On the Harmfulness of Redundant Batch Requests

Henri Casanova
Dept. of Information and Computer Sciences
University of Hawai‘i at Manoa
henric@hawaii.edu

Abstract

Most parallel computing resources are controlled by batch schedulers that place requests for computation in a queue until access to compute nodes is granted. Queue waiting times are notoriously hard to predict, making it difficult for users not only to estimate when their applications may start, but also to pick among multiple batch-scheduled resources the one that will produce the shortest turnaround time. As a result, an increasing number of users resort to "redundant requests": several requests are simultaneously submitted to multiple batch schedulers on behalf of a single job; once one of these requests is granted access to compute nodes, the others are canceled. Using simulation as well as experiments with a production batch scheduler we investigate whether redundant requests are harmful in terms of (i) schedule performance and fairness, (ii) system load, and (iii) system predictability. We find that two main issues with redundant requests are load on the middleware and unfairness towards users who do not use redundant requests, which both depend on the number of users who use redundant requests and on the amount of request redundancy these users employ.

1 Introduction

Most parallel computing resources are accessed via batch schedulers [1] to which users send requests specifying how many compute nodes they need for how long. Batch schedulers can be configured in various ways to implement ad-hoc resource management policies and may maintain multiple queues of pending requests. In addition, most batch schedulers use "backfilling", which allows some requests to jump ahead in a queue to reduce queue fragmentation. Backfilling may happen when a request is submitted, canceled, or when a job runs for less time than initially requested (which is common). The above makes queue waiting time difficult to predict. Some batch schedulers can provide an estimate of queue waiting time based on the current state of the queue. Unfortunately, these estimates do not take backfilling into account and are typically pessimistic. In spite of recently developed forecasting methods for estimating lower or upper bounds on queue waiting time with certain levels of confidence [2], most users today have at best a fuzzy notion of what queue waiting times to expect. At the same time these users have access to multiple batch-scheduled resources that can be used for running their applications, possibly at different institutions. As a result, rather than picking one target resource based on a poor estimate of queue waiting time, if any, users can send a request to each resource; when one of these requests is granted access to compute nodes the others are canceled. This can be easily implemented by having the application send a callback to the user (or to the program that submitted the requests) when it starts executing.

The admittedly brute-force strategy described above, which we term "redundant requests", is becoming increasingly popular because it obviates the need for difficult resource selection. However, there is a widespread but not verified notion that if "everybody were to use redundant requests" then "bad things would happen". In this paper we answer the question: are redundant requests harmful? We identify three broad reasons why they may be:

1. **Negative impact on scheduling:** redundant requests may disrupt the resource management policies implemented by batch schedulers and may give some users an unfair advantage.
2. **Negative impact on system load:** redundant requests cause higher load on the batch schedulers, on the network, and on the middleware infrastructure used to access remote resources.

3. **Negative impact on predictability:** submissions and cancellations of redundant requests cause churn in batch queues, which may make them less predictable.

We show, via simulations, real-world experiments, and analysis of results obtained by others, that two problems with redundant requests are: (i) load on the middleware; and (ii) fairness towards users who do not use redundant requests. This paper is organized as follows. Section 2 presents background on redundant requests and discusses related work. Sections 3, 4, and 5 focus on the three negative impacts above. Section 6 concludes the paper with a discussion of our findings and with perspective on future work.

2 Background and Related Work

Redundant requests can be sent to:

(i) individual batch queues on multiple resources;
(ii) multiple batch queues of multiple resources;
(iii) multiple batch queues of a single resource; or
(iv) a single batch queue of a single resource.

In (i), (ii), and (iii), the goal is to avoid selecting a batch queue a priori but instead to use the batch queue on which the shortest queue waiting time is experienced. When using multiple resources, as in (i) and (ii), a difficulty may be the heterogeneity of these resources. The computation times requested by each redundant request could be scaled to reflect resource heterogeneity and different numbers of compute nodes could be requested on different resources. Sophisticated users could thus attempt to tailor their requests to extract the best response times on each candidate resource. (Note that typical users are not sophisticated, request computation times that are gross overestimations of needed computation times [3], and may not even have a good understanding of the scaling properties of their applications). More importantly, the resource with the shortest queue waiting time could also be a resource with slow compute nodes, meaning that the shortest queue waiting time may not lead to the shortest turnaround time (i.e., queue waiting time plus execution time). Users then face a conundrum: should one wait possibly a long time for a faster resource? Another conundrum arises when using (iii) above. Different queues typically correspond to higher service unit costs. The question is then whether one should wait possibly a long time for a cheaper resource allocation. Option (iv) above can be useful for “moldable” jobs that can accommodate various numbers of compute nodes. Moldable jobs are common but requesting the optimal number of nodes is difficult [4]. Typically, a larger number will lead to a longer queue waiting time and to a shorter execution time, while a smaller number will lead to a shorter queue waiting time and to a longer execution time. One approach is then to send redundant requests for different numbers of nodes. With this approach, one is faced again with a conundrum similar to the one for option (ii): should one wait possibly a long time for a larger number of nodes? Note that option (iv) can be combined with the other three.

