2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences

Design and Implementation of a Distributed Metascheduler

Janko Heilgeist*, Thomas Soddemann*, and Harald Richter†

*Fraunhofer SCAI, Sankt Augustin, Germany
Email: {janko.heilgeist, thomas.soddemann}@scai.fraunhofer.de

†Clausthal Technical University, Clausthal-Zellerfeld, Germany
Email: harald.richter@tu-clausthal.de

***Abstract*—This paper describes a metascheduler for high-performance computing (HPC) grids that is build upon a distributed architecture. It is modelled around cooperating peers represented by the local proxies deployed by participating sites. These proxies exchange job descriptions between themselves with the aim of improving user-, administration-, and grid-defined metrics. Relevant metrics can include, e.g., reduced job runtimes, improved resource utilization, and increased job turnover. The metascheduler uses peer-to-peer algorithms to discover under-utilized resources and unserviced jobs. A selection is made based on a simplified variant of the Analytic Hierarchy Process that we adapted to the special requirements imposed by the Grid. It enables geographically distributed stakeholders to participate in the decision and supports dynamic evaluation of the necessary utility values. Finally, we identify four intrinsic problems that obstruct the implementation of metaschedulers in general.**

***Keywords*-Grid computing; metascheduling; resource discovery; decision making**

I. Introduction

The problem of optimally scheduling jobs across loosely coupled distributed compute resources is still to be solved. Products such as Gridway [1] or Platform's LSF [2], [3] promise to provide out of the box solutions. On closer examination, most of these solutions still have problems ingrained in their design. The major drawback from our point of view is that despite the fact that the resources are distributed, some of them work in a centralized fashion and resemble a staging queue in a classical batch system. Others are somewhat distributed, but intrusive as far as the interaction with a site's local batch scheduling system is concerned.

Unlike the batch scheduler, a metascheduler supports the exchange of job descriptions across the boundaries of different sites. Such a migration can arise either directly from a user's explicit request for a remote resource or indirectly from a metascheduler's attempt to perform a grid-wide load-balancing. In the latter case, it is the metascheduler's responsibility to discover the best destination. However, existing batch schedulers provide no separate point of entry for metaschedulers. Therefore, a metascheduler can not control the underlying hardware but has to use the site's local batch schedulers — or, more commonly, a grid middleware. It has to extract the job description from a queue at the source site and submit it, customarily assuming the original user's identity, into a target queue. However, the migration should be transparent to the end user, who should, optimally, never notice that his computations were performed non-locally.

The architecture of a metascheduler can be designed either *centralized* or *distributed* [4]. A *centralized* metascheduler is controlled by a dedicated entity that is installed at a single site and has, typically, all the required information for a decision on whether, when and where to migrate a job. That is, it collects data on which sites participate in the grid, which hardware they provide, what the speed of their connection to the grid is, which load their hardware has to bear, and to which degree their queues are filled, etc.

On the other hand, in a *distributed* (or *decentralized*) architecture the metascheduler is split up into multiple independent instances that cooperate among each other. Each instance is separately deployed and represents its site in the grid, that is, it is responsible for all jobs entering and leaving the site via the grid. We call such an instance a *proxy* of the metascheduler. Naturally, a proxy has only limited information to act on compared to the centralized design, as it only gathers data locally and from its neighbors for performance reasons. It is therefore necessary to provide the proxy with sufficient additional input to perform its scheduling. This input can be obtained, e.g., by inter-proxy communication or by a shared pool of available jobs [4].

While the centralized concept of metascheduling is easier to design, implement, deploy, and maintain, the drawbacks nevertheless outweigh the benefits. We see three crucial problems which need to be addressed: First, a centralized metascheduler always represents a single point-of-failure. Problems with the metascheduler will have an immediate impact on the grid because access to remote resources is disrupted. Failures like a broken Internet link separate parts of the grid from the central scheduler. Furthermore, a successful attack on the scheduler compromises the grid.

Second, the centralized design suffers from reduced scalability when faced with increasing demand. The metascheduler represents a serious bottleneck as it is solely responsible for the migration of jobs in the grid. As the number of sites,

978-0-7695-3829-7/09 $26.00 © 2009 IEEE
DOI 10.1109/ADVCOMP.2009.17

users, and jobs grows, it will become more and more difficult to collect all the information that is required to determine the optimal schedule. The common countermeasure to deploy multiple backup servers running the metascheduler software prolongs the decline in performance but leads to additional costs and complexities.

Finally, the largest problem of the centralized architecture is of a political nature. Grids span across multiple administrative zones, companies and institutions, countries, or continents. Each site has its own local policies that a central metascheduler knows nothing about. This can simply be to prefer local users contending for the limited resources or can be as strict as national laws to prevent certain user-groups from accessing HPC resources. Additionally, the willingness of administrators to relinquish control over their hardware to an external entity is unpredictable.

All of the above problems do not emerge in a distributed design. Resilience is increased because a proxy failure affects mostly its home site. Link failures don't disable the grid, as separate sub-grids will continue to function independently. The overall performance scales with the size of the grid as the computation of the schedule is distributed across the grid nodes. And finally, political issues are mitigated by each site's ability to configure its proxy independently.

With only limited information to act on, distributed metaschedulers usually produce sub-optimal schedules. While it would be theoretically possible to achieve full information at every proxy through a complete data exchange between all sites, the cost of $O(n^2)$ is prohibitive and makes this idea infeasible. Determining good schedules is an algorithmic problem that is, as of now, best tackled by contenting oneself with approximations. Therefore, we favor a distributed approach to metascheduling over the centralized design.

In the remaining sections of this paper, we will describe a distributed architecture that is currently being implemented for the Distributed European Infrastructure for Supercomputing Applications (DEISA 2, [5]). There, its decentralized design will be fully exploited in an international grid environment of autonomous sites.

