel Science Foundation (grant no. 1129/10).

This research was supported in part by the EC 7th Framework SAIL project Scalable and Adaptive Internet Solutions, (FP7-ICT-2009-5-257448).

A specific request for the service is processed at a specific server in one of these data centers. The service has to make an online decision regarding the specific server to process a given request. This decision has a critical impact on the time it actually takes to serve the request (i.e., user experience) and on the overall performance of the service. This assignment problem is a very difficult optimization problem that depends on many parameters such as network latency, server loads, and data access capacity. Moreover, minimum service time is not always the only criterion, and service owners may have other objectives such as balancing the load across all their servers or reducing operational costs. Regardless of the exact optimization goal, any adaptive system that tries to find such an assignment needs to access a significant amount of information from various locations in the network. Moreover, this data must be synchronized and up-to-date in order to avoid the stale data problem [19]. To address this, many common load balancing techniques, such as the "supermarket Model" [18], collect fresh data upon a jobs arrival. This introduces additional delay into the critical path of serving the request, as the data collection process hinders the selection of the executing server and the arrival of the job to that server.

The increasing size of data centers and their distribution across the globe limit the ability to use any centralized load balancing solutions, and thus studying of oblivious[1] distributed load sharing system is even more important.

In this study we introduce a novel approach for addressing oblivious load sharing. This approach is based on employing a two-priority service discipline at the servers, and duplicating each job request into two copies, with different priorities, which are sent randomly to different sites. Now, if the high priority copy arrives at a loaded server, the low priority copy may end up in a lightly loaded server and be processed there, decreasing the average time of the job in the system. Since at each server high priority jobs are always served before any low priority job, the performance of such a system is always at least as good as the basic random assignment technique and, depending on the load, it has the potential of offering considerably improved performance. Informally, one can think

[1] An *oblivious system* (also termed *static system*) is a system that is independent of the current state, and does not use any dynamic input.

978-3-901882-48-7

of the low-priority job scheme as an auxiliary mechanism that uses "leftover" capacity to increase the overall response time of the system.

The two servers assigned with the job coordinate the removal of one of the copies. In case the high priority copy arrives to the head of the queue and begins processing, a signal is sent to trigger the removal of the matching low priority copy. In case the low priority copy completes processing, a similar signal is sent to trigger the removal of the matching high priority copy. Note that since servers preempt low priority jobs upon the arrival of high priority ones, the removal signal is sent only upon *completion* of a low priority job.

Our method takes away signaling and monitoring delay from the execution path. Upon the job's arrival, no data collection is performed, the job is sent to two servers with no decision-making delay. Only after the job begins processing (in the high priority copy case) or completes processing (in the low priority copy case) do we spend (very little) time in executing the required signaling. Overall, this approach generates less overhead than the Supermarket Model [18]. Moreover, the delay induced does not directly affect the request's completion time.

We test our technique under two load profiles. The first is a "traditional" setting in which jobs arrive based on a Poisson process, and job lengths (required processing times) are exponentially distributed. Such load profiles have been widely considered in the literature, due to their amenability to formal analysis as well as often being suitable approximations of reality. In such a setting, it seems reasonable that the scheme will not help much at low load levels, since the high priority copy is most likely to be handled fast; the same could be expected to apply in very high loads, where the auxiliary capacity is very small, as all servers are busy satisfying high priority jobs. However, for intermediate load levels, the probability that a low priority copy will end up in a free server and be executed before the main copy is not negligible, and as a result we may get a considerable improvement in the overall time a job spends in the system. Indeed, we show, by analysis, simulations and real-world implementation[2], that our scheme yields a significant improvement for such loads. Moreover, and quite surprisingly, we get an even better improvement for higher levels of load. This rather counter-intuitive phenomenon is due to the fact that, in high loads, even a very small amount of auxiliary capacity has a significant impact on the service time. Clearly, this is an important virtue of the proposed scheme, as it can rescue the system when it hits the particularly difficult working zone.

It has been indicated that job lengths in computer systems often exhibit a heavy-tailed distribution [7], [12], [16], which the above ("traditional") profile fails to exhibit. Accordingly, we test our scheme also under a load profile in which jobs arrive based on a Poisson process but their lengths abide to a heavy-tailed distribution, with very high variance. In such a setting, some jobs may be unfortunate to be scheduled for execution on a server behind a very long job. It turns out that the benefit of our proposed scheme is much higher under this setting. Specifically, we show, through simulations, that our scheme yields a dramatic improvement in low loads. In higher loads, low priority replicas end up getting stuck in low priority queues behind long jobs. Once we tune our scheme to drop such jobs, we show that the performance gain for high loads remains significant. These findings are corroborated by our real-world implementation.

