To be submitted to CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
November 2006

A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-Grid interoperability

Erik Elmroth[†] and Johan Tordsson[‡]

Dept. of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden

SUMMARY

The problem of Grid-middleware interoperability is addressed by the design and analysis of a feature-rich, standards-based framework for all-to-all cross-middleware job submission. The service implements a decentralized brokering policy, striving to optimize the performance for an individual user by minimizing the response time for each job submitted. The architecture is designed with focus on generality and flexibility and builds on extensive use, internally and externally, of (proposed) Web and Grid services standards such as WSRF, JSDL, GLUE and WS-Agreement. The external use provides the foundation for easy integration into specific middlewares, which is performed by the design of a small set of plugins for each middleware. Currently, plugins are provided for integration into Globus Toolkit 4, NorduGrid/ARC, and LCG2. The internal use of standard formats facilitates customization of the job submission service by replacement of custom components for performing specific well-defined tasks. Most importantly, this enables the easy replacement of resource selection algorithms by algorithms that addresses the specific needs for a particular Grid environment and job submission scenario. The algorithms in our implementation perform resource selection based on performance predictions, and provide support for advance reservations as well as coallocation of multiple resources for simultaneous job start. The performance of the system is analyzed with focus on overall job service throughput (up to over 250 jobs/minute) and individual job response time (down to under one second).

KEY WORDS: Grid resource broker; standards-based infrastructure; interoperability; advance reservations; coallocation; service-oriented architecture (SOA); Globus Toolkit; NorduGrid/ARC; LCG2;

[†]E-mail: elmroth@cs.umu.se
[‡]E-mail: tordsson@cs.umu.se
Contract/grant sponsor: Swedish Research Council (VR); contract/grant number: 343-2003-953, 621-2005-3667

1. INTRODUCTION

Following the recent development of Grid infrastructures that facilitates interoperability between heterogeneous resources, there is a somewhat contradictory consequence that a new level of portability problems have been introduced, namely between different Grid middlewares. Although the reasons are obvious, expected, and almost impossible to circumvent (as the task of defining appropriate standards, models, and best practices must be preceded by basic research and real-world experiments), it makes development of portable Grid applications hard. In practice, the usage of largely different tools and interfaces for basic job management in different middlewares forces application developers to implement custom solutions for each and every middleware. By continued or increased focus on standardization issues, we expect this problem to decrease over time, but we also foresee that it will take long time before the problem can be considered solved, if at all. Hence, we see both a need to more rapidly improve the conditions for Grid application development, and for gaining further experience in building general and standards-based Grid computing tools.

We argue that the conditions for developing portable Grid applications can be drastically improved already by providing unified interfaces and robust implementations for a small set of basic job management tools. As a contribution to such a set of job management tools, we here focus on the design, implementation, and analysis of a feature-rich, standards-based tool for resource brokering and cross-middleware job submission. This job submission service, designed with focus on generality and flexibility, relies heavily on emerging Grid and Web services standards both for the various formats used to specify resources, jobs, requirements, agreements, etc, and for the implementation of the service itself.

The service is designed for all-to-all cross-middleware job submission, which means that it takes the input format of any supported middleware and (independently of which input format) submits the jobs to resources running any supported middleware. Currently supported middlewares are the Globus Toolkit 4 (GT4) [25], NorduGrid/ARC [14], and LCG2 [10]. The service itself is designed in compliance with the Web Services Resource Framework (WSRF) [21] and its implementation is based on GT4.

The architecture of this service includes a set of general components, each designed to perform one specific task. The inter-component interaction is supported by the use of (proposed) standard formats, which increases the flexibility by facilitating the replacement of individual components. The service can be integrated for use with a specific middleware by the implementation of a few minor custom components at well-defined integration points.

The service implements a decentralized brokering policy, working on behalf of the user [34, 17]. The resource selection algorithms strive to optimize the performance for an individual user by minimizing the response time for each job submitted. The selection is based on resource information as opposed to resource control, and this is done regardless of the impact on the overall (Grid-wide) scheduling performance. The resource selection algorithms include performing time predictions for file transfers and a benchmarks-based procedure for predicting the execution time on each resource considered. For enhanced quality-of-service, the broker also includes features for performing advance resource reservations and simultaneous coallocation of multiple resources.

