Resource Management Architecture for Fair Scheduling of Optional Computations

Frédéric Camillo[‡], Eddy Caron[†], Ronan Guivarch[‡], Aurélie Hurault[‡], Cristian Klein[*], Christian Pérez[†]
[*] Umeå University, SE-901 87 Umeå, Sweden, firstname.lastname@cs.umu.se
[†] LIP / INRIA, ENS de Lyon, France, firstname.lastname@inria.fr
[‡] University of Toulouse - INPT (ENSEEIHT) - IRIT, firstname.lastname@enseeiht.fr

***Abstract*—Most High-Performance Computing platforms require users to submit a pre-determined number of computation requests (also called jobs). Unfortunately, this is cumbersome when some of the computations are optional, i.e., they are not critical, but their completion would improve results. For example, given a deadline, the number of requests to submit for a Monte Carlo experiment is difficult to choose. The more requests are completed, the better the results are, however, submitting too many might overload the platform. Conversely, submitting too few requests may leave resources unused and misses an opportunity to improve the results.**

This paper introduces and solves the problem of scheduling optional computations. It proposes a generic client-server architecture and an implementation in a production GridRPC middleware, which auto-tunes the number of requests. Real-life experiments show that several metrics are improved, such as user satisfaction, fairness and the number of completed requests. Moreover, the solution is shown to be scalable.

I. Introduction

High-Performance Computing (HPC) resources, such as supercomputers and clusters, are managed by resource management systems, such as batch schedulers [1]. In order for a user to do computations on such resources, she has to submit a pre-determined number of *requests* (also called jobs). For example, in order to execute a Parameter-Sweep Application (PSA), users generally submit one request for each of the parameters that are to be explored. Otherwise, they can coalesce multiple parameters in a single request, as done using the pilot job abstraction [2].

Unfortunately, choosing the requests to submit is cumbersome to do in advance for applications which have **optional computations**, i.e., computations that are not critical to the user, but their completion would improve results. For example, a widely used method to test the numerical stability of complex simulations is *sampling-based uncertainty analysis* [3], such as *Monte Carlo* experiments. Applications range from aerospace engineering to validating nuclear power plant design. At its core, the method consists in varying input parameters and studying the changes in the output parameters. The larger the number of tested input parameters, the better the quality of the results are. Hence, a typical user would like to test as many parameters as possible before a given deadline.

To run such computation on an HPC platform, the user would have to choose a number of requests to submit. If too few requests are submitted, resources might be left idle, thus, the user lost an opportunity to improve her results. If too many requests are submitted, the user might be penalized for overloading the resources and preventing *other* scientists from completing their simulations in due time. Hence, the user faces the difficulty of choosing the number of requests to submit.

Given the current platforms, finding a solution to this problem is difficult. A tremendous amount of work has been dedicated to scheduling HPC applications without [4], [5], [6], [7], [8] or with preemption [9], [10], [11], [12], [13] (for brevity, the list is non-exhaustive), but all of the cited works consider the "amount of work" to be completed a fixed quantity, without considering the possibility to "drop" some work to improve system metrics. In other words, all computations are assumed mandatory.

Differentiating between mandatory and optional computations had been attempted in the context of Cloud computing. Amazon proposes spot instances [14], which are virtual machine instances that can be terminated whenever the Cloud manager chooses to. Spot instances can be used to contain optional computations, as opposed to HPC instances [15] which contain mandatory ones. But even such an approach does not guarantee fairness. Intuitively, there is no guarantee that the system balances resources among the optional computations of each user.

This paper introduces and solves a new scheduling problem: fair scheduling of optional computations. Our contribution is three-fold: first, we present a motivating example and formulate the problem statement of this novel scheduling problem; second, we present a resource management architecture which efficiently solves this problem; third, we evaluate our approach and show through real experiments its feasibility using a production-level GridRPC [16] middleware called Diet [17]. Results show that **user unhappiness** can be reduced to 0 and that **unfairness** can be decreased by up to 150 times.

The remaining of this paper is organized as follows: Section II presents a motivating use-case, which is then formalized into a problem statement in Section III. An architecture which solves the stated problem is described in Section IV, which is evaluated in Section V. Section VI compares our approach with related work. Section VII concludes the paper.

II. Multiple Threshold Pivoting

This section briefly motivates the present work with a use-case brought forward by the Grid-TLSE project [18],

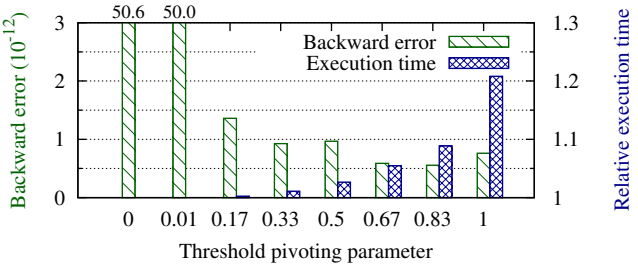

Fig. 1. Example of the results output by a direct solver; backward error is defined as $||b - Ax||/||b||$

a testbed for experts in linear algebra. The project has started in 2002 and is currently being used by 157 users.