Furthermore, we indicate that our two-priority scheme can boost the performance of other load balancing schemes. Specifically, we demonstrate it on the following two instances. First, we augment the Supermarket Model with a low priority copy. In the basic Supermarket scheme, d servers are sampled once a new job arrives. The new job is sent to the least loaded server. When augmenting this method, the high priority copy is sent to the least loaded server and the low priority copy is sent to a randomly chosen server. In [20], we applied our priority-based job duplication scheme also to the well-known (centralized) Round-Robin scheme. Here, the high priority jobs are scheduled in a Round-Robin fashion, and the low priority copy is sent to a randomly chosen server.

To summarize, the main contributions of this paper are as follows.

1) Introduction and analysis of a priority-based job duplication scheme for distributed oblivious load sharing in networked server systems such as the Cloud.
2) Detailed performance study of the proposed scheme, under two representative load profiles, with the following findings:
   - significant improvement for job lengths that are exponentially distributed;
   - dramatic improvement for job lengths that follow a (typical) heavy-tail distribution.
3) An implementation of the scheme within Amazon's EC2 framework, along with testing under simple workloads in conditions equivalent to those of a real-life commercial environment.

The paper is organized as follows. After discussing related work in the next section, in Section III we formalize the model. In Section IV we analyze the proposed solution for exponentially distributed job lengths and validate the results also through simulations. In Section V we show the improvement gained when applying our technique to systems where job lengths adhere to a heavy-tailed distribution. In Section VI we describe our real-world implementation. In Section VII we demonstrate how usage of low priority jobs can improve the Supermarket Model. Finally, conclusions appear in Section VIII.

II. Related Work

The problem of load balancing has been studied extensively in a variety of contexts over the last two decades. Several variations of the problem exist, sometimes under different names (e.g., *load sharing*, *scheduling* and *task assignment*).

[2]We implemented our scheme on the Amazon EC2 services system, as described in Section VI.

Solutions to the problem, often termed *scheduling algorithms* or *assignment policies*, can be classified according to several fundamental properties.

Some solutions assume a centralized system [2], [18], i.e., all jobs arrive at a single entity (called *scheduler* or *dispatcher*), which is responsible for determining which server will process the job. Other solutions assume a distributed system [5], [9], i.e., jobs may arrive to any server in the system, which needs to carry out the scheduling policy and determine which server will process the job. In the sequel, the term *scheduler* refers to the entity administrating the policy, whether it is a central one or not.

Some solutions require *a priori* (i.e., already when the job arrives) knowledge as to the time required to process the job [11]. Other solutions attempt to guess this information as the job is being processed [2]. Yet most solutions do not assume any such knowledge, but rely on either other parts of the system's state, or simply on the number of pending jobs in the different servers.

In certain systems, jobs may be preempted, that is, halted while being processed and be served again later on, either at the same or at a different server [4], [8]. When preemption is allowed, we distinguish between two different behaviors once the preempted job resumes its service, namely *preempt-resume* and *preempt-restart*. In the former, the state of the job is saved upon preemption and processing continues from the point it was stopped. In the latter, the state of the job cannot be saved, therefore, processing starts "from scratch" (and the processing effort invested prior to preemption is lost).

As explained in the introduction, *dynamic* (or *adaptive*) policies require a mechanism to collect the state information from the distributed system, and thus are associated with significant overhead and possible delays. To reduce the amount of state information, a very interesting approach, called the Supermarket Model, was proposed in [18]. The main idea is to consider only d random servers (where d is a small constant) out of the available N, and to assign the job to the one with the shortest queue. It was shown in [18] that even for $d = 2$ this scheme achieves a much better delay than random assignment, and this improvement gets better as we increase d. The scheme described in [18] was centralized and did not consider the time required to collect the data as part of the delay each job incurs. Furthermore, its evaluation did not consider the direct overhead associated with each load query (on the executing servers). A recent study [5] proposed a fully distributed implementation of the Supermarket Model and showed that, under appropriate tuning, this version performs very well even when the overhead is considered. However, the scheme in [5] is still dynamic, does not account for the data collection delay, and uses partial knowledge regarding the current system state. As such, it either incurs delays (if we acquire information on demand), or else the use of stale state information. As mentioned, oblivious schemes, such as those considered in this study, are exempted from these shortcomings. We note that our priority-based job duplication technique can be applied also to the Supermarket Model as described in Section VII.

The idea of leveraging idle cycles on one server to offload work from another server has also been explored [21] (and many references therein). However, the problem defined there focuses on two servers in an asymmetric system, i.e., one server takes the role of the "donor", which can assist another server (the "beneficiary"), but not the other way around; moreover, extensive state-tracking is required, since the donor needs to have access to the current state of the beneficiary.

Recently, in [17], the authors address a very similar problem, that is, how to efficiently load balance a large set of servers while minimizing the average time in the system. However, their approach requires from the back-end servers to be familiar with the scheduling component (front-end servers) and is therefore not applicable to architectures where end-users directly access servers (such as in DNS-based scheduling [6]).