In Section II, we present the architecture of the metascheduler and describe the general interaction between a proxy and its peers. The P2P algorithms required for this communication are portrayed in Section III and the idea of situation-based selection is introduced. Afterwards, the implementation of the metascheduler is explained in Section IV, where we will also present the details of the decision making algorithm. An overview of related work on methods for resource discovery is given in Section V. Finally, we finish this paper in Section VI with a conclusion and an outlook.

II. Architecture

The architecture uses cooperating peers rather than a centralized master. Grid administrators install a local proxy

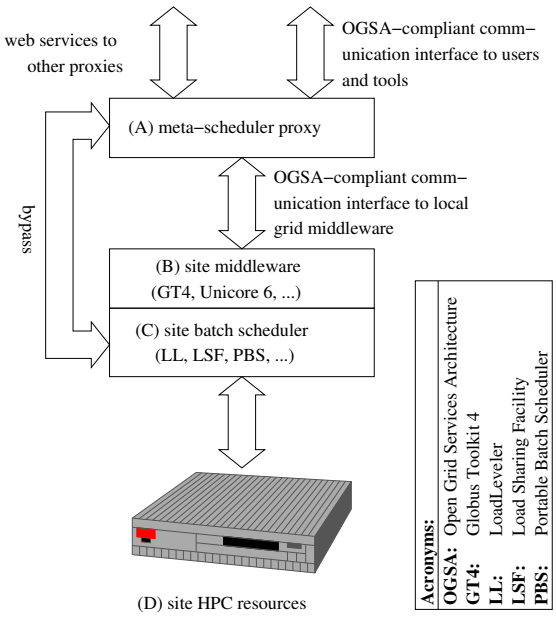

Figure 1. Architecture of the distributed metascheduler at the site level. Relevant references are OGSA [6], GT4 [7], UNICORE 6 [8], LL [9], LSF [2], [3], PBS [10].

software, whose block diagram is displayed in Figure 1. The figure shows the metascheduler proxy (A), the grid middleware (B), and the local batch scheduler (C). The proxy will have to interact with users, remote proxies, grid middleware, and local batch schedulers. Towards the user, the proxy provides web services that are compatible to the standards of the Open Grid Services Architecture (OGSA, [6]). Thereby, users and administrators can continue to use existing tools and client software to monitor jobs and hardware, oblivious to the fact that they communicate with a proxy.

Among themselves, the proxies will communicate using custom-designed but open web services. However, the communication will be mostly restricted to the search for available computing resources and the exchange of scheduling information. For the actual migration of the job descriptions, the proxies rely on the grid middleware. The interface between proxy and middleware is compatible to the OGSA standard. Thus, the proxy can use any OGSA-supporting middleware. Popular examples include, e.g., Globus Toolkit 4 (GT4, [7]) and UNICORE 6 [8].

The actual metascheduling work of the proxies is performed in three stages, that is by 1.) deciding that a job is to be migrated, 2.) discovering available resources, and 3.) deciding to which target site the job is to be migrated. In the first stage, the proxy determines which job to migrate away from the current site. Jobs may be selected for migration because they require resources not available locally, access remote data, or reduce utilization of the resource by fragmenting the schedule.

Afterwards, the *resource discovery* is carried out in an *active* as well as *passive* fashion. In the first case, the proxy actively searches for available resources, while in the second case it only listens for messages sent by other proxies announcing underutilized resources. Having received a collection of offers from its peers, the proxy re-enters the *decision making* phase with the goal to select the best offer to accept. Now, the criteria considered include, e.g., the bandwidth available at the remote site, the cost incurred by using a resource, or other potential benefits offered to the user.

The remote side in this scenario influences the sequence of events twice: first, when it receives a request for computation time, it opts to either provide an offer itself or to ignore the request. Second, if it has disseminated an announcement about available resources, it chooses between all the candidates that took an interest in this offer. The criteria that control the decision in this case could be selected to, e.g., prefer jobs that best utilize the vacant resource, are immediately available for execution, or belong to a site that should be recompensed for previously accepting jobs itself.

The preceding paragraphs exhibited the fact that scheduling is based on local pieces of information and policies. While the examples focused mostly on criteria that resource providers find profitable, the decision making process can be easily extended to include measures that benefit the users or the grid community. Currently, we restrict ourselves to the criteria *queue size*, *average waiting time*, and *waiting time*. They cover an acceptable range of interests insofar as their optimization is designed to result in greater customer satisfaction, increased fairness, and improved utilization of resources. Furthermore, the values can be simply determined and should be deducible from the information provided by any existing local batch scheduler. The real accomplishment will be to find a weighting of the criteria that considers the advantages of all parties involved in a grid environment.

In addition to the illustrated processes, there are other details to be accounted for in the migration of a job description. An offer obtained by a site will usually be valid only for a limited time. The offering site may 1.) reserve the offered timeslot until acknowledged or canceled by the receiving site, 2.) reserve the offered timeslot for a limited time, or 3.) don't provide any guarantees on the period of availability at all. Each of these scenarios requires the receiving site to react accordingly, by either canceling unused offers in a timely fashion, acknowledging offers within their restricted lifetime, or re-issuing a request if previous offers are withdrawn.

In an unstructured grid of independent peers, all these scenarios may occur and have to be supported by a metascheduler proxy. For this reason, our decision making algorithm ranks the offers in order of decreasing preference. It iterates through this list and tries to acknowledge the offers successively. The first offer that is available will be accepted and the remaining offers explicitly canceled. Thus, the algorithm will always return the best possible resource for the job regardless of the types of offers received.

III. P2P-Algorithms

Over the last decade, peer-to-peer (P2P) networks have changed the way a search algorithm's usability is evaluated. With sizes of hundreds of thousands of simultaneous peers, they have rendered most traditional algorithms obsolete. Instead, a whole new class of *distributed hashtable (DHT) based* search algorithms has been created such as CAN [11], Chord [12], Pastry [13], and Tapestry [14]. The combination of a particular network overlay structure with an efficient routing algorithm makes them especially suited to deal with large distributed networks.