Solving a linear system of the form $Ax = b$ usually consists of three steps: analyse, factorization and solve. During the factorization step, the matrices L and U are computed such that $LU = A$. In order to preserve the numerical stability (e.g., avoiding the division by a small number) pivoting can take place. One way to select the pivot is to choose a diagonal entry according to a given threshold. The selection of the threshold is important and for some values, the result of the solution of the linear system can be very bad (see Figure 1).

An often requested feature is to allow a user to run as many factorizations as possible, with different thresholds, until a given deadline is reached. If the deadline is too tight or resources are highly loaded, the system should test at least 3 thresholds, for example. In other words, the user needs to submit some **mandatory requests**, which need to be solved whether the deadline is due or not, and a (potentially large) number of **optional requests**, which the user would like to have computed, but are not as useful as to wait for their completion past the deadline.

The choice of mandatory and optional requests is left to the user. In small data centers with few users, optional requests might be used as a way to be fair towards workmates. In large data centers, users may be charged differently for mandatory and optional requests, thus, declaring their requests as optional may allow them to more efficiently use their quota. Nevertheless, devising a cost model is outside the scope of this paper.

Using classical HPC resource management systems, such as batch schedulers, choosing the number of optional requests to submit to the platform cannot be done (without resorting to workarounds). As a consequence, if too few requests are submitted, the number of tested thresholds is suboptimal. If too many are submitted, the optional requests of other users might not have a chance at getting executing, which would be **unfair**. Even worse, the platform might be so overloaded that the other users might need to wait past their deadlines for the completion of the mandatory requests, which makes them **unhappy**.

Ideally, the system should auto-tune the number of optional requests for each user.

III. Problem Statement

This section formalizes the problem. First, the resource and user models are described. Second, the metrics that the system has to optimize are defined.

A. Resource Model

Let the platform be composed of n_R resources, which are *homogeneous* (a computation request has the same execution time on any resource), *static* (resources are neither added nor removed during execution) and *reliable* (resources do not fail).

This model is somewhat simple but still applicable in many cases. For example, it fairly well approximates production-level multi-cluster systems such as the Decrypthon Grid [19]. Nevertheless, these assumptions will be relaxed in future work.

B. User/Application Model

Let the platform be used by n_U users. A user i enters the system at time $t_0^{(i)}$ (which is not known in advance) and needs to solve at least $n_{min}^{(i)}$ requests (called **mandatory**) and at most $n_{max}^{(i)}$ requests (including mandatory and **optional** requests).

The user sets a "tentative" **deadline** $d^{(i)}$ which acts as follows. If at time $d^{(i)}$ all mandatory requests are completed, the remaining optional requests are cancelled and the user exits the system. Otherwise, the user waits until all mandatory requests are completed, even if this means waiting past the deadline. In the latter case, optional computations can still be executed until the last mandatory request finishes. In other words, the hard deadline is equal to the maximum between the user-provided deadline and the last completion time of the mandatory requests.

To completely characterize the workload, the execution times need to be modeled. We consider that the requests of user i are homogeneous, having the same **execution time** $T^{(i)}$. This is a reasonable approximation for the targeted use-cases (Figure 1), as well as many parameter-sweep applications [20]. However, execution times are not known in advance.

C. Metrics

To evaluate how well a system deals with a workload, the following metrics are of interest: the number of unhappy users, unfairness and the number of completed requests.

The **number of unhappy users** is the number of users who did not complete their mandatory requests before their deadline $d^{(i)}$. These users had to wait additionally, after the tentative, user-provided deadline. Ideally, the number of unhappy users should be 0, provided the workload permits such a solution.

Before defining fairness, let us introduce some helper notations. For each user i, the amount of *deserved* resources $r_{deserved}^{(i)}$ (i.e., the amount the system *should* allocate the user) is computed as follows. The set of users in the system as a function of time is piece-wise continuous. Let

$U^{(j)}$ be the set of users in the system during the time-slot $\left[S^{(j)}, S^{(j+1)}\right)$. The resource area (number of resources times duration) available during that time-slot is divided equally among the users in $U^{(j)}$:

$$r^{(i)}_{deserved} = \sum_{j:i \in U^{(j)}} \frac{n_R \cdot \left(S^{(j+1)} - S^{(j)}\right)}{\#U^{(j)}} \quad (1)$$

Next, for each user i, the satisfaction $s^{(i)}$ is defined as the amount of resources the system allocated her $r^{(i)}_{allocated}$ over the amount of resources she deserved $r^{(i)}_{deserved}$. A satisfaction $0 \leq s^{(i)} < 1$ means that the user i was allocated fewer resources than deserved, while $s^{(i)} > 1$ means that the user i was allocated more resources than deserved. Ideally, the satisfaction of all users should be 1, i.e., they are allocated as many resources as deserved.

Having all prerequisites, let **unfairness** be defined as the difference between the maximum and the minimum among the user satisfactions (unfairness $= \max_i s^{(i)} - \min_i s^{(i)}$). Ideally, unfairness should be equal to 0.