The job submission service presented here represents the final version of a second generation job submission tool. For resource allocation algorithms, it partly extends on the algorithms for single job submissions in the NorduGrid/ARC-specific resource broker presented in [17]. With the development of the second generation tools, a transition was made to service-based architectures. Early work on this WSRF-based job submission tool is presented in [16]. The current contribution completes that work and extends it in a number of ways. Hence, one major result of the current article is the completion of the WSRF-based tool into a production quality job submission software. Extensions and other new major contributions include algorithms and software for simultaneous coallocation of multiple resources, support for interoperability with multiple Grid middlewares, as well as a thorough performance analysis of the system. Moreover, this contribution clearly illustrates why and how to use a number of different (proposed) Grid standards in practice when developing an advanced job submission service.

The outline of the rest of the paper is the following. The overall system architecture is presented in Section 2. The resource brokering algorithms used in this implementation are presented in Section 3, including some further discussions on the intricate issues of resource coallocation and advance reservations. Section 4 illustrates how to design the custom components required to allow job submission to and from additional middlewares, by summarizing the steps required for integration with GT4, LCG2, and NorduGrid/ARC. The performance of the system is analyzed in Section 5, followed by a brief presentation of related work in Section 6 and some concluding remarks in Section 7. Information about how to retrieve the software presented is given in Section 8.

2. A STANDARDS-BASED GRID BROKERING ARCHITECTURE

The overall architecture of the job submission and resource brokering service is developed with focus on flexibility and generality at multiple levels. The service itself is made independent of any particular middleware and uses (proposed) standard formats in all interactions with clients, resources, and information systems. It is composed of seven components that perform well-defined tasks in the overall job submission process. Also in the interaction between these components, (proposed) standard formats are used whenever available and appropriate. This principle increases the overall flexibility and facilitates replacement of individual components by alternative implementations. Moreover, some of these components are themselves designed in a similar way, e.g., making it possible to replace the resource selection algorithm inside the component that performs the brokering task.

The service is implemented using the GT4 (Java WS Core) Web service development framework [20]. This framework combines WSRF primitives with the Axis Web service engine [7] in order to facilitate the development of OGSA-compliant Web services. As the service itself is made independent of any particular middleware, all middleware-specific issues are handled by a few, well-defined, plugins. Currently such plugins are available for the GT4, ARC and LCG2 middlewares. A typical set of middleware plugins constitutes less than ten percent of the general code. Descriptions of the middleware-specific components, including a short discussion about their differences, are found in Section 4.

The service supports job submission to and from any middleware for which plugins are implemented, including cross-middleware job submission, as illustrated in Figure 1. The figure illustrates that job requests formulated in any of the job description languages of GT4, ARC, and LCG2 can be sent to the job submission service (denoted JSS in the figure), which can dispatch the job to a resource that runs any (supported) middleware.

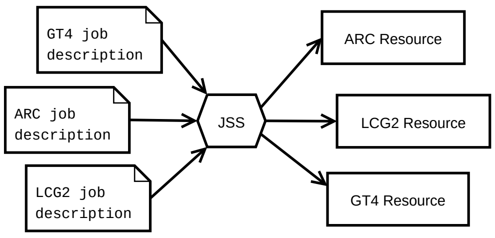


Figure 1. Logical view of interoperability in the job submission service.

In addition to the main job submission and resource brokering service, the framework includes user clients, and an advance reservation component to be installed on the Grid resources. Figure 2 illustrates all component interactions, and all components are briefly presented in sections 2.1–2.3.

2.1. Job submission clients

The client module contains two user clients for standard job submission and for submission of jobs that require coallocation, respectively. The module also includes a plugin for job description translations. An implementation of this plugin converts a job description from the native format specified as input by the user to the Job Submission Description Language (JSDL) [6], a proposed job description format standard developed by the Open Grid Forum (OGF) [45]. The clients can also be configured to transfer job input files stored locally at the client host. This file staging is required if the local files cannot be accessed directly by the job submission service or the Grid resource.

