J Grid Computing (2007) 5:235–250
DOI 10.1007/s10723-007-9068-6

Benefits and Drawbacks of Redundant Batch Requests

Henri Casanova

Received: 5 September 2006 / Accepted: 12 January 2007 / Published online: 6 February 2007

Abstract Most parallel computing platforms 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 platforms 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 evaluate the impact of redundant requests on (1) average job performance, (2) schedule fairness, (3) system load, and (4) system predictability. We find that some of the popularly held beliefs about the harmfulness of redundant batch requests are unfounded. We also find that the two most critical issues with redundant requests are the additional load on current middleware infrastructures and unfairness towards users who do not use redundant requests. Using our experimental results we quantify both impacts in terms of the number of users who use redundant requests and of the amount of request redundancy these users employ.

Key words job scheduling · batch scheduling · redundant requests

This work was supported by the NSF under Award 0546688.

H. Casanova (✉)
Department of Information and Computer Sciences,
University of Hawai'i at Manoa, 1680 East–West Rd.,
Post 317, Honolulu, HI 96822, USA
e-mail: henric@hawaii.edu

1 Introduction

Most parallel computing platforms are accessed via batch schedulers [5] 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. 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, which makes them pessimistic. Conversely, they do not take future submissions of high priority requests into account either, which makes them optimistic. Although very recently developed forecasting methods for estimating lower or upper bounds on queue waiting time with certain levels of confidence are promising [1], most users today have at best a fuzzy notion of what queue waiting times to expect. At the same time many of these users have access to multiple batch-scheduled platforms that can be used for running their applications, possibly at different institutions. As a result, rather than picking one target platform based on a poor estimate of queue waiting time, if any, users can send a request to each platform; 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 gaining popularity because it obviates the need for difficult platform 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 attempt to determine the effects of redundant requests. More specifically, we quantify the four following impacts of redundant batch requests:

1. *Impact on average job performance* while redundant requests may intuitively lead to better load balancing across individual platforms, they may also disrupt the resource management policies implemented by batch schedulers and thereby decrease overall job performance. The question is: by how much do redundant requests improve or worsen average job performance?
2. *Impact on schedule fairness* redundant requests give users who use them an advantage as they have the luxury to pick the shortest queue waiting times. The question is: how much of an advantage do redundant request provide and how penalized are users who do not employ them?
3. *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 platforms. The question is: do redundant requests cause any of the system's component to become a bottleneck, and if so, which one?
4. *Impact on predictability* submissions and cancellations of redundant requests cause churn in batch queues, which likely makes them less predictable. The question is: what is the decrease in queue waiting time prediction accuracy when redundant requests are used?

To answer the above questions we use simulations, real-world experiments, and analysis of results obtained by others. We find that several popularly held beliefs regarding the negative effects of redundant batch requests are unfounded. For instance, our experiments show that a batch scheduler, even when running on a mere 1 GHz Pentium III processor, can most likely handle large amounts of request redundancy without becoming a bottleneck. In fact, we find that the two main issues with redundant requests are: (1) additional load on the middleware; and (2) 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, 5, and 6 focus on the four questions above. Section 7 concludes the paper with a summary of our findings and with perspectives on future work.

2 Background and Related Work

Redundant requests can be sent to:

(1) Individual batch queues on multiple platforms;
(2) Multiple batch queues of multiple platforms;
(3) Multiple batch queues of a single platform; or
(4) A single batch queue of a single platform.

In (1), (2), and (3), 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 platforms, as in (1) and (2), a difficulty may be the heterogeneity among these platforms. The computation times requested by each redundant request could be scaled to reflect platform heterogeneity and different numbers of compute nodes could be requested on different platforms. Sophisticated users could thus attempt to tailor their requests to achieve the best response times on each candidate platform. (Note that typical users are not sophisticated, request computation times that are gross overestimations of needed computation times [21], and may not even have a good understanding of the scaling properties of their applications). More importantly, the platform with the shortest queue waiting time could also be a platform 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 platform? Another conundrum arises when using (3) 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 platform. Option (4) 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 known to be difficult [18]. Typically, a larger number of nodes will lead to a longer queue waiting time and to a shorter execution time, while a smaller number of nodes 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 (2): should one wait possibly a long time for a larger number of nodes? Note that option (4) 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 [1], or could use real-time status information from the batch schedulers that gives a sense of the request's place in the queue (unfortunately many currently deployed schedulers do not provide such information). Finally, note that the number of redundant requests that can be used for (2), (3) and (4) can be bounded by each batch scheduler. Indeed, batch schedulers can typically be configured so that each user can only have a limited number of pending requests in the batch queue(s). In this paper we study option (1), use a simple model for generating redundant requests in a heterogeneous environment, and leave options (2), (3), and (4) for future work.