Finally, the **number of completed requests** is a performance-oriented metric, computed as the sum of all the requests (mandatory and optional) belonging to any user that have completed.

To sum up, we aim at finding a system, which minimizes the number of unhappy users, minimizes unfairness and maximizes the number of completed requests, in this order. Note that, the three presented metrics can only be computed a posteriori, after all users exited the system.

IV. DIET-ethic

This section presents the DIET-ethic platform architecture for fair scheduling of optional computations. First, an abstract, implementation-independent description is given. Second, our implementation of DIET-ethic in the production-level DIET GridRPC middleware is detailed.

A. DIET-*ethic Extension*

DIET-ethic is an extension over a client-server architecture. The clients (representing the users of the system) are resource consumers that generate computational requests, while the servers are resource providers, doing computations on behalf of the clients. Clients and servers are connected through a middleware, that implements a discovery mechanism.

Before describing DIET-ethic, let us highlight some design choices. First, we chose to keep the server-side scheduling algorithm simple and make servers unaware of the user deadlines. This choice lets clients, which are under the control of the user, be able to evolve their scheduling algorithms separately from the functionality offered by the server. For example, an extension of DIET-ethic might schedule workflows containing optional computations, without having to change the servers.

Second, the number of requests stored in the platform have to be minimized. This is important, since, for the targeted use-cases, any user could alone fill the whole platform with her optional requests. Therefore, having a system in which all users submit all their requests to the system would clearly not scale.

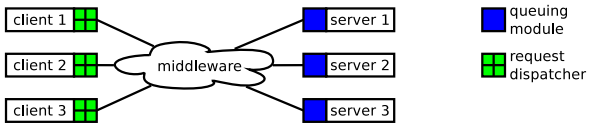

Fig. 2. DIET-ethic architecture

Let us now describe the DIET-ethic architecture consisting of a server-side queuing module and a client-side request dispatcher (Figure 2).

1) Server-side Queuing Module: When a request arrives at a server, it is not immediately executed, but is added to a local queue. A scheduling algorithm is regularly triggered to determine which requests should be started, cancelled (erased from the queue) or killed (prematurely terminated after having been started).

Scheduling Algorithm: First, order the queued requests as follows (earlier rules have higher priority):

1) mandatory requests before optional; this ensures that the number of unhappy users is minimized;
2) started requests before waiting ones; otherwise resources might be wasted as requests are killed and computations completed so-far are lost. In the end, this improves the number of completed requests, paying a small price on fairness;
3) mandatory requests are ordered by submit time, i.e., the First-Come-First-Serve (FCFS) scheduling strategy is used; this allows users who arrived first in the system to get a better chance at completing their mandatory requests before the deadline, thus decreasing the number of unhappy users;
4) order requests by the amount of resources the corresponding user was allocated so far; this ensures server-local fairness;
5) for users having the same amount of allocated resources (such is the case when users enter the system as the same time) a request is chosen randomly; this ensures global fairness, as each server most likely chooses to execute the request of a different client.

Next, if a request gets to the front of the queue it is started. Otherwise, if sorting moves a request from the front of the queue, it is killed. For example, if a user submits a mandatory request to a server which is currently executing an optional request, the latter is killed. Requests can be cancelled on the client's demand.

2) Client-side Request Dispatcher: On the client-side, a custom request dispatcher is required. It works in three phases: setup, monitoring and cleanup.

The **setup phase** starts with a resource discovery, asking the middleware to return at most $n^{(i)}_{max}$ servers. Next, mandatory requests are dispatched to discovered

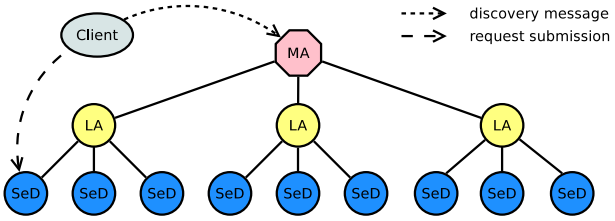

Fig. 3. DIET architecture

servers, for example, in round-robin. Finally, one optional request is submitted to each discovered server.

In the **monitoring phase**, the client enters an event-loop. It stays in this phase until all mandatory requests are completed and the deadline has not expired. When an optional request is completed, the client submits a new optional request to the server which executed the former request. As a result, as long as the user is in the system, provided $n_{max}^{(i)}$ is large enough, each server has at least one optional request in the queue, ready to be executed.

Finally, in the **cleanup phase**, the client cancels all requests that have been submitted but not yet completed. Note that, this phase is entered when all mandatory requests have been completed, thus, only optional requests need to be cancelled.

DIET-ethic is **generic** and can be applied to any client-server architecture. For example, it can be applied to a video conversion platform, accessed through REST or RPC-XML, in which users tag certain videos as mandatory and others as optional. As this paper focuses on improving resource management in HPC data centers, we chose to implement it on top of an existing GridRPC middleware.

B. Application to the DIET Middleware

To simplify access to HPC resources, various programming models have been proposed. One of them is the GridRPC API [16], standardized by the Open Grid Forum. At its core, it extends the familiar Remote Procedure Call (RPC) paradigm to Grid environments.