III. Model

We consider a system composed of N identical servers, $1, 2, \ldots N$. Each server has two infinite queues, one for *high priority jobs*, and the other for *low priority jobs*. Jobs arrive at the system according to a Poisson process at an aggregate rate of λN. The length of each job shall be characterized, through two possible distributions, in later sections. We consider batch jobs, so by a "job" we mean a "job request", and the job is performed at any of the servers in the system. We assume a distributed system, therefore there is no single entity that decides on the exact assignment of jobs to servers. Instead, for each job, two servers are chosen uniformly at random out of the N servers, one is assigned with the *high priority* copy of the job, and the other is assigned with the *low priority* copy. The two copies are placed, immediately, at the end of the two respective queues in the chosen servers. The servers process jobs according to the following preemptive priority discipline. As long as there are jobs in the high priority queue, the server serves them on a first-come-first-served (FCFS) basis. Once the high priority queue becomes empty, the server turns to process jobs from the low priority queue (again, on an FCFS basis). In case a high priority job arrives at the server while it is processing a low priority job, the processing is preempted, the low priority job is returned to the head of the low priority queue, and the server turns to process the high priority job. Once the server's high priority queue becomes empty again, the processing of the low priority job starts again, "from scratch" (*preempt-restart*).

When a server begins the processing of a high priority job, or completes the processing of a low priority job, an immediate notification (signal) is provided to the server holding the job's replica, and that copy is removed from the queue.

While the notifications mentioned above introduce some form of coordination between the servers, we note that this coordination is carried out *after* job assignment has been determined, and in a way that does not delay job execution; thus, the load balancing process is indeed state-free, i.e., oblivious.

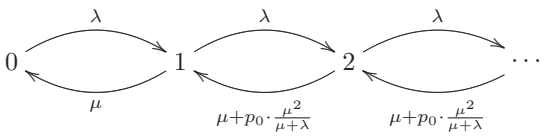

Fig. 1: State-transition-rate diagram

IV. Exponential Job Lengths

In this section we assume that job lengths are exponentially distributed and, accordingly, we analyze the performance improvement gained by the addition of the low priority jobs, as described in the previous section, to a "classical" system of M/M/1 queues, i.e., with Poisson arrivals and exponentially distributed service times (with rate μ).

We employ a state-transition-rate diagram [15] to analyze the problem from the perspective of a single server. We denote by p_i the probability that a server has i jobs in its high priority queue.

While the low priority replicas of a given server's high priority jobs may be distributed among various other servers, to simplify the analysis we make an "averaging" assumption that each server is assisted by a single additional server.

We need to quantify the rate at which high priority jobs held by the server under analysis are processed by other servers (as low priority jobs). Low priority jobs are processed at rate μ whenever there are no high priority job in the server's queue. In addition, we need to take into account the fact that low priority jobs may be preempted during execution. This can be quantified as follows: once a server starts processing a low priority job, two exponential processes compete - one for the arrival of a new high priority job (at rate λ, which will cause the low priority job to be preempted and returned to the queue) and the other for the completion of the low priority job (at rate μ). Therefore, the probability that the next event out of these two will be the completion of the low priority job is $\frac{\mu}{\mu+\lambda}$. Therefore, the overall rate of low priority job completion is

$$\mu \cdot \frac{\mu}{\mu+\lambda} \cdot p_0.$$

Figure 1 depicts a snippet of the state-transition-rate diagram that corresponds to the above model of the problem. We note that when there is a single job in the high priority queue, its processing rate is μ, since its low priority replica is removed from the remote server holding it once processing starts, in accordance to the scheme described above.

The arrival rates in the diagram are the same as in a standard M/M/1 system, while the processing rate is the same or higher. Therefore, the system has a steady state *at least* in the same settings as a standard M/M/1 system, i.e., when $\lambda < \mu$. When steady-state conditions are met, we get the following set of equations:

$$p_0 \cdot \lambda = p_1 \cdot \mu,$$

$$\forall i \geq 1, p_i \cdot \lambda = p_{i+1} \cdot \left(\mu + \mu \cdot \frac{\mu}{\mu+\lambda} \cdot p_0\right)$$

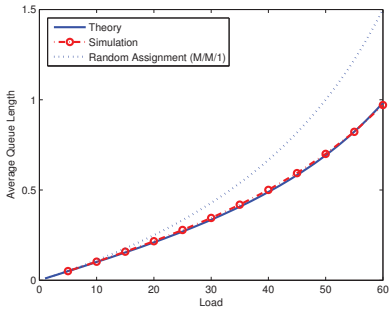

Fig. 2: Results of model and simulation, low loads

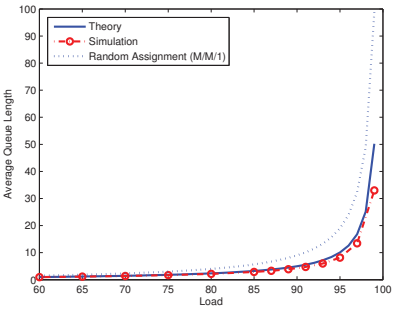

Fig. 3: Results of model and simulation, high loads

In addition, since the p_i's are probabilities, we require that $\sum_{i=0}^{\infty} p_i = 1$.