Unfortunately, their approach of hashing a search request is not particular useful in a grid environment. Here, most search requests are multi-criteria queries, e.g., "10 nodes with 32 CPUs each for 5 hours", and composed of ranged criteria, e.g., "10–15 nodes". Hashing such a request results in loss of valuable information. Recent research tackles this problem by mapping these extended queries to the routing layers of, e.g., CAN or Chord. We describe three examples of such algorithms in Section V. An excellent overview and a taxonomy for grid-enabled DHT-based algorithms is given in [15].

DHT-based algorithms derive their scalability, speed, and fault-tolerance from the fact that queries are distributed evenly amongst the participants of the network. This spread is a direct result of the mathematical properties of the used cryptographic hashing functions such as SHA-1. However, in a grid environment the necessary difference between the queried values is not always guaranteed. It is typical to encounter limited types of different hardware, common software stacks, and restrictions by resource providers on the requestable numbers of nodes and runtimes of jobs. All of these constraints reduce the possible values an attribute can adopt. As a result, the routing of queries focuses on a few grid nodes and the DHT-based algorithms' benefits are seriously diminished. Considering the cost and complexity and their fairly insubstantial benefits for the targeted grids, we decided against using DHT-based algorithms.

A. Forwarding-based Algorithms

Instead, we employ traditional *forwarding-based* search algorithms. As their name suggests, these algorithms work by forwarding a request from one peer to another peer. Starting at the initial site, the request recursively spreads through the grid until some stop criterion is met. Each receiving peer tries to satisfy the request locally and otherwise forwards it to its direct neighbors. Resource offers take the reverse path of the request back to the initial site.

In most instances, different algorithms vary in the way the request is routed, the stop criterion is chosen, or additional information is collected to augment the routing. Two exemplary algorithms that are representative of the class

of forwarding-based algorithms are the Breadth-First-Search (BFS) and k-RandomWalks.

Breadth-First-Search: The BFS is the simplest of the forwarding-based algorithms without all the bells and whistles other variants attach to it. A request is endowed with an integral "time-to-live" (TTL) that is chosen by the initial site. Each peer that receives the request decrements this TTL by one. If it hasn't reached zero yet, the peer forwards a copy of the request to all of its neighbors indifferently. Afterwards, the peer tries to generate offers matching the request and returns them to the initial site.

Basic details will usually differ between implementations of this algorithm. In general, peers will check whether they have already seen a request before handling and forwarding it a second time. In this spirit, they will not forward a request to the neighbor they originally received it from either. Further, an implementation will usually see to it that each peer aggregates the offers from its neighbors. By bundling the replies at each peer, the number of return messages is kept down. But even with these enhancements, the number of requests nevertheless grows exponentially with the depth of a search. Also, because the requests continue to be forwarded even if a satisfying offer was discovered, there is no chance to abort a search early.

Still, the BFS exhaustively examines the grid up to the search depth. If there is a matching resource on any of the peers it can reach in TTL hops, then the BFS is guaranteed to discover it. In an unstructured distributed grid, the BFS is the only algorithm with this kind of promise. Nevertheless, the algorithm is too expensive to be employed on its own.

k-RandomWalks: The k-RandomWalks algorithm is a variant of the BFS that reduces the number of messages disseminated in the grid. The initial peer issues its request to a maximum of k neighbors. Each subsequent peer will forward the request only to a single random neighbor of its own. Therefore, the number of messages is bound by $k \times$TTL, and any of the two parameters can be adjusted independently.

The BFS' major drawback — its huge cost in terms of messages — is somewhat reduced by k-RandomWalks. By modifying the parameters k and TTL the character of the algorithm can be changed. Increasing or decreasing TTL directly influences the distance that can lie between a requesting site and the potential offers it receives. Increasing or decreasing k directly influences the thoroughness with which the grid is searched for results.

However, regardless of the values of the parameters the algorithm will never yield the quality of results produced by the BFS. Instead, it will often find no results even though a matching resource is nearby. For this reason, the k-RandomWalks is rarely employed by itself as well.

B. Situation-dependent algorithm selection

The balance between proper results and incurred costs is crucial when employing forwarding-based algorithms. Naturally, there are more advanced algorithms in this class than the cited examples BFS and k-RandomWalks. Variants such as, e.g., Adaptive Probabilistic Search (APS, [16]) and Distributed Resource Location Protocol (DRLP, [17]) learn from previously performed searches. They collect data about which neighbor returned promising results and adjust their routing correspondingly. Nevertheless, the costs associated with these variants can not be justified by their results.

Instead of designing yet another forwarding-based algorithm, we chose to examine the existing methods more closely. It emerged that most algorithms are not to be rejected out of hand. Rather, it is the situation that determines the suitability of a method. No algorithm is always perfectly applicable. But given the right circumstances and an intelligently selected technique, the results can be acceptable. Therefore, it is advantageous to investigate the situations in which a search algorithm is to be invoked.

Coming back to our metascheduler, we discovered two main cases in which a P2P algorithm is necessary: I.) *resource requests*, that is, active inquiries into available resources matching specific requirements, and II.) *resource announcements*, that is, disseminations of vacant resources that other peers passively listen for. We further identified two sub cases each: I.a.) *necessary* and I.b.) *optional* requests and, on the other hand, announcements of II.a.) *immediate* and II.b.) *future* resource vacancy. Of these four cases, each makes other demands on an appropriate search algorithm.

The key difference between the main cases lies in the way a particular message needs to be distributed. In I.) the system seeks a resource that matches detailed requirements. Thus, it is beneficial to employ algorithms that, e.g., learn from previously performed searches and can route a request specifically to sites where such a resource is known to exist. However, jobs are transient compared to HPC resources. In II.) a job that can utilize the vacant resource might be located at any site in the grid. Although the probability to find such a job is higher on sites with a matching resource, the remaining sites are not to be neglected.