There is no one-size-fits-all answer to these conundrums as the solution strongly depends both on the expected application execution times and on the system load, forcing users to use heuristics. These heuristics could be ad-hoc, could use queue waiting time statistics and/or forecasting [2], or could use real-time request status information from the batch schedulers that gives a sense of the request’s place in the queue (unfortunately most currently deployed schedulers do not provide such information). Finally, note that the number of redundant requests that can be used for (iii) and (iv), and possibly (ii), can be bounded by the batch scheduler. Indeed, batch schedulers can typically be configured so that a single user can only have a limited number of pending requests in the batch queue(s). In this paper we study option (i), use a simple model for generating redundant requests in a heterogeneous environment, and leave options (ii), (iii), and (iv) for future work. Nevertheless, we believe that our main findings apply to these options as well.

Previous works have explored the use of redundant requests. Most notably, Subramani et al. [5] and Sabin et al. [6] have studied them as a way to perform job scheduling in a grid platform. In their works, the redundant requests are not initiated by the users but by a metascheduler [5, 6, 7, 8, 9, 10, 11] to potentially offload work to remote resources. These works show that using redundant requests can lead to better overall performance, and more so in systems containing clusters with different numbers of nodes. Although related, our work studies redundant requests generated by users without the knowledge of the scheduler(s). This has two important implications. First, a metascheduler can choose remote clusters based on some global knowledge about the system (e.g., queue sizes) in order to let redundant requests ”play nice” with each other. By contrast, we study user-driven redundant requests that may negatively disrupt the schedule at remote clusters. Second, in our case the use of redundant requests may provide users who use them with an unfair advantage, which we study thoroughly. Note that as of today, no widely accepted metascheduler is deployed but many users resort to redundant requests. Also, we study the impact of redundant requests on the load on the batch scheduler, on the load on the middleware, and on the predictability of the system, which is not done in [5]. Another related work in the "placeholder scheduling" technique introduced in [12], which allows for a late binding of the application to the resources allocated by a batch scheduler. It provides a simple way to implement redundant requests since a callback is sent to the user when the application is ready to execute. At that time, the user may cancel redundant requests.

3 Impact on Scheduling

In this section we investigate whether redundant requests negatively impact job scheduling, both in terms of overall system performance and of fairness among users who use them and users who do not.

3.1 Experimental Methodology

3.1.1 Simulation Model

We use simulation because experiments on production systems would be prohibitive both in terms of time and money (i.e., service unit allocations on batch-scheduler resources), because they would be limited to a specific configuration, and because they would hardly be repeatable. We have implemented a simulator using the SIMGRID [13] toolkit as it provides the needed abstractions and realistic models for the simulation of processes interacting over a network. We simulate a platform that consists of a number of *sites*, where each site holds a parallel platform, say, a *cluster*. Each cluster is managed by its *batch scheduler*. We also simulate a *stream of jobs* at each cluster. Each job requires some number of compute nodes for some duration, sends a request to the local cluster, and may send redundant requests to other clusters. We detail below the components of our simulation model.

Clusters and Batch Schedulers – We simulate a set of N clusters, $C_1, \ldots, C_N$. Cluster C_i contains n_i identical compute nodes. Different node speeds could be accounted for by scaling the request compute times and number of nodes (as discussed in Section 2), but this is not straightforward to model. Instead, we limit heterogeneity to the number of nodes and we generate potentially different workloads at identical clusters (i.e., more or less requests per second), as described below. Each cluster is managed by a batch-scheduler, which can use one of three job scheduling algorithms: EASY [14], Conservative Backfilling (CBF) [15], or First Come First Serve (FCFS). The EASY algorithm enables backfilling and is representative of algorithms running in deployed systems today. Although widely studied, the more complex CBF algorithm is, to the best of our knowledge, not implemented in any real-world system. This is also the case for FCFS, but it is a simple algorithm that is commonly used as a base-line comparator. We model each batch scheduler as managing a single queue and we do not model any request priorities.

Workload – Simulating a stream of jobs can be done either by using a workload model or by "replaying" traces collected from the logs of real-world batch schedulers. The results presented in this paper were obtained with the former approach. We use the model by Lublin et al. [16], which is the latest, most comprehensive, and most validated batch workload model in the literature. Accordingly, we model request arrival times using a Gamma distribution (corresponding to the "peak hour" model in [16]). We model the requested number of nodes with a two-stage log-uniform distribution biased towards powers of 2. We model the requested compute times with a hyper-Gamma distribution whose p parameter depends on the requested number of nodes. Unless specified otherwise, we instantiate the parameters of all the distributions using the "model" parameter values derived in [16], to which we refer the reader for all details. We conducted some simulations using real-world traces made available in the Parallel Workloads Archive [17] but, expectedly, did not observe significantly different results. We opted for a workload model rather than traces for the experiments presented in this paper as it is straightforward to modify the model's parameters to study different scenarios.

3.1.2 Assumptions

To isolate the effects of redundant requests on scheduling we do not simulate any network overhead. This includes the overhead for sending a request to a potentially remote cluster, which is arguably small. More importantly, we also ignore the overhead for sending application input data, if any, to a remote cluster. To use a remote cluster, a user must pre-stage input data on that resource (unless the application streams or downloads its input data directly). However, when using redundant requests, users usually do not pre-stage input data to all remote clusters but wait until nodes are allocated on a particular cluster. The typical approach is then to request extra computation time that will be used to upload application input data to the cluster. Therefore, the only direct impact of redundant requests on the specifics of the workload is that requested computation times may be higher than when there are no redundant requests. (Note that the workload model in [16], which we use in this work, was developed based on logs of requests that most likely were not redundant.) However, we performed experiments in which we increased the requested duration of redundant requests by 10% and 50% and interestingly observed no difference in our results. This showed that the results in this paper hold when users request more compute time to allow late binding of application data to remote resources. Finally, it is shown in [5] that the added overhead of using redundant requests when proactively transferring application data to all candidate resources does not impact the effectiveness of using redundant requests. Similarly, we do not simulate the overhead for processing requests. However, in for the experiments in this section it is insignificant when compared to the queue waiting times and compute times. (We study network, middleware, and scheduling overheads in Section 4.)