And thus, the average high priority queue length is:

$$\bar{Q} = \sum_{i=0}^{\infty} i \cdot p_i = p_0 \cdot \frac{\lambda}{\mu} \cdot \left(\frac{\mu+\tau}{\mu+\tau-\lambda}\right)^2$$

where $\tau = p_0 \cdot \frac{\mu^2}{\mu+\lambda}$ and

$$p_0 = \frac{\sqrt{\lambda^2\mu + 4(\mu+\lambda)(\mu^2-\lambda^2)} - \lambda}{2(\mu+\lambda)}.$$

Figures 2 and 3 depict the expected queue length as predicted by our model as well as by data gathered in simulation, when compared to the simple random assignment scheduling policy (i.e., N servers each being an M/M/1 system). We used an in-house event-driven simulator to simulate the behavior of the system. Simulations were carried for various loads in a system of 20 servers, with $\mu = 1$. Each simulation sequence contained 200,000 jobs, and each value was computed as the average of 100 such runs.

The figures show that simulation results match our theoretical model to a high level of accuracy in loads under 90%. Above these loads, standard deviation is high.

As expected, our results indicate that, in very low loads, the scheme of employing duplicate low priority jobs does not improve system performance by much. Indeed, at low loads, system performance is very good as it is, and an arriving job has a high probability of reaching an idle server. It is with higher loads where our scheme should come to the rescue. Yet, at the high-load end, we might not hope for much, as low priority jobs apparently have little chance of getting through.

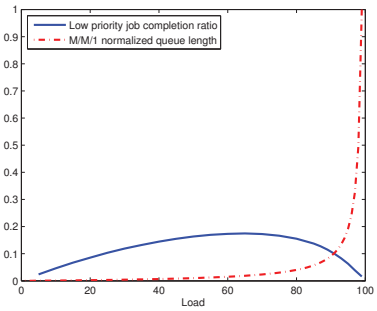

Fig. 4: Average M/M/1 queue length (normalized) and ratio of low priority job completions

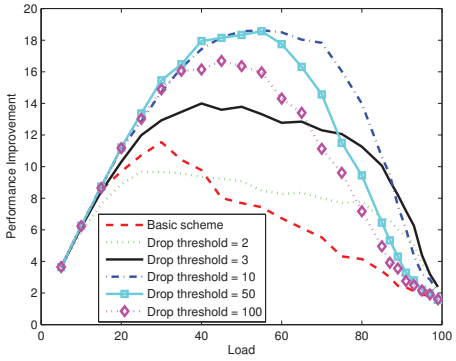

Fig. 5: Improvement gained for various drop thresholds, $\alpha = 1.4$, maximal job length $= 25000$

However, quite counter-intuitively, we observe that, with very high loads, our proposed scheme exhibits an improvement of up to a factor of 1.97 (according to the more conservative results of the theoretical model).

These good news can be explained as follows. Consider Figure 4, where the dotted line shows the expected queue length of an M/M/1 server (namely, $\frac{\lambda}{\mu-\lambda}$), with $\mu = 1$, normalized to 1, and the solid line shows the ratio of jobs (of all user requests) that have been completed as low priority jobs, as obtained through simulations. One can see that, when the load rises over 60%, the expected queue length of the M/M/1 server rises reciprocally, while the low priority job completion ratio decreases almost linearly. Thus, while the contribution of the low priority jobs decreases, it is still sufficient to make a significant impact on the system's performance. Consider, for example, a case when the system operates at 80% load. The expected queue length of an M/M/1 system under such load is 4. Our simulation results indicate that just 15.54% of the jobs first complete as low priority jobs; however, this is enough to bring the system to operate as if the effective arrival rate is 64.46%, where the expected queue length is just 1.81.

V. Power-Law Job Lengths

Several studies have indicated that job lengths in computer systems often exhibit a heavy-tailed distribution. In such systems, the vast majority of jobs are very small, while a few jobs are extremely long. Examples of such systems include the sizes of files transferred through the Internet [7] and the duration of process lifetimes in UNIX systems [12], [16].

A process that adheres to a heavy-tailed distribution can be described based on the following probability cumulative distribution function:

$$P[X > x] \sim x^{-\alpha}$$

where $0 < \alpha < 2$. That is, the number of jobs that are longer than x is an inverse power of x. In our work, we study the system behavior for a variety of α values in this range.