The difference between the situation's sub-classes lies in the urgency, by which the metascheduler requires its results. In the cases I.a.) and II.a.) a higher priority is placed on the search than in the corresponding counterparts I.b.) and II.b.). Due to the higher priority, it is especially important to find actual results and preferably in a speedy manner. Therefore, the threshold is raised that marks an acceptable cost. Taking the announcements as an example, it is more vital to obtain jobs for an immediately available resource.

Each class of situations requires another type of algorithm. It will always lead to an inferior performance if a single technique is chosen. Therefore, we opted to implement various forwarding-based algorithms and dynamically select the particular method to apply in a situation. Compared to DHT-based algorithms, the forwarding-based algorithms are mostly easy to implement and make no demands on a

specific network overlay structure. With a basic framework in place, the simplicity of implementing these techniques allows us to support several methods simultaneously.

Each time a P2P search algorithm is about to be invoked, the circumstances are analyzed and the situation is categorized into one of the four classes I.a, I.b, II.a, and II.b. Then the appropriate algorithm is determined and the search initiated. Which algorithm is deemed appropriate can be configured by an administrator of a site. While it is necessary that every proxy supports all employed algorithms, each proxy can be differently configured. However, not every combination will actually lead to improvements. Due to the limited amount of space, we refer the interested reader to [18] for feasible configurations.

IV. Software Design

A huge part of the development time of any complex software system is spent dealing with secondary issues such as handling threads, writing web service stubs and skeletons, and managing database access. Therefore, we decided to build the metascheduler as an enterprise application archive (EAR) which is deployable in any application server conforming to the JavaEE 5 standard. The environment provided by a JavaEE application server supports the programmer by taking over many of the more arduous tasks.

Moreover, the user benefits from an easy installation, configuration, and maintenance of the final application. Still, minor adjustments are generally required before an EAR can be deployed in a container as the JavaEE specification grants compliant implementations some leeway. Our development team uses the Apache Geronimo [19] application server because it offers a free, open, and complete implementation of the JavaEE 5 standard. In addition, we plan to actively support Red Hat's JBoss [20] and SUN's Glassfish [21] in the fourth quarter of 2009, too.

The metascheduler is structured into three main modules: 1.) resource discovery, 2.) decision making, and 3.) resource management. Whereas the first two constitute the core modules of the scheduler, the latter embodies the interface to the grid middleware or batch scheduler. Additional auxiliary components provide the glue binding the three main modules together. However, they are secondary to the logical modular design, and we will neglect them for now.

The *resource discovery* module is chiefly responsible for the discovery of remote resources that match certain specified requirements. This task includes the active search for resources as well as the dissemination of vacant resources to other sites. The module incorporates the various forwarding-based P2P algorithms referred to in Section III-A and exposes them as web services to the remote peers. In cooperation with the decision making module, it implements the situation-based selection of one of these algorithms.

In the *decision making* module, we have concentrated the logic that steers the metascheduler. A variant of the Analytic

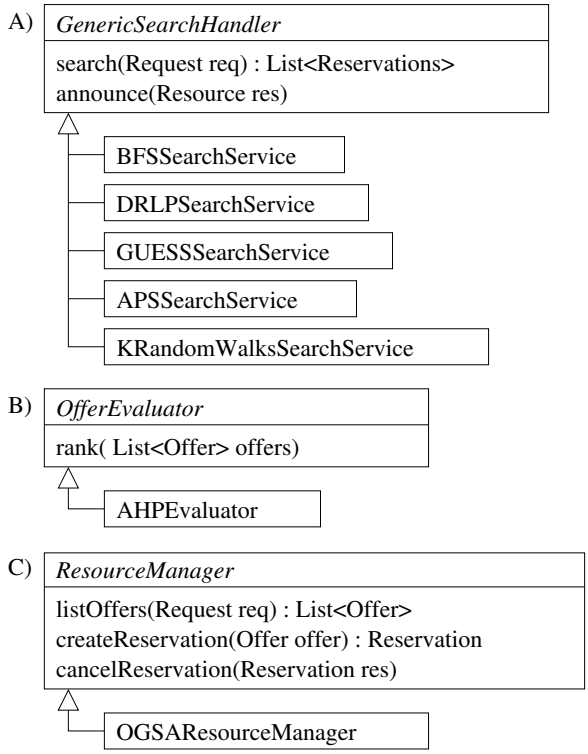

Figure 2. Software modules of the distributed metascheduler.

Hierarchy Process (AHP) [22], [23] is used throughout the system to choose between alternative solutions. Its hierarchical design allows us to consolidate the opinions of several parties into an overall decision. The selection of the target site of a migration can therefore incorporate different viewpoints such as the interests of the job owner, the resource provider, and the grid community. In contrast to the standard AHP, the utilities of an alternative are determined dynamically and with minimal human interaction.

Finally, the *resource management* module serves as an interface between the metascheduler and the lower layers of a site's scheduling stack. It provides a set of abstract methods that allow the reservation and management of timeslots of the underlying hardware resource. Different grid middleware will generally require different implementations of this module. But with an OGSA-compatible default implementation, the metascheduler is adequately equipped to handle a majority of the existing installations on the grid.

A. Resource Discovery

The implementation of the P2P algorithm's web services follows the "contract first" design approach. First, the abstract web services are unambiguously described in the Web Service Description Language (WSDL, [24]). Then, JAX-WS-compliant (Java API for XML Web Services, [25]) tools are used to generate an interface from the description. The interface contains the definitions of the web service methods.

Implementing these methods and deploying the code into an application server results in a usable web service.

Each P2P search algorithm supported by a metascheduler proxy is made available as a separate web service under its own URL. Thus, a proxy is characterized by a bundle of different service URLs. It is therefore reasonable to provide a specialized "get-to-know-service" that serves as a central point of entry for remote peers. It can be seen as some kind of directory that a remote peer can query to identify the additional services offered by the proxy. We call this service the *gatekeeper service*. Note, that this is not a central directory service but a local web service provided by a proxy to supply information about itself in agreement with the distributed nature of the design.