3.2 Performance Metrics

We use two popular metrics to evaluate (steady-state) schedule quality in this paper. First, we compute the *average stretch* over all jobs in the system, where the *stretch* (also called slowdown) of a job is the job's turnaround time, which is its execution time plus its queue waiting time, divided by the job's execution time. We did not use the average turnaround time as a metric because it can be skewed by long jobs. Furthermore, the stretch makes it possible to easily compare results obtained with different workloads, i.e., different job durations. Our results were essentially unchanged when analyzed with the turnaround time metric. Second, we compute the *coefficient of variation of stretches* (i.e., the standard deviation divided by the average, in percentage, over all jobs in the system), which is a way to evaluate the fairness of a schedule: a lower standard deviation indicates a fairer schedule. Another popular metric that is related to the fairness of a schedule is the maximum stretch, and here again the conclusions from our results were not change when using this metric.

3.3 Simulation Results

We first simulate an environment that consists of N identical clusters, for $N = 2, 3, 4, 5, 10, 20$. Each cluster contains 128 compute nodes and is managed by a

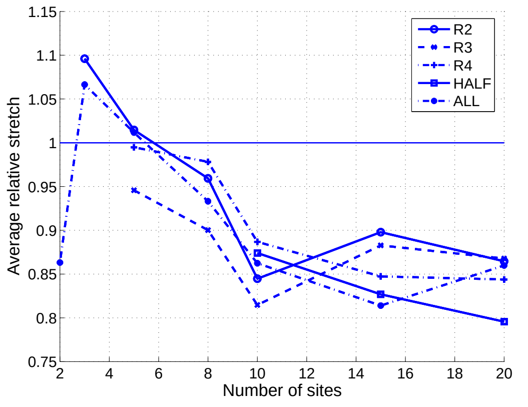

Figure 1. Average stretch for redundant request schemes relative to the scheme using no redundant requests, versus the number of clusters, averaged over 50 experiments.

scheduler that uses the EASY algorithm. Each cluster receives a stream of jobs generated according to the model described in Section 3.1.1. We simulate 6 hours of job submissions (around 4,000 jobs given that the mean job interarrival time for the base model in [16] is roughly 5 seconds). We evaluate five redundant request schemes: $R2$, $R3$, $R4$, $HALF$, ALL, in which a request is sent to 2, 3, 4, half, and all clusters, respectively. One request is always sent to the local cluster, and remote clusters are chosen randomly according to a uniform distribution. Other methods for choosing remote clusters are possible. For instance, inspired by the work in [5], one may select remote clusters based on batch queue lengths. However, it is not clear why users would go through the trouble of picking remote clusters as opposed to just blindly sending requests to all clusters on which they have accounts. Consequently, our method of random selection merely reflects the fact that different users have accounts on different clusters. We also report on results obtained with non-uniform distributions of accounts across clusters.

For now we assume that the same redundant request scheme is used by all jobs. For each experiment we generate N random job streams and simulate the six schemes above for these streams. We compute the average job stretch achieved by each scheme relative to that achieved by the scheme using no redundant requests. Figure 1 shows the relative average stretch, averaged over 50 experiments. Over these 50 samples,

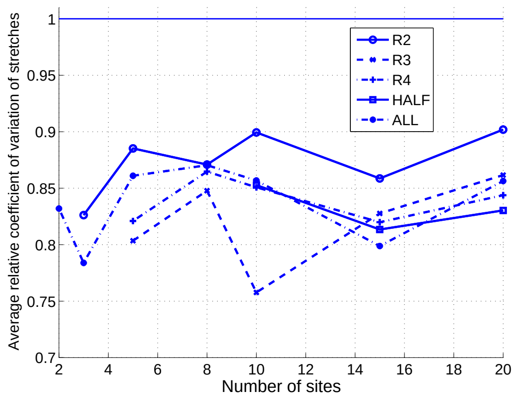

Figure 2. Coefficient of variation of stretches for redundant request schemes relative to the scheme using no redundant requests, versus the number of clusters, averaged over 50 experiments.

we measure coefficients of variation ranging approximately from 50% to 5% when going from $N = 2$ clusters to $N = 20$ clusters. The same holds true for all experiments that follow. (We do not show error bars on the graphs to avoid clutter.) Values below 1 in Figure 1 mean that using redundant requests is beneficial.

One can see that in the worst case using redundant requests leads to an average stretch 10% higher than when not using them, on average. Using redundant requests becomes beneficial regardless of the redundant request scheme for $N > 5$. For $N \leq 5$, it seems that a few short jobs get penalized due to a few lost opportunities for backfilling, thereby increasing their stretch. Accordingly, when using the average turnaround time as a metric, using redundant requests is always beneficial even for $N \leq 5$. This phenomenon disappears, or at least becomes insignificant, as the number of cluster increases. Note that many high-performance computing resource users today have accounts on five different clusters or more, and that this number will increase as the number of deployed clusters continues to grow.