Following other studies in this domain [2], [10], and due to practical reasons (namely, in practice, a job never requires infinite processing time), we opt to model job lengths using the *Bounded-Pareto distribution*. This distribution is defined by three parameters: L - the minimal job length; H - the maximal job length; and α - the exponent of the power-law. Given these, the probability density function of a Bounded-Pareto process is:

$$\frac{\alpha \cdot L^{\alpha} \cdot x^{-\alpha-1}}{1 - \left(\frac{L}{H}\right)^{\alpha}}.$$

We base our findings on simulations. In all settings, for all α values, we keep the maximal possible value fixed and the mean value at 1, adjusting the minimum as needed. We present results for the following maximal job length values: 1000, 10000, and 25000.

When applying our scheme to such arrival patterns, the following phenomenon has been encountered: long jobs reach the head of the low priority queue, get preempted over and over again, thus blocking other low priority jobs from completing. We address this by adding a *drop threshold* to the low priority queue, as follows. We maintain a counter for the job at the head of the low priority queue, keeping track of the number of times it has been preempted; once the preemption counter reaches the designated threshold, the job is dropped from the queue and the counter is set back to 0 (the high priority copy is untouched, thus guaranteed to complete). [3]

Figure 5 depicts one example of the impact of the drop threshold value.[4] Specifically, it shows that, when the drop threshold is set to a low value, performance improvement may be fairly low, and, in some cases (namely, when the load is between 5% and 40%), it may actually be lower than that gained by the basic scheme. In case the drop threshold is set to a too high value, long jobs are kept at the low priority queue for a long time, delaying short jobs from completing. The remainder of the results described in the section have been gathered when the drop threshold was set to 10.

Figures 6, 7 and 8 show the performance improvement gained by applying our scheme to systems where inputs vary according to a Bounded-Pareto distribution. Each figure shows the results for six different α values. The figures differ in the maximal job lengths. Performance gain is computed by dividing the original average time in the system (when jobs

[3]When job lengths are exponentially distributed, this phenomenon is not encountered since variance in job lengths is small, therefore, there is no need to add the drop threshold counter.

[4]The complete results, covering a variety of α values and maximal job lengths, are available online at [20].

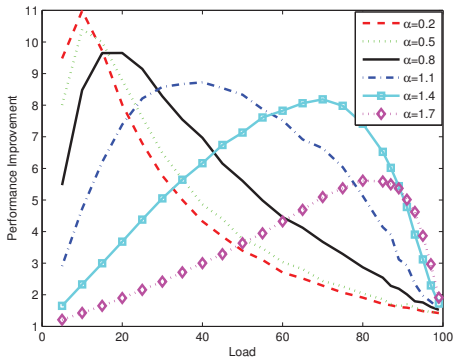

Fig. 6: Improvement gained for various α values, maximal job length $= 1000$, drop threshold $= 10$

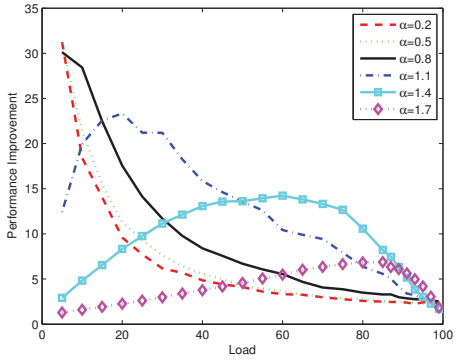

Fig. 7: Improvement gained for various α values, maximal job length $=$ 10000, drop threshold $= 10$

are scheduled according to a random assignment policy) by the average time in the system when our scheme is applied.

As can be seen, our method dramatically improves the system's performance. For typical loads, performance improvement is over a factor of 5; even in the worst cases, improvement remains considerable - ranging from a minimum of 1.4 to 2 for different values of maximal job length.

VI. Real-World Implementation and Evaluation

To demonstrate the effectiveness of our technique, we implemented such a server-based system and tested its performance on the Amazon EC2 system [1].

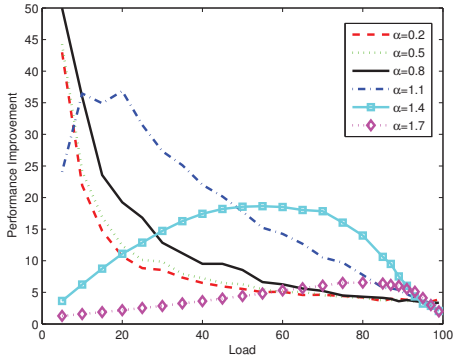

Fig. 8: Improvement gained for various α values, maximal job length $=$ 25000, drop threshold $= 10$

Each server comprises of the following three main components.