DIET is a GridRPC middleware. It is composed of the following elements (Figure 3). A **client** is an application that uses the DIET infrastructure to solve problems using a GridRPC approach. A **SeD** (**Server Daemon**) acts as the service provider, exporting functionality through a standardized computational service interface.

The third element of the DIET architecture, **agents**, facilitate service location and collectively provide higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy composed of a **Master Agent (MA)** and several **Local Agents (LA)**.

For implementing DIET-ethic, since DIET already offers many prerequisites, adding a client-side request dispatcher and a SeD-side queuing module were done with little effort. Thus, we obtained a production-ready implementation which is used next to evaluate our proposition.

V. EVALUATION

This section evaluates the proposed architecture. First, we show the gains that can be made with DIET-ethic by comparing it to a standard system which has not been designed to support optional computations. Second, we show that the architecture is scalable. We would like to highlight that all experiments have been done on a **real platform**. Finally, we discuss the benefits that can be observed by an end-user.

A. Gains of Supporting Optional Computations

To highlight the gains of supporting optional computations, let us consider increasingly complex scenarios and make a comparative analysis between DIET-ethic and a system without optional computation support. For the latter, we used the DIET middleware as it was before our contribution: the SeDs serve incoming requests in FCFS order, without distinguishing mandatory from optional requests. On the client-side, we implemented the following behaviour. When a client i enters the system it has to blindly choose $n_{submit}^{(i)}$, a number between $n_{min}^{(i)}$ and $n_{max}^{(i)}$, representing the number of requests to submit. First, it submits $n_{min}^{(i)}$ mandatory and $n_{submit}^{(i)} - n_{min}^{(i)}$ optional requests (in this order). Then, it waits for the mandatory requests to finish. If the deadline has not expired, it sleeps until the deadline is reached. Finally, it gathers the results of all completed requests and cancels the remaining optional requests submitted to the system. To simplify the analysis of the results, all clients "guess" the same value n_{submit}. Let us call this system the **legacy system**.

Before detailing the scenarios, let us present the common methodology. The platform consists of 1 MA and $n_R = 10$ SeDs. The SeDs only implement a sleep service, i.e., the service itself consumes no CPU nor network bandwidth. The platform is used by $n_U = 10$ identical clients with their parameters chosen as follows: to make experiments as useful as possible, but at the same time reduce the time it takes to complete them, we have chosen to "compress" the time: 8 hours are normalized to 100 s. Therefore, we set the execution time $T^{(i)} = 1\,\text{s}$ and the deadlines $d^{(i)} = 100\,\text{s}$. These values are large enough compared to the time scheduling decisions take, yet small enough so that the time of experiments be reasonable. Next, we set the number of mandatory requests $n_{min}^{(i)} = 3$ and the number of total requests $n_{max}^{(i)} = 1000$. These parameters have been chosen so that the following two conditions be met: (i) there is a solution which makes all users happy and (ii) each user can generate enough optional computations to fill all resources.

The above conditions are the ones in which our system is the most interesting to be studied. Otherwise, if the number of mandatory computations is too high, the platform has no choice but to schedule them in a FCFS fashion, being forced to make some users unhappy. Also, if the number of optional requests is too low, its fairness properties cannot be highlighted.

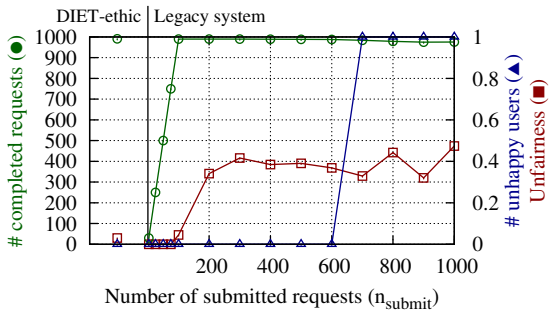

Fig. 4. Results: night-time simultaneous submissions scenario

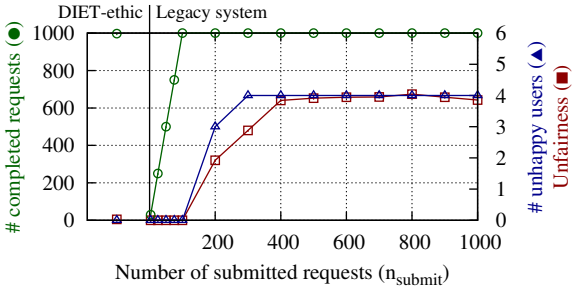

Fig. 5. Results: night-time consecutive submissions scenario

The metrics we are interested in are those presented in Section III-C. All measurements have been done at least 10 times and, since we found deviations to be small, we only plot the median to make graphs more readable.

The figures in this section are structured as follows. The x-axis is divided in two. On the left, the metrics obtained with DIET-ethic are plotted. There is a single data point, since DIET-ethic requires no manual tuning and automatically chooses the number of requests to submit. On the right, the metrics are plotted for the legacy system. There are several data points on the x-axis, representing some of the possible choices for the manually selected number of submitted requests n_{submit}.