Redundant requests are beneficial because they allow for a better load-balancing of requests across clusters. Using redundant requests leads to better average stretches in more than 95% of the experiments for $N = 20$, more than 90% for $N = 15$, and more than 85% for $N = 10$. When using redundant requests leads to a worse average stretch, the stretch is worse by at most 0.4%, 1.7%, and 2.1% for $N = 20$, $N = 15$, and $N = 10$, respectively. Conversely, when using redundant requests leads to a better average stretch, the stretch is better on average by 15 to 25%. We examined some of the cases in which using no redundant requests is better than using redundant requests. These cases seem idiosyncratic and we could not derive any general reason why redundant requests may not lead to better results beyond particularly "unlucky" sequences of request submissions.

While the average job stretch is related to the performance perceived by users, the coefficient of variation of the stretches is related to fairness. Figure 2 shows the coefficient of variation of job stretches relative to that experienced when no redundant requests are used, averaged over 50 experiments. One can see that in all cases using redundant requests improves the fairness of the schedule by approximately 10 to 25%. The maximum job stretch, which is often used as a measure of fairness, is improved even more by the use of redundant requests, from 10 to 60% on average (not shown on the figure). Here again, the improved fairness can be attributed to the fact that redundant requests lead to better load-balancing.

Other algorithms and runtime estimates – The results above were obtained for batch schedulers using the EASY algorithm and assuming that jobs request precisely the amount of compute time that they need. Table 1 shows results obtained for $N = 10$ clusters and for the $HALF$ redundant request scheme for the EASY, CBF, and FCFS algorithms, and for exact and non-exact estimates. We obtained similar results (i.e., all average relative metrics under 1) for other redundant request schemes and other values of $N > 5$. One can see that the average stretch and coefficient of variance of stretches, relative to when no redundant requests are used, are all below 1 no matter what scheduling algorithm is used (EASY, CBF, or FCFS), and whether jobs request exactly the compute time they need ("Exact Estimates") or more compute time than they actually need as is the case in practice ("Real Estimates"). We model requested compute times using the "ϕ model" proposed in [18], with $\phi = 0.10$, which leads to a uniformly distributed overestimation factor with mean 2.16. A more sophisticated model was recently proposed in [3], and although we may use it to repeat these experiments in future work, we do not expect the results to be significantly different.

Non-uniform redundant request distribution – We repeat our first experiment for $N = 10$ clusters, sim-

Table 1. Results for three different job scheduling algorithms and for real runtime estimates and exact runtime estimates, relative to the scheme using no redundant requests, averaged over 50 experiments.

	Relative Average Stretch		Relative C.V. of Stretches	
	Exact Estimates	Real Estimates	Exact Estimates	Real Estimates
EASY	0.88	0.83	0.83	0.83
CBF	0.90	0.83	0.86	0.83
FCFS	0.93	0.93	0.93	0.93

Table 2. Results for non-uniformly distributed redundant requests, relative to the scheme using no redundant requests, averaged over 50 experiments.

	$R2$	$R3$	$R4$	$HALF$
Relative Average Stretch	0.94	0.95	0.88	0.89
Relative C.V. of Stretches	0.94	0.92	0.88	0.86

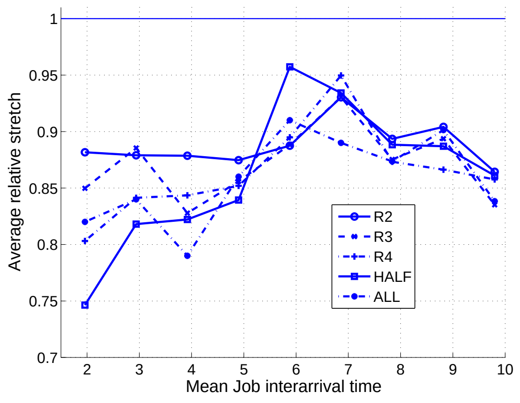

Figure 3. Average stretch for redundant request schemes relative to the scheme using no redundant requests, versus job interarrival times, averaged over 50 experiments, for N=10 clusters.

ulating redundant requests schemes that pick remote clusters at random but biased towards some clusters. We model the probability of picking remote cluster C_1 is twice as high as the probability of picking remote cluster C_2, which is twice as high as the probability of picking remote cluster C_3, and so on. This (arbitrary) distribution is heavily biased (half of the clusters are each picked with only probability 6.25%). Results are shown in Table 2, averaged over 50 experiments. One can see that on average the use of redundant requests is beneficial both for performance and for fairness, even when the targets of the redundant requests are not uniformly distributed. In fact, the results are similar to those obtained with a uniform distribution.

Job interarrival times – One may wonder whether using redundant requests is more or less harmful given different levels of workload. Figure 3 shows relative average stretch for our redundant request schemes in a 10-cluster platform, averaged over 50 experiments, for various job interarrival times (in seconds). The base model proposed in [16] produces mean interarrival times of 5.01 seconds, which is the mean of a Gamma distribution with parameters $\alpha = 10.23$ and $\beta = 0.49$ (the mean is equal to $\alpha \times \beta$). To simulate other workloads we vary the value of α from 4 to 20, leading to interarrival times between approximately 2 and 10 seconds. One can see on the figure that using redundant requests improves average stretch regardless of the job interarrival time. The same is true for the coefficient of variation of stretches (not shown).