The information supplied by the gatekeeper currently contains a user-friendly name and a list of all locally available web services. The name is used exclusively as a convenience for the users, while the proxies identify themselves by their gatekeeper URLs instead. It can be specified by an administrator to label the managed resource, e.g., "IBM Power6 system at RZG". In the list of services, each available web service is identified by its fully-qualified name (FQN) and mapped to its respective local URL. We established the provided information as scarce and restricted to the absolute minimum. It is easily extendable to include, e.g., a free text description of the resource, usage policies, service level agreements, detailed costs, etc.

The search algorithms are defined via a generic abstract web service displayed in part A) of Figure 2. An actual algorithm needs to implement only two methods: `search` and `announce`. The `search` method takes a resource request as an argument and returns a list of matching reservations. Where these reservations actually come from, e.g., multiple reservations from the queried proxy or additional reservations from remote sites, depends on the particular algorithm considered. Naturally, the reservations will have to contain the appropriate pieces of information that a querying site requires to positively identify the source of the offer.

As of now, the resource request incorporates a requirements definition in the format of the Job Submission Description Language (JSDL, [26]) plus a data block for each the search algorithm and the decision making algorithm. These blocks are used to map a received request to the appropriate algorithm implementation and, additionally, used by the particular algorithms to attach custom data to a request. Regarding the search algorithms, such a piece of custom data can, e.g., include the timeout of a request, the original source of the request, routing directives, etc.

The `announce` method takes a structure similar to the request as an argument but has no return value. The argument's JSDL component is used to describe the offered resource including the address of the offering site. A return value is not required, because a site that wishes to apply for the resource, is required to submit its request in a separate step. Thus, an announcement can be handled as a "fire and forget" message. By removing the reply message for this type of search, an announcement's costs is kept down.

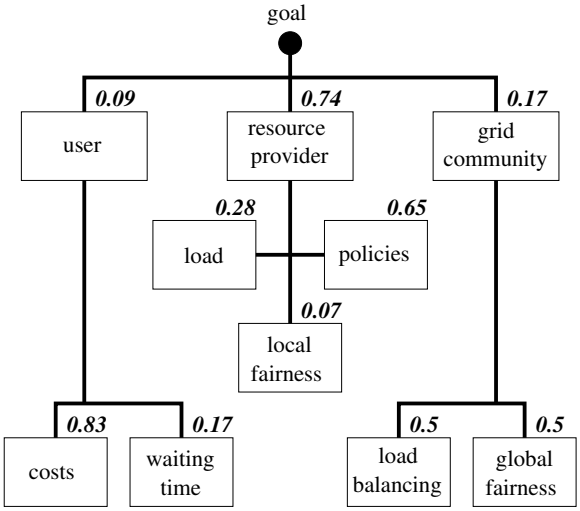

Figure 3. Exemplary AHP hierarchy with multiple parties. The weights of each node's children add up to 1.0.

B. Decision Making

The module in charge of *decision making* is a simplified implementation of the Analytic Hierarchy Process (AHP) [22], [23] displayed in part B) of Figure 2. It considers various facets of an item to sort a given set of similar items in order of preference. The module is applied in several stages of the metascheduling process to rank available timeslots, received offers, migration candidates, etc. By making the algorithm a replaceable module, the different sites could, in theory, employ different decision making modules without interfering with each other.

The AHP is a hierarchical multi-criteria decision making algorithm originally developed for the domain of economics. Its decision is based on a tree of criteria that have been chosen as relevant to the decision at hand. Figure 3 shows an exemplary AHP tree with two levels that combines several criteria relevant to a migration decision. The inner nodes of the tree define meta-criteria which are fully described by their child nodes. Leaf nodes represent criteria where the utility of an option can be explicitly determined. Each criterion in the tree has a weight assigned by the user. These weights determine the importance of the criterion with regard to the corresponding parent node.

The ranking of alternatives starts at the leaves of the tree. Here, each alternative item is evaluated and has its utility computed. In the original AHP, an item's utility is determined based on pairwise comparisons provided by a human decision maker. For this purpose, each pair of items has to be judged and the preference for an item has to be expressed on a scale of nine values. These values represent notions

from "equally preferred" to "extremely preferred" defined by Saaty [22]. A pairwise comparison matrix $A = (a_{ij})_{1 \leq i,j \leq n}$ is the result of this rating process. Finally, the normalized principal Eigenvector of A is determined and its kth entry is used as the utility of the kth alternative.

The utilities with regard to the inner nodes of the tree are computed recursively. At every node, the utility of an alternative is determined as the weighted sum of the corresponding values at the child nodes. That is, for each inner node the utilities at its immediate child nodes are multiplied by the corresponding child's weight. These products are then added for every alternative and result in the alternative's utility with respect to the parent node. As a by-product of the normalized utilities at the leaves and the normalized weights, the process also produces normalized utilities at each inner node. The obtained values are then used at the root node to rank the alternatives from best to worst.

Usually, consent in an AHP context is reached by personal interaction: stakeholders come together to debate the pairwise comparisons. In a grid, the involved parties are geographically distributed, yet the situations where a decision has to be reached are abundant. Because debate and direct interaction are infeasible, we exploit the hierarchical structure of the criteria tree to account for distinct opinions. Each stakeholder is allowed to build its own tree from a common set of criteria and assign its own weights. These separate trees are brought together under a common root to construct the final tree.

At this point, the persistent normalization protects an automated decision making process from malicious users. As the utilities at each subtree's root are normalized, the expressed opinions are initially of equal strength. In the next step, every subtree is connected to the root of the final hierarchy with a different weight. These *root weights* exclusively control the share of participation given to a particular stakeholder. In a grid environment, it is reasonable to assume that resource providers will reserve the right to define these values to themselves. As the values can be defined on a per resource basis, different providers will not interfere with each other. Additionally, a provider can encourage participation — and even advertise the possibility thereof — without loss of control. Due to their importance, we assume that the exact values of the root weights will soon become a sales argument.