The systems are compared in four different, increasingly complex scenarios: night-time submissions, day-time submissions with regular arrivals and two other day-time scenarios with irregular arrivals.

1) Night-time Submissions: Let us start with a simple scenario. Users want to do computations during the night, so that their results are ready in the morning and can be analysed during the workday. Effectively, users enter the computation platform in the evening, just before leaving work and have a tentative deadline for the next morning, when they arrive at work. In our experiments, we can model them by setting the same arrival-time $t_0^{(i)} = 0$ and deadline $d^{(i)} = 100$ for all users.

Figure 4 shows that DIET-ethic managed to find a solution with no unhappy users, with good (almost ideal) fairness, while maximizing the number of completed requests. Regarding the legacy system, one observes that, if n_{submit} is small, the resources are not filled with computation requests, thus, the number of completed requests is suboptimal. However, if n_{submit} is high enough, on average, the legacy system behaves fairly well. This is due to an experiment artifact that, since all users enter the system at precisely the same moment of time, their requests favorably interleave, therefore, the FCFS policy is mostly finding the optimal solution: all mandatory requests are started first, followed by the optional requests.

2) Night-time Consecutive Submissions: However, in production systems, users never enter the platform at exactly the same time. In fact, the time the users enter the system might be quite different: some people leave work earlier, others later. Their deadlines are about the same, since those who leave work earlier, often come earlier the next day. To model this scenario, we insert a very small inter-arrival gap $t_0^{(i)} = i \cdot 0.1\,\text{s}$ and keep $d^{(i)} = 100\,\text{s}$.

Figure 5 shows that, unless all users guess the ideal solution (that of each user submitting exactly $n_{ideal} = 100$ requests), the legacy system either does not manage to optimize the number of completed requests (if $n_{submit} < n_{ideal}$) or makes users unhappy (if $n_{submit} > n_{ideal}$). The latter happens because the FCFS policy fills resources with optional requests of users who arrived early in the system. Therefore, the mandatory requests of users who arrive later start later and can be delayed past the tentative deadline. A similar observation applies to fairness: the FCFS policy favors users who enter the system early, instead of trying to balance requests equally among them.

In contrast, since DIET-ethic distinguishes mandatory and optional requests, it makes sure that mandatory requests have priority over optional ones. Also, instead of favoring users who arrive early, resources are allocated equally among the optional requests of all the users. In the end, DIET-ethic improves fairness up to 150 times and behaves as if all users chose the ideal number of requests to submit, without having to guess it.

3) Day-time Submissions with Regular Arrivals: Let us pass on to a different scenario: day-time submissions. During the day, the users do not enter the system during a short time interval, but are separated by significant inter-arrival times. Let us start with a simple scenario with constant inter-arrival time between consecutive clients.

To model this, we took the night-time scenario and set $t_0^{(i)} = i \cdot 10\,\text{s}$. Except arrival times, all other experimental parameters are kept the same. When comparing the two scenarios, the main difference is that, in the previous one all clients have entered the system before the mandatory requests of the first client are completed. In contrast, in the current scenario, mandatory requests of a client are already completed when the next client enters the system.

The results (not plotted due to space constraints) show that the legacy system behaves best for $n_{submit} = 190$. No users are unhappy, unfairness is low and the number of completed requests is the highest, even when compared to DIET-ethic. The latter happens because, when a new

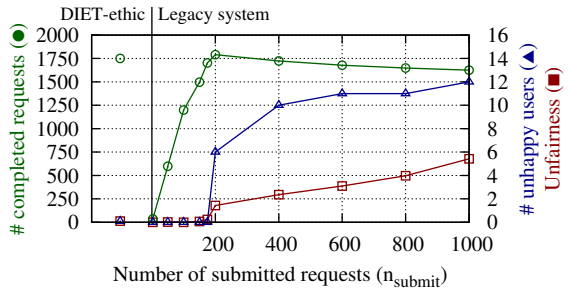

Fig. 6. Results: day-time scenario with irregular arrivals

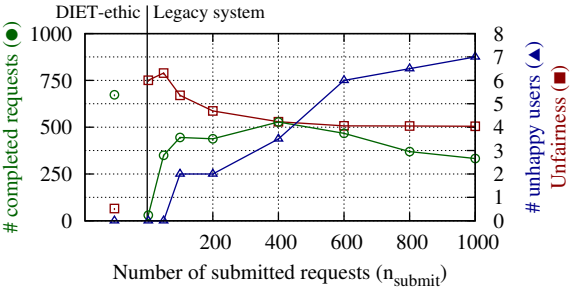

Fig. 7. Results: irregular arrivals and random execution times

client enters the system, DIET-ethic immediately starts its mandatory requests, killing optional request if necessary. Therefore, some started computations are interrupted, thus reducing the number of completed requests. Nevertheless, one observes that DIET-ethic's solution has a lower (0.047, almost ideal) unfairness and the number of completed requests stays competitive to the legacy system (1842 vs. 1890: ≈ -2.5%).