Heterogeneity – So far, all our experiments were conducted on a homogeneous system with identical clusters and statistically identical job streams at all clusters. To evaluate whether redundant requests are

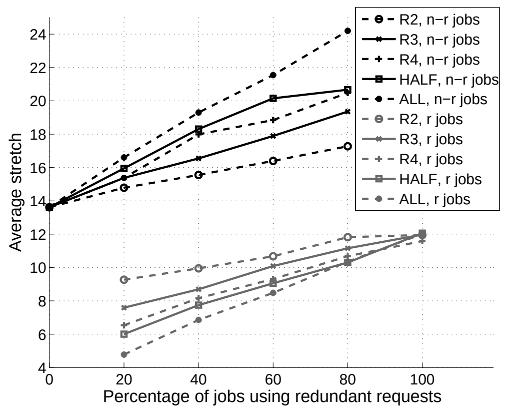

Figure 4. Average stretch for jobs using redundant requests (”r jobs”) and jobs not using redundant requests (”n-r jobs”), for different redundant request schemes, versus the percentage of jobs using redundant requests, averaged over 50 experiments.

harmful in a heterogeneous environment we conduct the following experiment. We simulate $N = 10$ clusters, where each cluster contains a number of nodes picked randomly among 16, 32, 64, 128, and 256. Each cluster receives a job stream with job interarrival times picked randomly between 2s and 20s. Jobs arriving at a cluster do not request more compute nodes than available at that cluster. Therefore, some jobs may only be runnable on a few, or even on one cluster. Table 3 shows results for all redundant request schemes, relative to the scheme using no redundant requests, averaged over 50 experiments. One can see that using more redundant requests is even more beneficial (higher performance and better fairness) than in the homogeneous case across the board. This is because the load balancing due to redundant requests is more effective in a heterogeneous system.

Table 3. Results for heterogeneous platforms, relative to the scheme using no redundant requests, averaged over 50 experiments.

	Relative Average Stretch	Relative C.V. of Stretches
$R2$	0.83	0.90
$R3$	0.74	0.85
$R4$	0.71	0.84
$HALF$	0.63	0.81
ALL	0.67	0.79

Penalty for not using redundant requests – We have seen that using redundant requests is almost always beneficial (when it is harmful it is only when $N = 2$, or by a small margin when $N > 2$). However, in all our experiments we have assumed that all jobs benefit from the (same) opportunity to use redundant requests. This is most likely not the case in the real-world. Some users may have accounts only on one cluster. Some users may have accounts on all clusters. Some applications may need to run on a specific cluster due to the proximity of a large data set. An important question is then: how harmful is it to have only some fraction of the jobs employ redundant requests? To answer this question we conduct experiments assuming that only p percent of the jobs use redundant requests (these jobs all use the same redundant request scheme). Figure 4 plots average job stretch versus p, for jobs using redundant requests (“r jobs”) and jobs not using them (“n-r jobs”), for different redundant request schemes. These experiments are for $N = 10$ clusters, and each data point is averaged over 50 experiment. One can see several interesting features on this graph. The average stretch is better when $p = 100$ than when $p = 0$, which confirms the results of the previous experiments. One can see that when p increases, the average stretch increases, for both types of jobs. This means that the more jobs use redundant requests, the worse off are the ones not using redundant requests. Therefore, jobs using redundant requests negatively impact the performance perceived by jobs not using redundant requests. This impact grows roughly linearly with p. Furthermore, this negative impact is larger for the schemes using more redundant requests per jobs. For instance, if 40% of jobs send requests to all N clusters, their stretches are reduced on average by approximately a factor 2, but the stretches of the other jobs increase by over 40%. Note that jobs using redundant requests all benefit from using more redundant requests.

4 Impact on System Load

To isolate the effects of redundant requests on scheduling, our simulations so far have ignored all overheads involved in job submissions and job cancel-

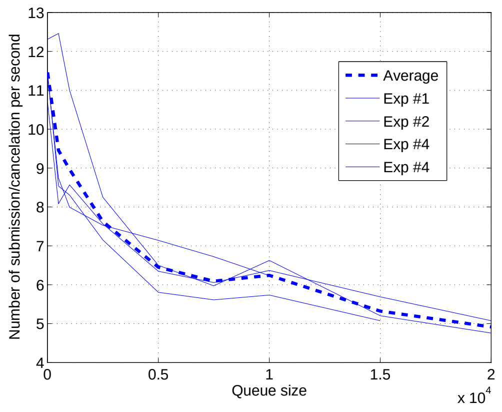

Figure 5. Measured throughput of the OpenPBS/Maui batch scheduler in number of request submissions/cancellations per second versus batch queue size in number of pending requests.

lations. In this section we study the impact of such overheads in terms of the batch scheduler, the network, and the middleware.

4.1 Batch Scheduler

Our experiments were conducted assuming that batch schedulers could make scheduling decisions instantaneously. But in reality redundant requests require that batch schedulers perform more work due to more request submissions and cancellations.

To estimate the job submission/cancellation throughput of a batch scheduler we perform the following experiment. We use an installation of OpenPBS v2.3.16 using the Maui scheduler v3.2.6p.13, running on a 1GHz Pentium III, which is the front-end of a 16-node Linux cluster. The cluster runs a long job that monopolizes all compute nodes for the duration of the experiment, so that pending jobs are never executed. We submit a fixed number of random jobs to generate a given queue size (in number of pending jobs). We then saturate the batch scheduler with job submissions and job cancellations by running multiple processes that continuously submit new jobs using the `qsub` command and delete the job at the head of the queue using the `qdel` command. Deleting the job at the head of the queue causes the maximum amount of churn. Figure 5 shows the average number of job submissions/cancellations per second versus the queue size. Each curve corresponds to a 12-hour experiment, and the thick dashed line shows the average over all experiments. The variations among experiments are in part due to non-deterministic load on the front-end node, which was not dedicated to our experiments but was only mostly quiescent. Note also that some curves do not show values for the higher queue sizes as experiments were interrupted due to the job scheduler process running out of memory, due to memory leaks.