In our metascheduler, three stakeholders participate in a decision: 1.) the owner of a job, 2.) the provider of a resource, and 3.) the grid community. The ephemerality of a subtree will vary according to the party that defined it. A user's tree is optionally created as part of the job submission process and permanently attached to the job description. It exists only as long as the job remains in the grid. A provider's tree is specific to a particular resource and part of a proxy's configuration. It will generally be constant for extended periods of time and will be used in every local decision making process. Finally, the grid community's tree is defined at the grid level and assumed to be part of the initial agreement to found a grid. It is configured at the proxy level, too, but identical across participating sites. Typically, this tree will be the most static of the three subtrees. The root weights are configured in the same XML document that sets the provider and grid community subtrees.

Furthermore, we extended the original AHP to dynamically compute the utilities at the leaves of the tree. Such an extension is necessary, before the AHP can be employed in a grid environment. Here, alternatives are defined by measurements or predictions such as waiting time (in minutes), cost (in dollars), and utilization (in percent). Again, AHP is clearly focused on human interpretation of the alternatives. Saaty gives several examples in [23], where a human decision maker's preference may differ significantly from the values suggested by a standard scale. We tried to accommodate this view and simultaneously make the algorithm feasible for a grid environment. Hence, we made the computation of the utilities configurable based on the underlying criterion.

We provide two evaluation methods in the stock installation of the metascheduler: a direct mapping module and a hyperbolic tangent based module. The direct mapping module allows the provider to map ranges of input values to specific utilities. It supports open ranges, ranges bounded above or below, and exact values. The resulting utility function is obviously non-continuous. It might be tempting to use this method for many continuous criteria to express notions such as, e.g., waiting times of up to 5 hours are highly preferred. This is generally a bad idea, because it leads to situations where the priorities do not reflect a stakeholder's intent. A job with a predicted waiting time of 4:59h is in most cases only marginally more preferred than a job with 5:01h waiting time. Thus, this method should only be used for discretely valued criteria such as user IDs, group IDs, queue names, etc.

In contrast, the evaluation method based on a hyperbolic tangent is designed for continuous criteria. It reflects the idea, that distinct utilities should be computed for the range of input values where the majority of the alternatives are located. Above average quality should yield higher utilities but eventually be bounded. An analogous reasoning is true for below average quality. We chose the hyperbolic tangent as the underlying utility function because it converges to 0 (resp. 1) for small (resp. large) input values. The notion of a *control box* is used to scale and shift the hyperbolic tangent with the aim of focusing the transition from 0 to 1 on the average alternatives (see Figure 4).

Upon receiving a set of alternatives, the evaluation module first starts to determine the dominant cluster with regard to the given criterion. It iteratively aggregates the closest input values until a cluster exceeds 51% of the total values. The horizontal position and width of the control box is set to reflect the position and width of this dominant cluster.

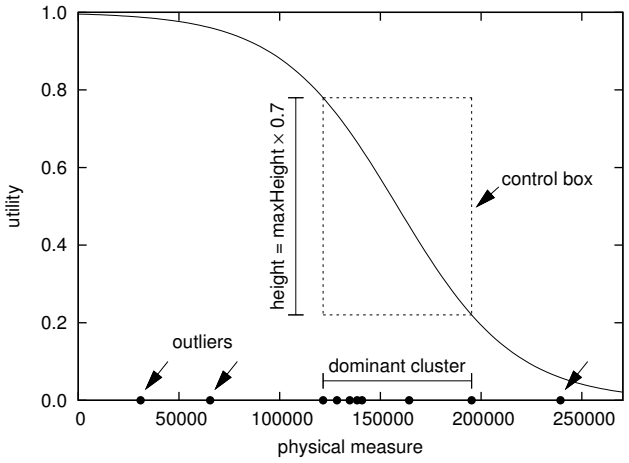

Figure 4. Continuous utility function for minimization criteria based on the hyperbolic tangent. The control box is determined automatically from the distribution of the measures.

Vertically, the box is centered around the midpoint 0.5 of the available utilities. The height of the control box is at most 0.8 and scaled with the ratio of elements in the dominant cluster. Determining the control box is independent of whether the current criterion is about maximizing or minimizing the measures.

Finally, the hyperbolic tangent is scaled and shifted to match the control box, i.e., the curve runs through the lower left and upper right corners of the control box for maximizing criteria. Accordingly, it runs through the upper left and lower right corners for minimizing criteria. The resulting function is used to map the input values to raw utility values which are then normalized to bring them in a form suitable for use in the AHP.

Direct mapping and hyperbolic tangent based evaluation represent two methods designed for non-continuous and continuous criteria, respectively. However, the metascheduler is easily extendable with additional evaluation methods.

C. Resource Manager

The interface between the metascheduler and the locally installed grid middleware or batch scheduler is represented by the resource manager module displayed in part C) of Figure 2. It provides the means to inquire for a set of offers matching a request's JSDL description and, optionally, to fix such an offer into a reservation. The definitions of offer and reservation are deliberately kept as broad as possible to account for the different systems which the metascheduler is required to interact with.

Conceptually, an offer represents a timeslot on some resource that is endowed with additional metadata. The contents of the structure must allow the resource manager to identify the offered resource and assign the job correctly to it. The attached pieces of metadata constitute a mapping of arbitrary criteria to corresponding measures taken or predicted by the resource manager. An offer satisfies the preconditions put forth by the decision making module. Thus, a set of offers can be ranked in order of preference if some weighting of the criteria is supplied externally.

As mentioned previously, the metascheduler currently supports the criteria queue size, average waiting time, and waiting time. These points of comparison were selected because every existing batch scheduler should be able to supply them easily. However, the metascheduler is not restricted to any fixed set but can be configured to understand arbitrary criteria. Thus, the set can be freely extended as long as a resource manager knows how to obtain the related values.