2.2. Job submission service

The clients send the translated job descriptions to the *JobSubmissionService*, which exposes a Web service interface to the functionality offered by the broker, namely submission of a single job and coallocation of a set of jobs. The JobSubmissionService stores information about submitted (or coallocated) jobs as WS-Resources, with state information exposed as WS-ResourceProperties. This mechanism for storing state information in Web Services is specified by the WSRF [21]. Before storing the state information, the JobSubmissionService forwards incoming requests either to the *Submitter* or the *Coallocator*, depending on the type of the request.

The Submitter (or the Coallocator) coordinates the job submission process, a task which includes to discover the available Grid resources, gather detailed resource information, select the most suitable resource(s) for the job, and to send the job request to the selected resource. The main difference between the two components is that the Coallocator performs a more complex resource selection and reservation procedure in order to simultaneously allocate multiple resources. The algorithms used by these two components are presented in sections 3.1 and 3.2, respectively.

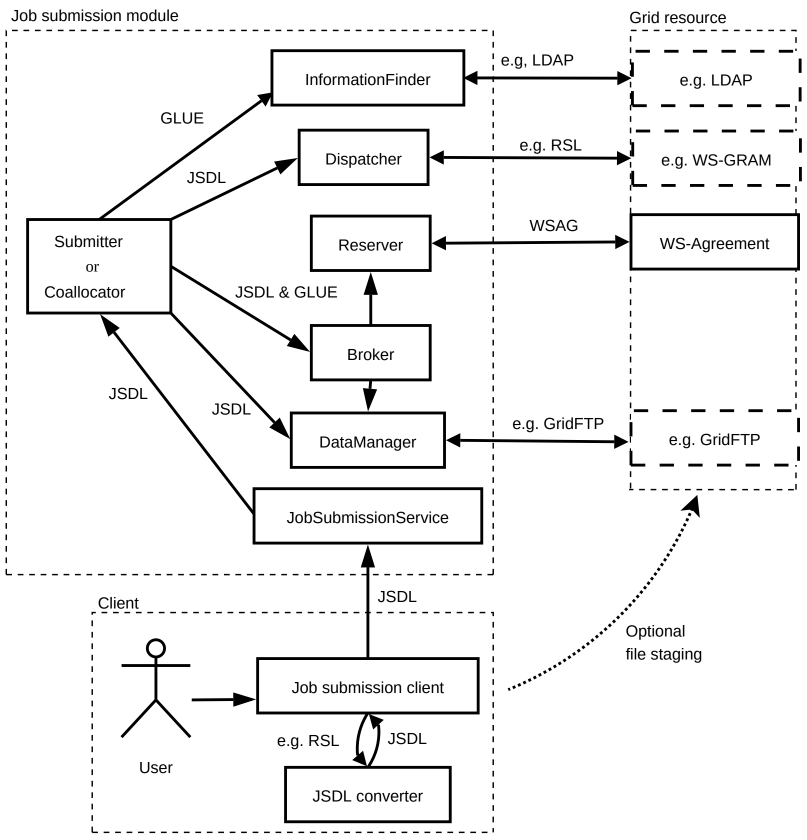


Figure 2. Architecture overview showing components, hosts, and information flow. The boxes show the modules and the dashed lines denote the different hosts.

The resource discovery and information gathering is handled by the *InformationFinder*. Due to differences in the both communication protocols and Grid information formats used by the various Grid middlewares, the InformationFinder consists of three parts each having a middleware-specific plugin. The *ResourceFinder* contacts a higher level index service to retrieve a list of available Grid resources. Its plugin determines which protocol to use and what query to send to the index service. The *InformationFetcher* queries a single Grid resource for detailed information, such as hardware and software configuration and current load. The *InformationConverter* converts the information retrieved from a Grid resource from the native information format to the format specified by the Grid Laboratory Uniform Environment (GLUE) [3]. The GLUE format defines an information model for describing computational and storage resources in a Grid. To improve the performance of the InformationFinder, a threadpool is used for each of the three tasks described above. Another performance improvement is that retrieved resource information is cached for a short period of time, which significantly decreases the number of information queries sent to Grid resources during heavy broker load.