One can see that as the queue size increases the throughput of the job scheduler decreases sharply at first and then slower, in a somewhat exponential manner. When the queue is empty the job scheduler can perform around 11 request submissions and 11 request cancellations per second. This number drops to about 5 when the queue contains 20,000 pending requests. In practice, the number of jobs in the queue of a batch scheduler is on the order of a few thousands. Using the workload model in [16], the queue of a batch scheduler grows by about 700 jobs per hour during so-called "peak" hours (independently of the size of the cluster). Therefore, after 12 hypothetical peak hours, the queue would contain under 10,000 jobs. The use of redundant requests should not increase queue sizes significantly since redundant requests are canceled upon the start of job execution (therefore, *in steady-state*, using redundant requests does not cause significantly more requests to be in the system). This simple observation is confirmed in simulation. For instance, when simulating $N = 10$ clusters for a 24-hour period, we found that the average maximum queue size across all clusters for the ALL redundant request scheme is larger than when no redundant requests are used by less than 2%.

Consider a system with N clusters, with mean job interarrival time of iat seconds at each cluster. If all jobs use r requests, then on average each cluster will receive r/iat requests per second and $(r-1)/iat$ request cancellations per seconds (assuming that the system is in steady state). Conservatively assuming that all queues contain 10,000 requests, the experimental results in Figure 5 indicate that the batch schedulers could support 6 submissions and 6 cancellations per second. Therefore, the batch schedulers operate within their achievable throughput if $r/iat \leq 6$. Assuming job interarrival times of 5 seconds, which corresponds to the peak hour rate of the model in [16], we obtain $r < 30$. This means that in a multi-cluster system, the load on batch schedulers due to redundant requests is tolerable as long as jobs do not use more than 30 redundant requests on average. This is well above the average level of request redundancy that we expect to see in real-world systems today.

The back-of-the-envelop calculation above is optimistic as it assumes that requests are uniformly distributed among clusters. But it is also pessimistic as it assumes that the job interarrival time is always 5 seconds and that all jobs use redundant requests. Furthermore, we measured the throughput of the batch scheduler on a 1GHz Pentium III, and architectures with higher performance would likely improve the throughput further. Our conclusion is that, at least in todays system, the use of redundant requests is unlikely to cause a noticeable load increase on batch schedulers.

4.2 Network and Middleware

Redundant requests also increase the load on the network and the middleware layer used to access remote resources. In the previous section we have determined that, with a queue size of 10,000, each batch scheduler can process on average 12 transactions per second. In this section we assess whether the network or the middleware infrastructure can support this throughput.

As discussed in Section 3.1.2, redundant requests cause no extra network load for transferring application data. Even if the network payload of a job submission or cancellation were on the order of hundreds of KBytes (for instance large SOAP [19] messages), most networks connecting a batch scheduler to the Internet can easily support tens of such interactions per second.

Beyond the network, redundant requests generate more load on the middleware system itself. Using a middleware layer to submit jobs to a Grid service, layered above a batch scheduler, entails many operations (service instantiations, marshalling and unmarshalling of SOAP transactions, disk writes, etc.). Some of these operations are known to lead to little overhead. For instance, in [20], authors measured the performance of the gSOAP [21] implementation of the SOAP protocol when marshalling and unmarshalling arrays of 30,000 data structures that each consist of two integer and one double precision numbers, for a total of more than 450KB, i.e., many more bytes than needed for a batch request submission. Results on a dual-processor Pentium 4 Xeon with 1GB of RAM showed that such transactions can be performed at a rate significantly higher than 12 per second, meaning that the batch scheduler itself would still be the bottleneck (according to the results in Section 4.1). These encouraging results must be put in perspective with the job submission throughput achieved by currently deployment and full-fledge middleware implementations such as the Globus® WS-GRAM [22]. Results obtained with GRAM running on a 2.16GHz AMD K7 processor are presented in [23] (Table 4). For the latest grid service version of GRAM (i.e., GT4 WS-GRAM), a throughput of slightly under 60 transactions per minutes can be sustained, or under one transaction per second. If a job cancellation causes roughly the same overhead as a job submission as far as GRAM is concerned, then .5 job submissions and .5 job cancellations can be processed per second. Let's follow the same reasoning as in Section 4.1: assuming that the job interarrival time is $iat = 5$ seconds and that all jobs use r requests, then $r/iat \leq 0.5$ leading to $r < 3$. This indicates that the current version of GRAM would be the bottleneck for a system in which all jobs use 3 or more redundant requests. (Recall that we found this number to be equal to 30 before the batch scheduler becomes the bottleneck.)

We conclude that, in the presence of redundant requests, the middleware layer, or at least its current implementation, is the bottleneck of the system, tolerating under 3 redundant requests per job during peak hours of job submissions (i.e., one job every 5 seconds). This may inherently preclude the wide-spread use of redundant requests, although one can expect that current implementations as well as the hardware on which the GRAM service runs will improve with time.