(1) **Execution Component:** this is the component that performs user requests. When no job requests exist, this component is idle. When a new job arrives, this component will dequeue it and process it. In case of a high priority job, a removal signal is first sent to the server holding the low priority replica of that job (this data is encoded within the job request), followed by the actual processing of the job request. Note that the removal signal is sent in a non-blocking manner, preventing any delay to the execution of the job. Finally, a response message is sent to the client indicating the completion of the job. The processing of a job is implemented in one out of two ways described later in this section. In case the dequeued job is a low priority one, the Execution Component starts by processing the job, and only after its completion it sends the removal signal to the server holding the high priority replica. As in the case of high priority jobs, after the completion of a low priority job, a response message is sent to the client. The Execution Component may be interrupted while executing a job. Interrupts may arrive from the Job-Removal Component (described below) when a signal has been received reporting that the current job has already been completed by the server that received its replica (note that this can occur both when processing a low priority job as well as when processing a high priority one). In addition, an interrupt may be received from the queuing component (described below) in case a high priority job arrives when a low priority job is being processed. Upon an interrupt, the Execution Component stops the processing of the current job (either dropping it, in case of a removal signal, or returning it to the head of the low priority queue, in case of a priority preemption), and proceeds with the processing of the next job request. Whenever there are multiple pending jobs, the Execution Component will first process high priority jobs (by order of arrival), and only when there are no pending high priority jobs, it will process low priority jobs (by order of arrival).

(2) **Queuing Component:** this component waits for job requests. Upon such a request, this component places it in the appropriate queue (either the high priority job queue, or the low priority job queue, according to the type of the job). In case the Execution Component is processing a low priority job, and the arriving job is a high priority one, the Queuing Component will interrupt the Execution Component in order to activate the preemption mechanism.

(3) **Job-Removal Component:** this component waits for signals arriving from other servers in the system. Upon the reception of such a signal, it indicates the completion of a job request by a remote server. In case the indicated job is currently being processed by the Execution Component, the Job-Removal Component will interrupt the Execution Component, so that the latter can discard this job; otherwise, the Job-Removal Component will search the queues for this job and remove it.

The entire server is implemented in Java, where each component is implemented as a Java thread. In addition, we also

implemented a simple client that generates job requests and sends them to the servers, gathers responses, and produces logs that were processed to create the system statistics. To simplify the implementation, all interactions between the client and the servers, as well as among the servers, were implemented over UDP.

In order to test the performance of our proposed scheme, we launched 10 copies of the above described server on different Amazon EC2 instances of type 'Small'. An Amazon AWS 'Small' instance includes a single computation unit (virtual core) with 1.7GB of RAM on a 32-bit platform. We chose to use instances running Linux. All instances were launched within the same geographical region. Every run described in the results below included 5000 job requests.

We ran our tests in two different setups. In the first setup, servers execute an application that mimics an online book store. In this application, each book entry includes the book's name (a unique random string), the book's cost (an integer between 10 and 200), its publication year (an integer between 1900 and 2010) and the name of its publisher (one out of five given strings). Our book store database holds 3 million book entries. The client acts as users sending queries to the book store. There are two general types of queries: a simple lookup of a book given its name (e.g., *what is the cost of the book called "Moby-Dick"?*, etc.), and queries that collect statistics over the entire database (e.g., *how many books were published since the year 1984?*, or *what is the average cost of a book published by Penguin Group?*, etc.). Our client application generates queries such that 90% of all queries are of the first type. Such queries are simple to process and require, on average, 100ms to execute. The remainder 10% of the queries are of the second type. These queries are more complex to process, since the entire database must be traversed, and require, on average, 8800ms to execute. The client can randomize many of the queries' parameters, such as the publication year to search for, or the type of match (e.g., *before* 1984, *after* 1984, *during* 1984), so overall, there are tens of thousands of possible different queries. When measured in a stand-alone test, the execution time of queries admits a power-law like behavior. While the average query execution time is $975ms$, over 90% of the queries are processed in under $250ms$, while the remainder 10% require 8600 to 9200 milliseconds to process.

Figure 9 depicts the results obtained when the servers run our book store application. Performance gain is computed by dividing the original average time in the system (when jobs are scheduled according to a random assignment policy) by the average time in the system when low priority jobs are supported. Every data point in the graph represents the average of 10 different runs. As can be seen, our method improves system performance by a factor of at least 1.4 for the majority of loads. While these results follow the same trend shown for the simulated Power Law distributions, they fail to reach the same level of improvement. This is because the query execution distribution has a fairly high number of long queries (10%) and the difference between short and long queries is