The *Broker* module initially validates incoming job requests, to ensure that a request includes all attributes required, such as what executable to run. Later in the submission process, the Broker is used by the Submitter/Coallocator to rank the resources found by the InformationFinder. The Broker first filters out resources that fail to fulfill the hardware and software requirements specified in the job description, then it ranks the remaining resources after their suitability to execute the job. The resource ranking algorithms may include requesting advance reservations using the *Reserver*, which can create, modify, cancel and confirm advance reservations using a protocol based on the WS-Agreement standard [4]. The details of the advance reservation protocol and the resource ranking algorithms are given in sections 2.3 and 3.1, respectively. When the ranking is done, the Broker returns a list of the approved resources, ordered by their rank.

The Broker may also use the *DataManager* during resource ranking, a module that performs data management tasks related to job submission. This module provides the Broker with estimates of file transfer times, which are predicted from the size and location of each job input and output file. Alternatively, if the Grid middleware supports data replication and/or network performance predictions, the DataManager can use these capabilities to provide better estimates of file transfer time. The DataManager can also stage job input files, unless this task is handled either by the client or by the Grid middleware.

The last module used by the Submitter (or Coallocator) is the *Dispatcher*, which sends the job request to the selected resource. As part of this process, the Dispatcher translates the job description from JSDL to a format understood by the Grid middleware of the resource and selects the appropriate mechanism to use to contact the resource. A plugin structure enables the Dispatcher to perform these tasks without any a prior knowledge of the middleware used by the resource.

2.3. Broker components deployed at the Grid resource

As one part of the job submission framework, we have made an implementation of the WS-Agreement specification [4], to be used for negotiating and agreeing on resource reservation contracts. The module includes an implementation of the AgreementFactory and the Agreement porttypes. However, the Agreement state porttype (which is also part of the WS-Agreement specification) has been left out from the implementation since monitoring the state is not of interest for this type of agreements. It should be remarked that the WS-Agreement implementation itself is completely independent of the service domain (resource reservations) for which it is to be used. We refer to [16] for further details. The WS-Agreement services are the only components that need to be installed on the Grid resource. They enable a client, e.g., the Reserver in the job submission module, to request an advance reservation for a job at the resource.

It should be stressed that it is a priori not known if a reservation can be created on a resource at a given time. The reservation request sent by the client to the AgreementFactory specifies the number of requested CPUs, the requested reservation duration and the earliest and latest acceptable reservation start times, the latter two forming a window of acceptable start times. Three replies are possible:

1. `<granted>` - request granted.
2. `<rejected>` - request rejected and never possible.

3. `<rejected,T_next>` - request rejected, but may be granted at a later time, `T_next`.

Reply number 1 confirms that a reservation has been successfully created according to the request. Reply number 2 typically indicates that the requested resource does not meet the requirements of the request, or that the resource rejects the request due to policy reasons. Reply number 3 indicates a failure, but suggests that a new reservation request, identical to the rejected one but specifying a later reservation start time (`T_next` or later) may be granted.

The client can include an optional flag, *flexible*, in the reservation request to specify that the local scheduler may alter the reservation start time within the start time window after the reservation is created. By allowing this, the local scheduler is given additional possibilities to improve the resource utilization and somewhat compensate for the performance penalty imposed by the usage of advance reservations [19].

Two plugins are required for the advance reservations. One is for interacting with the local scheduler, currently implemented for the Maui scheduler [40], the other one is for admission control of a job that requests to make use of a previously created advance reservation.

Notably, the job submission service can also handle resources that do not have the WS-Agreement services installed, but then, of course, without possibility to make use of the advance reservation feature.

2.4. The optional job preferences document

In addition to the job descriptions given in the native format of any supported middleware, the job submission framework allows for an optional job preferences document. For example, this document can be used to choose between different brokering objectives. The user can, e.g., choose between optimizing for an early job start or an early job completion, and can also specify absolute or relative times for the earliest or latest acceptable job start.

The job preferences document may also include information that can improve the brokering decisions, e.g., specification of benchmarks relevant for the application. By specifying such benchmarks and an expected job completion time on a resource with a specified benchmark result, the resource broker has better support for selecting the most appropriate resources. Notably, the results do not necessarily have to be for standard computer benchmarks only. On the contrary, performance results for real application codes for some test problem is often to recommend.

Just as it is optional to provide the document itself, all parameters in the document are optional.

3. ALGORITHMS FOR RESOURCE (CO)ALLOCATION

The general problem of resource brokering is rather complex, and the design of algorithms is highly dependent on the scheduling objectives, the type of jobs considered, the users' understanding of the application needs, etc. For a general introduction to these issues, see e.g., [61, 49].