5 Impact on Predictability

Although redundant requests are typically employed to obviate the need for resource selection, and thus for queue waiting time prediction, this does not mean that queue waiting time predictions become irrelevant for all users. Users always appreciate having some notion of when their job will start, whether redundant requests are used or not. Also, when submitting redundant requests to multiple clusters with different processor speeds, having estimates of queue waiting times would help in performing resource selection even if redundant requests are used (see the discussion in Section 3.1.2). Therefore, it is interesting to see whether redundant requests make queue waiting times more difficult to predict. One reason why they may is because they cause churn in batch queues. Therefore, users using redundant requests, although benefiting from shorter queue waiting times (as seen in the previous sections), may have a decreased ability to predict these queue waiting times. Furthermore, users not using redundant requests would then experience not only higher but also less predictable queue waiting times.

Batch schedulers can provide an estimate of queue waiting time upon request submission. This can be possible because the job scheduling algorithm assigns

Table 4. Queue waiting time overestimation statistics for $N = 10$ clusters.

	0% jobs using redundant requests	40% jobs using redundant requests (ALL)	
		jobs not using redundant requests	jobs using redundant requests
Average	9.24	77.54	36.28
C.V.	204.98%	189.47%	205.26%

a "reservation" to a request as soon as it is submitted, as with the CBF algorithm. Otherwise, the queue waiting time can be estimated via a simulation of the batch queue. For instance, the `show_guess` command used in the S-Cubed portal [24] performs such a prediction for the Catalina batch scheduler [25]. At any rate, queue waiting time forecasting by the job scheduler can hardly be accurate. For instance, with the EASY job scheduling algorithm, request submissions can delay the execution of a previously submitted request. Perhaps more importantly, queue waiting time forecasts based on analysis or simulation of the queue are always conservative because requested computation times are themselves known to be very conservative [3]. Therefore, running jobs finish executing earlier than expected and pending jobs can start executing earlier as well. Generally, batch schedulers are complex software systems that can be configured in many different ways that affect the job scheduling process. For instance, jobs requesting particular numbers of compute nodes, submitted by particular users, or submitted to a different queue can be assigned higher priority. The arrivals of high-priority jobs, which are difficult to predict, can disrupt the schedule.

We simulate $N = 10$ clusters with no redundant requests. We measure the average ratio of the predicted queue waiting time to the effective queue waiting time and the coefficient of variation of these ratios, across all jobs. Queue predictions were based on the reservations determined by the CBF job scheduling algorithm [15], which was used by all clusters. The left side of Table 4 shows the results, for 50 experiments. One can see that on average queue waiting times are over-predicted by a factor 9.24, with a coefficient of variation over 200%. The high amount of over-prediction is due to the fact that requested compute times are over-estimated by approximately a factor 2.16 on average in our simulations. The right side of the table shows the results for jobs using no redundant requests and for jobs using redundant requests, when 40% of the jobs use the ALL redundant request scheme. For jobs using redundant requests, the queue waiting time is predicted as the minimum predicted queue waiting time over all redundant requests. For both types of jobs, the average overestimation of queue waiting time is dramatically increased: about four times as much for jobs using redundant requests, and about eight times as much for jobs not using redundant requests. There is little difference in the coefficients of variation and one could perhaps just scale the predictions to achieve similar accuracy (or lack thereof) as when no redundant requests are used. We have observed similar results in other experiments, with higher increases in over-prediction for higher numbers of redundant requests per job and/or for higher fractions of jobs employing redundant requests. In all cases, jobs using no redundant requests are penalized.

These results are for predictions based on the state of the queue and on requested compute times. As discussed above, this prediction method, although in use, is known not to be very effective. Statistical forecasting methods such as the ones proposed in [2] have recently been shown to lead to good predictions. It would be interesting to quantify the impact of redundant requests on these promising prediction techniques.

6 Conclusion

In this paper, we have explored whether sending redundant requests to batch-scheduled parallel computing resources causes undesirable effects. Our results show that redundant requests typically improve the performance of jobs that use them, including cases in which all jobs use large amounts of redundant requests. These results hold across ranges of job scheduling algorithms, job interarrival times, number of clusters, and whether requested compute times are exact or conservative. We have also seen that even large numbers of redundant requests (i.e., 30 requests per job) will likely not impose unacceptable load on the batch schedulers or the network.

The above results are to be put in perspective with the performance of jobs that do not use redundant requests. Jobs not using redundant requests are penalized: their stretches increase roughly linearly as the number of jobs using redundant requests increases, and more so when these jobs use more redundant requests. When 80% of jobs send redundant requests to all clusters in a 20-cluster platform, the average stretch of jobs not using redundant requests is 75% higher than when there are no redundant requests in the system. In this case jobs using redundant requests experience stretches that are on average half of those experienced by jobs not using redundant requests. If the jobs using redundant requests send them to only 20% of the clusters, then the stretches of jobs not using redundant requests are only increased by roughly 20% on average. The advantage of users using redundant requests is not dramatic unless they use many such requests and a small fractions of users use none. One may wonder how many users use redundant requests in real systems, and to what percentage of the resources they send these requests. This paper does not answer this question, but our results (namely Figure 4) make it possible to quantify the negative impact on fairness for different values of these percentages.

We have determined that currently available middleware implementations running on currently available hardware would likely not scale to situations in which most users use redundant requests even during peak hours of job submissions. For instance, according to the performance measurements in [23], the GT4 WS-GRAM [22] can only support one job submission-cancellation every 2 seconds. With a job interarrival time of 5 seconds during peak hours (according to the model in [16]), if each job were to use 3 requests, the middleware would not scale. One can however expect that middleware implementations (and the hardware on which they run) will improve with time.