Previous works have explored the use of redundant requests. Most notably, Subramani et al. [19] and Sabin et al. [16] 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 [2, 7, 15, 17] to potentially offload work to remote platforms. A metascheduler serves as a (centralized or distributed) resource broker and thus controls to some extent how a set of individual platforms are shared and used by a community of users. 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. Note that as of today, no widely accepted metascheduler is deployed but users may resort to redundant requests directly. By contrast, we study user-driven redundant requests that may negatively disrupt the schedule at remote clusters. Second, in our study we consider that only some users may be using redundant requests and thus obtain an unfair advantage over users who do not use redundant requests. By contrast, in [16, 19] all users benefit from the same benefits from redundant requests. We argue that in real systems today this is not the case, partly due to the lack of a metaschedulers, but also due to the fact that not all users are created equal: some may not have accounts on multiple platforms, some may not be sufficiently sophisticated to use redundant requests. Another difference between our work and these previous works is that 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. Other relevant related work includes the “placeholder scheduling” technique [13], 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 request submitter (the user or, more likely, a program) may cancel redundant requests.

3 Impact on Average Job Performance

In this section we investigate whether redundant requests negatively impact job scheduling in terms of average job performance. Before presenting our results we detail our experimental methodology and define our metric for job performance.

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-scheduled platforms), because they would be limited to a specific configuration, and because they would hardly be repeatable. We have implemented a simulator using the SimGrid [9] toolkit which 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 requested compute times and numbers of nodes (as discussed in Section 2), but this is not straightforward to model. Instead, we limit heterogeneity to the number of nodes in each cluster and to potentially different workloads at these clusters (i.e., more or fewer requests per second). Each cluster is managed by a batch-scheduler, which can use one of three job scheduling algorithms: EASY [10], Conservative Backfilling (CBF) [12], or First Come First Serve (FCFS). The EASY algorithm enables backfilling and is representative of algorithms running in deployed batch schedulers today. Although widely studied, the more complex CBF algorithm is, to the best of our knowledge, only implemented in the OAR batch scheduler [3]. 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 consider 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. [11], 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 so-called “peak hour” model). Note that [11] goes further by providing a “combined” model that uses two Gamma distributions: one to model job inter-arrival times during peak hours, and one to model the fraction of jobs that arrive during each of the 48 half-hour periods of the day, so as to reflect nocturnal and diurnal trends in the number of submissions. We conducted simulations with the combined model, but the results did not change our conclusions. For simplicity we only present results obtained with the peak hour model. We model the requested number of nodes with a two-stage log-uniform distribution biased towards powers of two. 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 [11], to which we refer the reader for all details. We conducted some simulations using real-world traces made available in the Parallel Workloads Archive [4] but, expectedly, did not observe significantly different results. We opted for using a workload model rather than using 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 traffic. This includes the cost of 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 platform (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 [11], 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 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 platforms. Note that it is shown in [19] that the added cost of using redundant requests when proactively transferring application data to all candidate platforms does not impact the effectiveness of using redundant requests.

For the experiments in this section we ignore all overheads due to the network, the middleware, and the batch scheduler itself so as to isolate the effects of redundant request on job performance. We study these overheads in Section 5.

3.2 Performance Metric

We use a popular metric to assess the average job performance: the *average stretch* over all jobs in the system. The *stretch* 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. The stretch is often called "slowdown" because it measures by how much the job execution is slowed down when compared to execution on a dedicated platform. This metric has been used previously, both in practical works to evaluate the performance of batch schedulers and in theoretical works as objective functions, to be minimized, for job scheduling algorithms (which are ironically not used by batch schedulers). In this paper we often use the *average relative stretch*, that is the stretch relative to that when no redundant requests are used, averaged over all jobs. A value lower than 1 means that the use of redundant requests is beneficial, while a value higher than 1 means that the use of redundant requests is harmful.

We do 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. However, in our specific simulations, our results were essentially unchanged when analyzed with the turnaround time metric.

3.3 Simulation Results

We first simulate an environment that consists of N identical clusters, for $N = 2, 3, 4, 5, 8, 10, 15, 20$. Each cluster contains 128 compute nodes and is managed by a 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 h of job submissions (around 4,000 jobs given that the mean job inter-arrival time for the base model in [11] is roughly

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.