A reservation is just a wrapper around an offer that optionally assigns an owner to it. Whether this assignment is for a limited period of time, until the reservation is explicitly canceled, or not authoritative at all is up to the particular resource manager or its site. The main reason for the separation of offer and reservation was to distinguish between the mere availability of a timeslot and its dedication to a particular party.

V. Related Work

Current grid scheduling solutions focus on centralized resource discovery approaches. UNICORE 6 [8] is a grid middleware that is widely deployed, e.g., in the DEISA 2 grid [5] and in parts of the German D-Grid [27]. It performs resource discovery via a single registry called *Common Information Provider* (CIS). The CIS collects static and dynamic information on the grid's resources which stems from the *CIS Information Providers* (CIP) that monitor each resource. A CIP publishes an Atom-feed that is periodically polled by the CIS by means of a web service protocol. The migration of job steps in a workflow is performed by so-called *Service Orchestrators*. They employ different brokering strategies which in turn use the information from the CIS to reach their decisions.

A similar approach is taken by the second major grid middleware Globus Toolkit 4 [7]. The resource management is contained in the *Monitoring and Discovery Service* (MDS) which consists of *Aggregator Framework*, *Index*, and *Trigger*. The Index is the counter-part to UNICORE's CIS. There are one or more index servers per virtual organization (VO) that store all the information about a VO's resources. The Aggregator Framework is used locally to monitor a resource by periodically spawning shell scripts. The obtained data is then pushed into the central index.

Finally, the Trigger component can be configured to perform actions whenever a predefined event occurs. While the Aggregator Framework is the primary way of updating the index, the Trigger can also be used to poll information into the registry. Globus Toolkit 4 has no support for the automatic migration of jobs by itself. It requires the help of additional software such as the Gridway metascheduler [1] to

provide this functionality. The Gridway information manager accesses the MDS index via its web service interface to discover remote resources. Based on this data it reaches its scheduling decisions. Another centralized scheduling approach that resides on top of GT4 is described in [28].

Distributed designs for resource discovery have previously been produced in theory. Here, the focus has been on the transfer of grid principles to the domain of P2P networks. Examples for such algorithms are MAAN [29], Squid [30], and QuadTree [31]. We sketch these three methods because they represent interesting and unique approaches to the problem of mapping ranged criteria and multi-criteria queries to the underlying DHT-based substrates. Many more grid-enabled DHT-based algorithms are described in [15].

The Multi-Attribute Addressable Network (MAAN) uses two separate procedures to augment the underlying Chord algorithm. First, the multi-criteria requests are resolved by maintaining distinct hashing functions and querying separately for each involved attribute. Second, MAAN employs a *locality preserving hashing function*, that is, a function that maintains the order relation between numerical values for their hashed values. Each query for a ranged criterion is then represented by the actual value desired and the minimum and maximum allowed values. MAAN forwards the request for the actual value from the node responsible for the minimum value via its successors in the Chord ring to the node corresponding to the maximum value. Whatever results have been found at this point are returned to the querying node.

The Squid [30] algorithm is based on a Chord routing layer, too. It interprets the d criteria or attributes relevant to a grid-style query as spanning a d-dimensional space. This space is mapped to a 1-dimensional space by means of *space filling curves* (SFC). The resulting scalars are conventionally hashed and mapped to grid nodes using the Chord routing layer. Each multi-criteria query corresponds to a point or, in case of ranged criteria, a sub-region of the d-dimensional space. It is mapped by the SFC to distinct clusters of scalars. Each cluster corresponds to individual nodes in the grid which have to be queried separately. The results obtained by several of these traditional DHT-queries are then merged to receive the final results of the grid-enabled DHT-query.

Another method that gets by with a single Chord DHT-layer is the QuadTree algorithm [31]. It recursively subdivides the d-dimensional attribute space with the help of quad-trees. Each block is identified by its centroid and mapped to a Chord node. A ranged multi-criteria query intersects with one or more blocks. To execute a query, those relevant blocks are determined and the corresponding grid nodes are contacted. The combined results of these distinct queries form the final response. The performance of the QuadTree algorithm is further improved by a cache. Here, each node stores the addresses of its immediate children in the tree to reduce the number of lookups performed.

VI. Conclusion and Future Work

We have presented our implementation of a hybrid algorithm for a distributed metascheduler that efficiently links available resources and matching jobs. It supports the exchange of jobs between resources and, thereby, achieves improved resource utilization and shorter turn over times. The metascheduler is currently being implemented and will be deployed as part of the DEISA2 grid in fall of 2009.

The design of the metascheduler fully supports the JavaEE specification's security framework. Accordingly, the metascheduler can be extended to integrate existing security infrastructure including virtual organizations, Community Authorization Service [32], and Shibboleth [33]. Towards middleware or local batch schedulers it acts transparently, hence, letting these layers provide their own security arrangements.

During the design and implementation of the scheduler, we gained intense experience with input queues and local resource management systems (LRMS) of HPC sites. In cooperation with the authors of [28], we have isolated four intrinsic problems that have to be solved to make metascheduling a fully working feature in the future.

First, the grid scheduler generally has only the role of a "power user" from the perspective of the LMRS and has to compete with other users, e.g., other grid schedulers in the same grid. As a consequence, an optimal schedule is not possible for any individual metascheduler.

Second, grid schedulers have no control over a site's policy, and, therefore, over the prioritization of the jobs waiting in the queues. Thus, no control exists over the respective local resource management system for the metascheduler.

Third, every site uses a custom configuration of queues, and processors can be shared among queues or dedicated exclusively. Usually, the information from the LRMS does neither allow to reliably determine the number of free processors nor does it allow to determine the total number of processors. The only statistics commonly available are the number of running and waiting jobs. Some sites do not even provide this information because of nondisclosure. To conclude, local schedulers currently do not provide sufficient information for a good schedule.