However, on a real platform guessing the best number of requests $n_{submit}^{(i)}$ each client i should submit is difficult, as it depends on a number of factors, such as the number of resources, arrivals and requirements of other users. Some of this information is unknown at the time a client enters the system. When looking at the results for $n_{submit} \neq 190$, one observes that the legacy system is outperformed by DIET-ethic. As in previous scenarios, if fewer requests are submitted, then the number of completed requests is suboptimal. Deviating in the other direction, if too many requests are submitted then unfairness increases. In contrast, DIET-ethic auto-tunes itself and finds a good solution, without requiring any guess.

4) Day-time Submissions with Irregular Arrivals: To make sure that the proposed system is well-behaved in more realistic cases, let us present two more scenarios.

First, the arrivals are assumed to obey the well-known and widely recognized diurnal cycle [21]. To model this, in every second a new client enters the system with a probability given by the polynomial proposed in [22]. Figure 6 shows that, as for the previous scenarios, there is a value of n_{submit} in which the legacy system behaves well. However, this value is difficult to compute a priori in a real system. In contrast, DIET-ethic obtains almost the same values for the targeted metrics without requiring to manually choose this parameter.

TABLE I
GRID'5000 DEPLOYMENT FOR SCALABILITY EXPERIMENT

Cluster	# nodes	Configuration
capricorne	2	2×AMD Opteron 246 @ 2.0 GHz
sagittaire	69	2×AMD Opteron 250 @ 2.4 GHz

Second, we give each client a different execution time by choosing $T^{(i)}$ uniform randomly in $[0.125, 8]$. Requests generated by the same client are still homogeneous and the arrivals are considered to obey the diurnal cycle as in the paragraph above. The results of this scenario, presented in Figure 7, show that no matter how n_{submit} is chosen, the legacy system cannot optimize all targeted metrics. Indeed, due to the large variation in execution times, each user i should choose a different number of requests to submit $n_{submit}^{(i)}$. In practice, when a user i enters the system, optimizing $n_{submit}^{(i)}$ would require complete information about future arriving users (or at least accurate estimations) which is unlikely to be available. In contrast, DIET-ethic auto-tunes itself and manages to minimize the number of unhappy users, minimize unfairness and maximize the number of completed requests.

B. Scalability

To assess the scalability of our solution and measure the overhead, we have designed the following experiment. We reserved the **whole** Lyon site on the Grid'5000 experimental platform [23] (Table I), and divided it into three sets of nodes: 1 client node, 1 MA node and 69 SeD nodes.

Experiments has been done as follows: first, 1 MA has been deployed on the MA node. Second, SeDs have been deployed on each core of the bi-processor SeD nodes, totalling $n_R = 2 \times 69 = 138$ SeDs. Finally, on the client node, $n_U = 100$ client have been launched simultaneously with the parameters: $n_{min}^{(i)} = 3$, $n_{max}^{(i)} = 10000$, $d^{(i)} = 100$, $T^{(i)} = 1$. As a reference, the traces from the Grid Workload Archive [24] contain less that 100 users per day.

Concerning the metrics defined in Section III-C, there are 0 unhappy user and unfairness is low (0.7). Also, the number of completed requests is 13605, which, compared to the maximum of 13800 requests that could have been completed by 138 SeDs in the 100 s deadline, represents 98.6%. Hence, we conclude that the system managed to optimize the targeted metrics, even under stress conditions.

Figure 8 displays the CPU utilization on the client node, the MA node and one of the SeD nodes. On the client node, we have been careful to filter out CPU usage due to process creation and destruction. The SeDs implement a simple "sleep" service, which does not do any computations. Therefore, the measured CPU usage represents the **overhead** our system incurs for managing computational requests. It can be observed that the measured CPU usage before launching the clients and after the clients finished

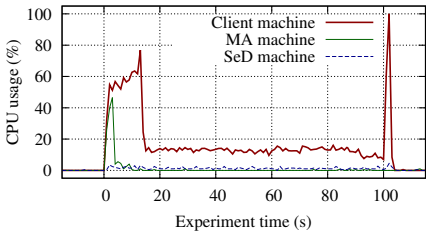

Fig. 8. System overhead: CPU usage for sleep requests

is below 1%, thus indicating a low measurement noise.

When looking at the CPU usage on the client machine, one can clearly distinguish the three phases of the clients (Section IV-A2). During, the *setup* phase ($t \in [0, 13]$) the CPU usage is somewhat high, due to all clients simultaneously discovering SeDs, then submitting requests to all of them. During the *monitoring* phase ($t \in [13, 100]$), one observes that the CPU usage stays at around 15%, as the clients have to submit 138 requests/s to keep SeDs busy. Finally, during the *cleanup* phase, the CPU usage on the client node tops at 100%, as cancellation requests are sent by all clients, simultaneously to all SeDs.

Regarding the other two types of nodes, the CPU usage on the MA node peaks to 48% during the setup phase, but then stays low, since the MA is only participating in this phase. The CPU usage on the SeD nodes is faily low ($< 5\%$) throughout the whole experiment.