Finally, we also determined that redundant requests reduce the accuracy of queue waiting time predictions based on the state of batch queues. However, these queue waiting time prediction techniques are not very effective in the first place, and thus losses in accuracy may not be an important problem after all. Instead, statistical techniques for predicting queue waiting times are more promising [2]. It would be interesting to explore the effect of redundant requests on these techniques. Conversely, these techniques, if truly effective, may reduce the need for redundant requests as one of their primary goal is to remedy the problem of unpredictable batch queue waiting times in the first place. We will explore this intriguing issue in future work.

The main problems with redundant requests are (i) fairness towards users who do not use them and (ii) load on the middleware. Depending on the fraction of users who use them (which is likely to grow until the establishment and deployment of metaschedulers) and perhaps more importantly on the levels of redundancy, solutions to prevent or limit their use may or may not be necessary. Note that preventing the use of redundant requests in federated distributed platforms in which each resource has its own local scheduler is likely not straightforward.

Acknowledgment

The author wishes to thank Prof. Wolski's MAYHEM Lab in the Computer Science Dept. at the University of California Santa Barbara for providing the cluster and OpenPBS/Maui installation used for the experiments in Section 4. Also, the author would like to thank the reviewers whose comments have greatly contributed to improving this paper.

References

[1] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling — a status report. In *Proc. of the 10th Workshop on Job Scheduling Strategies for Parallel Processing, LCNS*, volume 3277, pages 1–16, 2004.

[2] J. Brevik, D. Nurmi, and R. Wolski. Predicting Bounds on Queuing Delay for Batch-scheduled Parallel Machines. In *Proc. of ACM Principles and Practices of Parallel Programming (PPoPP)*, March 2006.

[3] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling User Runtime Estimates. In *Proc. of the 11th Workshop on Job Scheduling Strategies for Parallel Processing, ecture Notes in Computer Science*, volume 3834, pages 1–35. Springer Verlag, 2005.

[4] S. Srinivasan, V. Subramani, R. Kettimuthu, P Holenarsipur, and P. Sadayappan. Effective Selection of Partition Sizes for Moldable Scheduling of Parallel Jobs. In *Proc. of the 9th International Conference on High Performance Computing (HiPC)*, 2002.

[5] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. In *Proc. of the High Performance and Distributed Conference (HPDC)*, 2002.

[6] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling or PArallel Jobs in a Heterogeneous Multi-Site Environment. In *Proc. of Workshops on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2003.

[7] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of Job-Scheduling Strategies for Grid Computing. *Lecture Notes in Computer Science*, 1971:191–202, 2000.

[8] H. Shan, L. Oliker, and R. Biswas. Job Superscheduler Architecture and Performance in Computational Grid Environments. In *Proceeings of the 2003 ACM/IEEE conference on Supercomputing*, page 44, 2003.

[9] K. Ranganathan and I. Foster. Decoupling Computation and Data Scheduling in Distributed Data-intensive Applications. In *Proc. of the 11th IEEE International Symposium for High Performance Distributed Computing*, pages 352–358, 2002.

[10] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. In *Proc. of the 11th IEEE International Symposium for High Performance Distributed Computing*, pages 359–366, 2002.

[11] A. Bucur and D. Epema. The Performance of Processor Co-Allocation in Multicluster Systems. In *Proc. of the Third IEEE International Symposium on Cluster Computing and the Grid*, 2003.

[12] C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *Proc. of the 8th Workshop on Job Scheduling Strategies for Parallel Processing, LCNS*, pages 85–105, 2002.

[13] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SIMGRID Simulation Framework. In *Proc. of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003.

[14] D. Lifka. The ANL/IBM SP Scheduling System. In *Proc. of the 1st Workshop on Job Scheduling Strategies for Parallel Processing, LCNS*, volume 949, pages 295–303, 1995.

[15] A.W. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Computing*, 12:529–543, 2001.

[16] U. Lublin and D. Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *Journal of Parallel and Distributed Computing*, 63(11), 2003.

[17] Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`.

[18] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. An Integrated Approach to PAralel Scheduling Using Gang-Scheduling, Backfilling, and Migration. In *Proc. of the 7th Workshop on Job Scheduling Strategies for Parallel Processing, LCNS*, volume 2221, pages 133–158, 2001.

[19] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, S. Canon, and H.F. Nielsen. Simple Object Access Prototol 1.1. http://www.w3.org/TR/SOAP, June 2003.

[20] M. R. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis. A Benchmark Suite for SOAP-based Communication in Grid Web Services. In *Proc. of the 2005 ACM/IEEE conference on Supercomputing*, November 2005.

[21] R.A. van Engelen and K. Gallivan. The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In *Proc. of the 2nd IEEE Intnl. Symp. on Cluster Computing and the Grid (CCGrid'02)*, pages 128–135, May 2002.

[22] WS GRAM: Developer's Guide. `http://www.globus.org/toolkit/docs/4.0/execution/wsgram/`, 2006.

[23] Ioan Raicu. A Performance Study of the Globus Toolkit® and Grid Services via DiPerf, an Automated Distributed Performance Testing Framework. Master's thesis, University of Chicago, May 2005.

[24] S-Cubed Portal. `http://www.sdsc.edu/PMaC/S-cubed/`, 2005.

[25] K. Yoshimoto. The Catalina Batch Scheduler. `http://www.sdsc.edu/catalina/`, 2005.