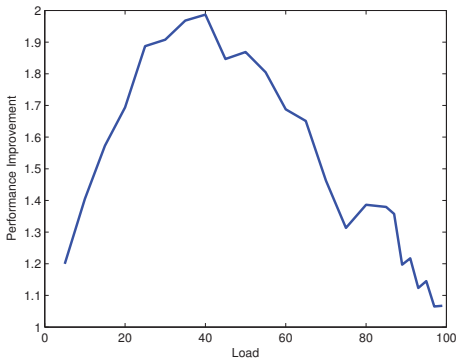

Fig. 9: Improvement Gained with the Book Store Application, Drop Threshold = 10

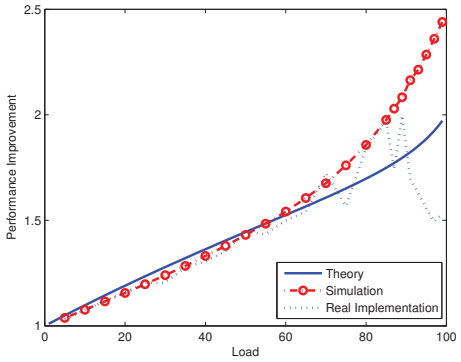

Fig. 10: Improvement gained with Exponential Service, compared to Simulation and Model Results

"only" a factor of 88 (rather than 2400 to 10^{18}, as for the power-law models employed in the simulations).

The second setup behaves as an exponential service. In this case, servers simply perform a busy-wait loop. The client determines the job's expected processing time (so that both copies require the same time to execute). We generated the jobs with an average of 1000ms execution time. Figure 10 depicts the improvement gain as measured in our real implementation, together with the results from our analytical model and simulation [5]. Each data point in the real implementation graph represents the average of 10 different runs. Note that the real-world implementation results are somewhat different than the theoretical and simulation results as they depict improvement in average time in the system (i.e., they present the ratio between the average time in the system in a matching M/M/1 N servers system and the average time in the system of the system that includes low priority jobs), while the theoretical and simulation results address queue length. Results show a good match between our theoretical model, simulation results, and real-world implementation results. As can be expected, the gain in the real-world implementation is somewhat lower than predicted by theory, due to real-world effects, such as signal propagation delay. In addition, the real-world results are much

[5]The data used to generate this graph is the same as that used to generate Figures 2 and 3, with the addition of the data on the real-world setup. The differences in form originate for a different choice of presentation method.

more sensitive to the large deviation in high loads, which can explain the differences in that region. Over 6000 server hours were required to gather the results described in this section.

As part of the testing of our real-world setup, several phenomena, characteristic of real-world systems, were encountered, such as UDP packet loss (which occurred for less than 0.002% of all packets), and effects of signal propogation time (the average measured propagation time between two AWS instances in the same geographical region was 12ms). One particular effect of signal propagation time, is that, on very rare occasions, the original message from the client arrives at the high priority server, starts processing there, and the signal that is supposed to trigger the removal of the low priority job arrives at the low priority server before the low priority job is queued there. To address this case, we added a shared object between the Job-Removal Component and the Queuing Component. This shared object is used to track pending job-removal messages, so that if this particular event occurs, the Queuing Component can determine, upon the job's arrival, that there is no need to queue it and it can be dropped.

VII. Augmenting the Supermarket Model

As explained in the introduction, the large amount of distributed data in modern datacenter settings makes any non-oblivious system complex and very hard to implement. The Supermarket Model, studied in [18], is a highly interesting approach that addresses this problem. The main idea is to consider only d random servers (where d is a small constant) out of the available N, and to assign the job to the one among them with the shortest queue. It is shown in [18] that, even for $d = 2$, this scheme achieves a much better waiting time than random assignment, and this improvement increases with d (the number of considered servers).

While the approach in [18] is centralized and does not consider the direct overhead associated with each load query, it was recently shown in [5] that the Supermarket Model admits a fully distributed implementation which, under appropriate tuning, works very well even when the overhead is considered. However, neither of these works address the problem related to the delay incurred due to the time required to collect the data needed to make the scheduling decision upon a job's arrival.

In order to test our general prioritized job replication paradigm also in non-oblivious scenarios, we applied it to the distributed Supermarket Model in the following way: when a job arrives at the dispatcher, it randomly selects d servers and queries them for the queue lengths (in this technique, queue lengths are used to evaluate the load at the server). In a fully distributed system, the dispatcher may be one of the servers (in which case it will only query $d-1$ servers, in addition to itself). The dispatcher then assigns a high priority copy of the job to the server with the shortest queue and a low priority copy to a random server. The servers work according to the basic priority scheme described in Section III.

Figure 11 depicts the results of the prioritized job replication paradigm compared to the "vanilla" Supermarket Model for exponentially distributed job lengths. As one can see, for a small d (say, 2), the improvement can be as high as a factor of 1.05; this is due to the fact that, in some cases, all of the d servers sampled are busy, and the random selection for the low priority job finds a less occupied one.