In order to facilitate the use of custom brokering algorithms, the job submission service architecture presented in Section 2 is designed for easy modification or replacement of brokering algorithms. The algorithms provided for single job submission and for submission of jobs requiring resource coallocation are presented below in sections 3.1 and 3.2, respectively.

3.1. Resource ranking algorithms

The algorithm for submitting single jobs, implemented in the Submitter module, strives to identify the resource that best fulfills the brokering objective selected by the user. The two alternative brokering objectives are to find the resource that gives the earliest job completion time or the one the gives the earliest job start time.

In order to identify the most suitable resource, the Broker makes a prediction of either the *Total Time to Delivery* (TTD) or the *Total Time to Start* (TTS), respectively, using two different *Selectors*. These predictions are based on time estimation algorithms originally presented in [17]. In order to estimate the TTS, the broker needs to predict the time that will be required for staging of the executable and the input files, and the time that will be spent on batch queue waiting, on all resources considered. In addition, the estimation of the TTD also requires predictions of the job execution time and of the time required for staging the output files. The time predictions for these four tasks are performed by four different *Predictors*, with basic functionality as follows.

Time predictions for file staging. If the DataManager provides support for predicting network bandwidth, for resolving physical locations of replicated files, or for determining file sizes (see Section 2), these features are used by the stage in predictor for estimating file transfer times. If no such support is available, e.g., depending on the information provided by the Grid middleware used, the predictor makes use of the file transfer times optionally provided by the user. Notably, the time estimate for stage in is important not only for predicting the TTD but also for coordinating the start of the execution with the arrival of the executable and the input files if an advance reservation is performed. The stage out predictor only considers the optional input provided by the user, as it without user input is impossible to predict the size of the job output and hence also the stage out time.

Time predictions for batch queue waiting. The most accurate prediction of the batch queue waiting time is obtained by using advance reservations, which gives a guaranteed job start time. If the resource does not support advance reservations or the user chooses not to activate this feature, less accurate estimates are made from the information provided by the resource about current load. This coarse estimation does however not take into account the actual scheduling algorithm used by the batch system.

Time predictions for program execution. The prediction of the time required for the actual job execution is performed through a benchmark-based estimation that takes into account both the performance of the resources and the characteristics of the application. This estimation requires that the user provides the following information for one or more benchmarks with performance characteristics similar to that of the application: the name of the benchmark; the benchmark performance for some system; and the application's (predicted) execution time on that system. Using this information, the application's execution time is estimated on other resources assuming that the performance of the application is proportional to that of the benchmark. If multiple benchmarks are specified by the user, this procedure is repeated for all benchmarks and then the average result is used as a prediction of the execution time. In order to handle situations where resources do not provide all requested benchmarks, the prediction algorithm includes a customizable procedure for making conservative predictions. We remark that a benchmark not at all need to be formal or well-established. It may equally well be a performance number of the actual application code for some predefined problem. The requirement for an (estimate of the) application execution time should furthermore be easy to fulfill as users typically submit the same application multiple times.

Notably, by redefining the selectors and/or predictors, or by defining new ones, the customization of the brokering algorithm is rather straightforward.

3.2. Coallocation

In order to start a Grid job that simultaneously needs a number of resources, a coallocation mechanism is required to make coordinated resource allocations. The algorithm used for performing coallocation is implemented in the Coallocator module (see Section 2), which makes use of the same underlying components as the Selector that allocates resources for single jobs.

The main algorithm for identifying and allocating suitable resources for a simultaneous job start is described in Section 3.2.3. The presentation of the overall algorithm is preceded by a more precise definition of the coallocation problem in Section 3.2.1, and an overview of the main ideas used in the algorithm in Section 3.2.2. In Section 3.2.4, the algorithm is illustrated by a coallocation scenario that highlights some of its key features. This is followed by a discussion of some of the more intricate parts of the algorithm.