Note that the scalability experiment is extreme. The clients arrive simultaneously, which is unlikely in real systems and the execution time is short (1 s). For comparison, the average job inter-arrival time on the LCG Grid is 5 s, whereas the average run-time is 2.5 h [24].

To sum up, the CPU usage on the SeD nodes is negligible when a sleep service is used. This means that our system involves **low overhead** and that SeDs can perform useful computations. The CPU usage on the client node is high during the setup and cleanup phase, nevertheless it managed to generate enough requests, so as to keep SeDs busy. The MA did not prove to be a bottleneck.

C. Real Application Testing

Let us return to the motivating use-case (Section II) and evaluate how DIET-ethic can help the end-user reduce the error when solving a linear system. Previously, when a user wanted to solve a linear system up to a given deadline, she had to choose the number of thresholds to test using a combination of trial-and-error and guesswork. If the user overestimated the number of thresholds, she would manually cancel the computations. Conversely, if the user underestimated, she would have to manually relaunch the computations with a new set of thresholds.

In contrast, thanks to DIET-ethic, a working prototype showed that the user only has to specify the minimum and the maximum number of thresholds to test, and a deadline. Preliminary experiments confirmed that the system is well behaved, as predicted by the synthetic benchmarks above: DIET-ethic auto-tunes the number of requests to minimize the number of unhappy users, minimize unfairness and maximize the number of completed requests.

To sum up, the evaluation shows that DIET-ethic properly supports optional computations. It improves several metrics, while at the same time being scalable.

VI. Related Work

A. Soft Real-time Systems

A system is considered **real-time** if there are tasks that need to be completed before a given deadline. In **soft** real-time systems, some deadlines can be missed. For example, the (m, k)-firm deadlines problem [25] consists in finding a schedule in a real-time system with periodic tasks, so that at least m out of k consecutive tasks meet their deadline. Various algorithms have been proposed that basically divide the tasks into mandatory and optional [26].

Our problem shares some vocabulary (mandatory and optional), but differs in several ways. First, arriving tasks are not periodic. In fact, both their arrival-time and their execution-time are unknown to the platform. Second, it is the user who decides which tasks are to be considered mandatory and which are to be considered optional. Third, the deadline is considered for a group of tasks (i.e., those belonging to the same user) and not for every task individually. Therefore, the solutions brought by cited works are not directly applicable to our problem statement.

Other works deal with offering users soft real-time guarantees on shared resources [27], [28]. However, in contrast to our problem statement, all computation tasks must eventually complete. "Dropping" tasks has been proposed on volatile platforms [29] to improve their throughput, but without considering multi-user issues, such as fairness.

B. Scheduling Parameter-Sweep Applications

Parameter-sweep applications [20] consist of many (usually sequential) tasks, which are relatively short when compared to the execution-time of the whole application. Common frameworks which deal with them, such as BOINC [30] or DIRAC [7], keep all tasks in a centralized master agent from which computation requests are *pulled* by worker agents. This is in contrast to traditional usage of HPC platforms (e.g., imposed by batch schedulers), in which jobs are *pushed* to resources.

Our contribution borrows some concepts both from push and pull scheduling. Mandatory requests are pushed from clients to resources, whereas optional requests are pulled by resources from clients. The former allows clients to dispatch mandatory requests to the most fitting resources, whereas the latter allows to reduce the strain on the platform and produce a natural load balancing.

Two aspects differentiate DIET-ethic from the cited frameworks. First, only a limited number of computation requests are stored on the platform, the rest being generated on-the-fly by the clients. Our design choice is

necessary to ensure the scalability of the system, since it is possible that only a small percentage of optional requests are actually executed: storing them all centrally would overload the system. Second, each client and each server contributes to taking scheduling decisions. This can be considered an advantage to having all decisions taken by a single master agent, since the two entities can evolve their scheduling algorithms separately. Hence, to implement a different scheduling algorithm, the client application can up updated independently of a platform upgrade.

VII. Conclusion

This paper presented DIET-ethic, a generic client-server architecture to efficiently support optional computations. Evaluation has been done using an implementation on top of the production-level DIET GridRPC middleware. Real-life experiments using a synthetic workload showed that several metrics can be improved, for example, user unhappiness can be reduced to 0 and unfairness can be decreased up to 150 times. At the same time, good scalability has been highlighted.

Additionally, evaluation with a real application, GRID-TLSE, showed that a previously unsupported use-case could be efficiently dealt with. Our work thus provides a solution to controlling the load of an HPC platform, while at the same time optimizing resource utilization (and thus return of investment) for resource providers.

As future work we propose improving the scheduling of mandatory requests by reallocating them to better suited servers [31], instead of mapping them at submittal as currently done. Scheduling of optional computations could be further improved by using preemption [32].

Acknowledgments

This work is supported by the French ANR COOP project under contract number ANR-09-COSI-001 (`http://coop.gforge.inria.fr`). Experiments were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (`https://www.grid5000.fr`). Special thanks to Gilles Fedak for his insightful comments.

References

[1] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling: A status report," in *JSSPP*, 2004.