Finally, resources are highly utilized and waiting times at clusters can last up to hours. Therefore, queue waiting time considerably exceeds the actual execution time for small jobs. Since small jobs constitute the majority of all jobs, the input queue waiting time is the dominant factor. However, input queue waiting time and input queue length have shown to be not continuously differentiable functions over time. Instead, they can vary within minutes by a factor of thousand or more. They behave more like fractals than continuous functions. This makes predictions of future queue waiting times and queue lengths a delicate task. However, scheduling always relies on such predictions. Future work will focus on finding solutions to these obstacles.

REFERENCES

[1] "Gridway," http://www.gridway.org, The Globus Alliance, [checked: 2009-07-17].

[2] "Platform Computing," http://www.platform.com, [checked: 2009-07-17].

[3] S. Zhou, "LSF: Load sharing in large-scale heterogenous distributed systems," in *Proc. Workshop on Cluster Computing*, 1992.

[4] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "Evaluation of job-scheduling strategies for grid computing," in *GRID '00: Proc. 1st IEEE/ACM Intl. Workshop on Grid Computing*. Springer-Verlag, 2000, pp. 191–202.

[5] "Distributed European Infrastructure for Supercomputing Applications," http://www.deisa.eu, [checked: 2009-07-17].

[6] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw *et al.*, "The Open Grid Services Architecture," Open Grid Forum, July 2006.

[7] I. Foster, "Globus toolkit version 4: Software for service-oriented systems." in *IFIP Intl. Conf. on Network and Parallel Computing*, ser. LNCS, H. Jin, D. Reed, and W. Jiang, Eds., vol. 3779. Springer-Verlag, 2005, pp. 2–13.

[8] "UNICORE," http://www.unicore.eu, [checked: 2009-07-17].

[9] *IBM Load Leveler: User's Guide*, IBM Corp., Sept. 1993.

[10] R. Henderson and D. Tweten, "Portable Batch System: External reference specification," NASA Ames Research Center, Tech. Rep., 1996.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proc. 2001 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM Press, 2001, pp. 161–172.

[12] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM '01: Proc. 2001 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM Press, 2001, pp. 149–160.

[13] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware '01: Proc. IFIP/ACM Intl. Conf. on Distributed Systems Platforms*. Springer, 2001, pp. 329–350.

[14] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE J. Sel. Area Comm.*, vol. 22, no. 1, pp. 41–53, 2004.

[15] R. Ranja, A. Harwood, and R. Buyya, "Peer-to-peer based resource discovery in global grids: A tutorial," *IEEE Commun. Surveys Tuts*, vol. 10, no. 2, pp. 6–33, 2008.

[16] D. Tsoumakos and N. Roussopoulos, "Adaptive probabilistic search for peer-to-peer networks," in *P2P '03: Proc. of the 3rd Intl. Conf. on Peer-to-Peer Computing*. IEEE Computer Society, 2003, p. 102.

[17] D. Menascé and L. Kanchanapalli, "Probabilistic scalable P2P resource location services," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 2, pp. 48–58, 2002.

[18] J. Heilgeist, T. Soddemann, and H. Richter, "Algorithms for job and resource discovery for the meta-scheduler of the DEISA grid," *Advanced Engineering Computing and Applications in Sciences, 2007. ADVCOMP 2007. Intl. Conf. on*, pp. 60–66, 4-9 Nov. 2007.

[19] "Apache Geronimo project," http://geronimo.apache.org, The Apache Software Foundation, [checked: 2009-07-17].

[20] "JBoss Enterprise Middleware," http://www.jboss.org, Red Hat Middleware, LLC., [checked: 2009-07-17].

[21] "Sun GlassFish application server," https://glassfish.dev.java.net, Sun Microsystems, Inc., [checked: 2009-07-17].

[22] T. Saaty, *Multicriteria Decison Making: The Analytic Hierarchy Process*, 1988, revised and published by the author; Original version published by McGraw-Hill, New York, 1980.

[23] ——, "How to make a decision: The Analytic Hierarchy Process," *Eur. J. Oper. Res.*, vol. 48, no. 1, pp. 9–26, 1990.

[24] "Web Service Description Language," http://www.w3.org/TR/wsdl, World Wide Web Consortium, [checked: 2009-07-17].

[25] "The Java API for XML-based Web Services (JAX-WS)," http://jcp.org/en/jsr/detail?id=224, Sun Microsystems, Inc., [checked: 2009-07-17].

[26] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva, "Job Submission Description Language," Open Grid Forum, July 2008.

[27] "D-Grid," http://www.d-grid.de, [checked: 2009-07-17].

[28] D. Sommerfeld and H. Richter, "A two-tier approach to efficient workflow scheduling in MediGRID," in *Grid-Technologie in Göttingen - Beiträge zum Grid-Ressourcen-Zentrum GoeGrid*, U. Schwardmann, Ed. Göttingen, Germany: GWDG, 2009, vol. 74, pp. 39–51.

[29] M. Cai, M. Frank, J. Chen, and P. Szekely, "MAAN: A multi-attribute addressable network for grid information services," *4th Intl. Workshop on Grid Computing*, p. 184, 2003.

[30] C. Schmidt and M. Parashar, "Flexible information discovery in decentralized distributed systems," in *HPDC '03: Proc. of the 12th IEEE Intl. Symp. on High Performance Distributed Computing*. IEEE Computer Society, 2003, p. 226.

[31] E. Tanin, A. Harwood, and H. Samet, "Using a distributed quadtree index in peer-to-peer networks," *The VLDB Journal*, vol. 16, no. 2, pp. 165–178, 2007.

[32] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke, "A community authorization service for group collaboration," in *Proc. of the 3rd Intl. Workshop on Policies for Distributed Systems and Networks*, 2002, pp. 50–59.

[33] "Shibboleth — a project of the Internet2 Middleware Initiative," http://shibboleth.internet2.edu, [checked: 2009-07-17].