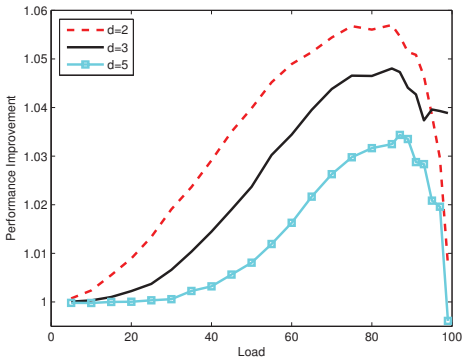

Fig. 11: Improvement gained by using the supermarket model with low priority jobs when compared to the supermarket model without low priority job

VIII. Conclusions

We proposed a scheme for oblivious distributed load sharing in the context of large data centers. Specifically, we showed that when job lengths are exponentially distributed, adding low-priority job replicas can significantly reduce the system service time, even when the overall load is high. Moreover, we showed that the improvement in system performance offered by our scheme is particularly significant when job lengths exhibit a heavy-tailed distribution.

Our findings have been obtained through analysis, simulation and a real-world implementation running in a commercial cloud. The results gathered in the AWS setup indicate the proposed scheme can bring high value in real-world setups, where characteristics of real-world systems, such as signal propagation time and packet loss, are present.

Several extensions to the scheme presented in the paper merit future research. Specifically, exploring the applicability of this concept for other server processing policies, and evaluating the value incorporating low priority copies in advanced load balancing techniques such as Join-Idle-Queue [17].

The idea of using low priority copies is not limited to the data centers or cloud computing settings; a similar idea can be used in many networking scenarios in order to improve response time. Consider for example multimedia packets being transmitted in a cellular or WiFi/WiMax setting, where the recipient may get packets from several base stations [3]. In such a case, one might send more than one copy of a packet to more than one base station, using priorities, and increase the probability that the packet will be received before its timeout.

In some systems, a simpler solution may be called for. In [20] we show how the replication concept, at its utmost simplicity – without signaling between the servers and with no buffering of low priority jobs – can still provide value.

Acknowledgment

The use of the Amazon infrastructure was generously supported by the AWS in Education Research Grant.

REFERENCES

[1] Amazon web services. [Online]. Available: http://aws.amazon.com/

[2] "Task assignment with unknown duration," *J. ACM*, vol. 49, no. 2, pp. 260–288, 2002.

[3] D. Amzallag, R. Bar-Yehuda, D. Raz, and G. Scalosub, "Cell selection in 4g cellular networks," in *INFOCOM*, 2008, pp. 700–708.

[4] A. Barak, S. Guday, and R. G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1993.

[5] D. Breitgand, R. Cohen, A. Nahir, and D. Raz, "On Cost-Aware Monitoring for Self-Adaptive Load-Sharing," *IEEE Journal on Selected Areas in Communication*, vol. 28, no. 1, pp. 70–83, 2010.

[6] V. Cardellini, M. Colajanni, and P. Yu, "Dns dispatching algorithms with state estimators for scalable web?server clusters," *World Wide Web*, vol. 2, pp. 101–113, 1999.

[7] M. E. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: evidence and possible causes," *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 835–846, 1997.

[8] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing," *SIGMETRICS Perform. Eval. Rev.*, vol. 16, no. 1, pp. 63–72, 1988.

[9] S. Fischer, "Distributed load balancing algorithm for adaptive channel allocation for cognitive radios," in *In Proc. of the 2nd Conf. on Cognitive Radio Oriented Wireless Networks and Communications (CrownCom*, 2007.

[10] B. Fu and Z. Tari, "A dynamic load distribution strategy for systems under high task variation and heavy traffic," in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2003, pp. 1031–1037.

[11] M. Harchol-Balter, M. E. Crovella, and C. Murta, "On choosing a task assignment policy for a distributed server system," Cambridge, MA, USA, Tech. Rep., 1998.

[12] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 253–285, 1997.

[13] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, no. 7, pp. 9–11, 2008.

[14] M. Isard, "Autopilot: automatic data center management," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 60–67, 2007.

[15] L. Kleinrock, *Queueing Systems, Vol. I: Theory*. Wiley Interscience, 1975.

[16] W. Leland and T. J. Ott, "Load-balancing heuristics and process behavior," *SIGMETRICS Perform. Eval. Rev.*, vol. 14, no. 1, pp. 54–69, 1986.

[17] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. G. Greenberg, "Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services," *Perform. Eval.*, vol. 68, no. 11, pp. 1056–1071, 2011.

[18] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094 – 1104, October 2001.

[19] ——, "How useful is old information?" *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 1, pp. 6–20, 2000.

[20] A. Nahir, A. Orda, and D. Raz, "Distributed oblivious load balancing using prioritized job replication," Department of Electrical Engineering, Technion, Haifa, Israel, Tech. Rep., 2012. [Online]. Available: http://www.ee.technion.ac.il/Sites/People/ArielOrda/Info/Other/NOR12.pdf

[21] T. Osogami, M. Harchol-Balter, and A. Scheller-Wolf, "Analysis of cycle stealing with switching cost," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 184–195, 2003.