[2] V. Garonne and A. Tsaregorodtsev, "Definition, modelling and simulation of a grid computing scheduling system for high throughput computing," *Future Generation Computer Systems*, vol. 23, no. 8, 2007.

[3] J. Helton, J. Johnson, C. Sallaberry *et al.*, "Survey of sampling-based methods for uncertainty and sensitivity analysis," *Reliability Engineering & System Safety*, vol. 91, no. 10-11, 2006.

[4] 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 Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[5] D. A. Lifka, "The ANL/IBM SP scheduling system," in *Job Scheduling Strategies for Parallel Processing (JSSPP)*, ser. Lecture Notes in Computer Science, vol. 949. Springer, 1995.

[6] B. Sotomayor, K. Keahey, and I. Foster, "Combining batch execution and leasing using virtual machines," in *High Performance Distributed Computing (HPDC)*. ACM, 2008.

[7] A. Casajus *et al.*, "DIRAC pilot framework and the DIRAC workload management system," *Journal of Physics: Conference Series*, vol. 219, no. 6, 2010.

[8] T. Glatard and S. Camarasu-Pop, "A model of pilot-job resource provisioning on production grids," *Parallel Computing*, vol. 37, no. 10-11, pp. 684–692, 2011.

[9] U. Schwiegelshohn and R. Yahyapour, "Analysis of first-come-first-serve parallel job scheduling," in *Symposium on Discrete Algorithms (SODA)*. SIAM, 1998.

[10] Y. Yuan, G. Yang, Y. Wu, and W. Zheng, "PV-EASY: a strict fairness guaranteed and prediction enabled scheduler in parallel job scheduling," in *HPDC*. ACM, 2010.

[11] O. O. Sonmez, B. Grundeken, H. H. Mohamed, A. Iosup, and D. H. J. Epema, "Scheduling strategies for cycle scavenging in multicluster grid systems," in *CCGrid*. IEEE, 2009.

[12] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 306–318, 1992.

[13] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, 2005.

[14] "Amazon EC2 spot instances." [Online]. Available: http://aws.amazon.com/ec2/spot-instances/

[15] "High performance computing (HPC) on Amazon web services." [Online]. Available: http://aws.amazon.com/hpc-applications/

[16] K. Seymour, H. Nakada, S. Matsuoka *et al.*, "Overview of GridRPC: A remote procedure call API for Grid computing," in *Grid Computing, 3rd International Workshop*. Springer, 2002.

[17] E. Caron and F. Desprez, "DIET: A scalable toolbox to build network enabled servers on the Grid," *International Journal of High Performance Computing Applications*, vol. 20, no. 3, 2006.

[18] P. R. Amestoy, I. S. Duff, L. Giraud, J.-Y. L'Excellent, and C. Puglisi, "Grid-TLSE: A web site for experimenting with sparse direct solvers on a computational Grid," in *SIAM Conference on Parallel Processing for Scientific Computing*, 2004.

[19] N. Bard, R. Bolze, F. Desprez *et al.*, "Décrypthon Grid - Grid resources dedicated to neuromuscular disorders," *Studies in Health Technology and Informatics*, vol. 159, 2010.

[20] O. O. Sonmez *et al.*, "Scheduling strategies for cycle scavenging in multicluster Grid systems," in *CCGRID*, 2009.

[21] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation, version 0.34*, 2011. [Online]. Available: http://www.cs.huji.ac.il/feit/wlmod/

[22] M. Calzarossa and G. Serazzi, "A characterization of the variation in time of workload arrival patterns," *IEEE Transactions on Computers*, vol. C-34, no. 2, 1985.

[23] R. Bolze *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed." *International Journal of High Performance Computing Applications*, vol. 20, no. 4, 2006.

[24] A. Iosup *et al.*, "The Grid workloads archive," *Future Generation Comp. Syst.*, vol. 24, no. 7, 2008.

[25] P. Ramanathan and M. Hamdaoui, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *IEEE Trans. Comput.*, vol. 44, no. 12, 1995.

[26] G. Quan and X. Hu, "Enhanced fixed-priority scheduling with (m,k)-firm guarantee," in *Proceedings of the 21st IEEE conference on Real-time systems symposium*, 2000.

[27] T. Cucinotta, D. Giani, D. Faggioli, and F. Checconi, "Providing performance guarantees to virtual machines using real-time scheduling," in *Euro-Par Workshops*, 2010.

[28] S. Gogouvitis *et al.*, "Workflow management for soft real-time interactive applications in virtualized environments," *Future Generation Computer Systems*, vol. 28, no. 1, 2012.

[29] D. Kondo, B. Kindarji, G. Fedak *et al.*, "Towards soft real-time applications on enterprise desktop grids," in *CCGRID*, 2006.

[30] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2004.

[31] Y. Caniou, G. Charrier, and F. Desprez, "Analysis of tasks reallocation in a dedicated Grid environment," in *Cluster*, 2010.

[32] J. Li, M. Qiu, Z. Ming *et al.*, "Online optimization for scheduling preemptable tasks on IaaS cloud systems," *Journal of Parallel and Distributed Computing*, vol. 72, no. 5, 2012.