3.2.1. Problem definition

The input to the coallocation problem is the following:

1. A set of $n \geq 2$ resource requests (job requests): Jobs $= \{J_1, J_2, \ldots, J_n\}$.
2. A set of n resource sets, where each of the n sets include the resources that are identified to have the capabilities required for one of the jobs:
   Resources $= \{R_1, R_2, \ldots, R_n\}$ where $R_1 = \{R_{11}, R_{12}, \ldots, R_{1m_1}\}, R_2 = \{R_{21}, R_{22}, \ldots R_{2m_2}\}, \ldots, R_n = \{R_{n1}, R_{n2}, \ldots, R_{nm_n}\}$ are the resources that can be used by $J_1, J_2, \ldots, J_n$, respectively. Notably, the same resource may appear in more than one R_i.

A coallocated job requires a matching $\{J_1 \rightarrow R_{1j_1}, J_2 \rightarrow R_{2j_2}, \ldots, J_n \rightarrow R_{nj_n}\}, J_i \in$ Jobs, $R_{ij_i} \in R_i, 1 \leq i \leq n, 1 \leq j_i \leq m_i$ such that J_i has a reservation at resource R_{ij_i}, with all reservations starting simultaneously.

The jobs and resources forming the input to the coallocation problem can be expressed as a bipartite graph as illustrated by Figure 3. An edge between a job and a resource in the graph represents that the resource has the capabilities required to execute the job.

The problem of pairing jobs with resources (by reserving the resources for the jobs) can be viewed as a bipartite graph matching problem. A matched edge in the graph of jobs and resources represents that a reservation for the job is created at the resource. In this context, a coallocated job is a complete matching of the jobs to some set of resources. We note that some resources can execute (or hold reservations for) multiple jobs concurrently, and can hence by matched with more than one job.

3.2.2. Algorithm overview

In overview, the coallocation algorithm strives to find the earliest common start time for all jobs within the job start window $[T_e, T_l]$, where T_e and T_l are the earliest and latest job start time the user accepts. The earliest common job start time is achieved by the creation a set of simultaneously starting reservations, one for each job. For practical reasons, a somewhat relaxed notion of simultaneous job

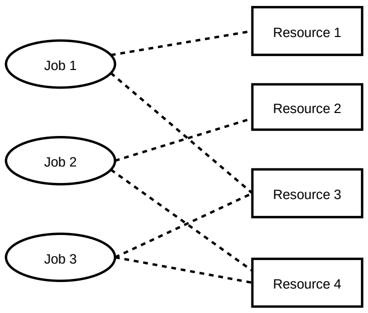


Figure 3. Subjobs and their possible resources viewed as a bipartite graph.

start is used, reducing this constraint to that all jobs must start within the same (short) period of time, expressed as a short time window $[t_e, t_l]$.

The algorithm operates in iterations. Before the first iteration, the $[t_e, t_l]$ window is aligned at the start of the larger $[T_e, T_l]$ window. In each iteration, reservations starting simultaneously, i.e., within the start time window $[t_e, t_l]$, are created for each job. Alternatively, previously created reservations are modified (moved to a new start time window), or reservations are exchanged between jobs, all to ensure that each job gets a reservation starting within the $[t_e, t_l]$ window. The exchange of reservations is performed to improve the matching when a critical resource is already reserved for a job that may use alternative resources. If no reservation can be created for some job during the $[t_e, t_l]$ window, the window is moved to a later time and the algorithm starts a new iteration. This sliding-window process is repeated with additional iterations either until each job has a reservation (success) or the $[t_e, t_l]$ window has been moved too far and t_e exceeds T_l (failure).

3.2.3. Coallocation algorithms

The main coallocation algorithm is given in Algorithm 1 and the procedure for updating augmenting paths is described in Algorithm 2. Further motivation for the most important steps of the algorithms and a discussion of their more intricate details are found in Section 3.3.

The input to Algorithm 1 are the set of jobs and the sets of resources capable of executing the jobs as defined in Section 3.2.1. Additional inputs are the $[T_e, T_l]$ window specifying the acceptable start time interval and ϵ, the maximum allowed job start time deviation.

In Step 1 of the algorithm, the currently considered start time window $[t_e, t_l]$ is set to the start of the acceptable start time interval. This $[t_e, t_l]$ window is moved in each iteration of the algorithm, but its size is always ϵ. The main loop, starting at Step 2 is repeated until either all jobs have a reservation within the $[t_e, t_l]$ window or the $[t_e, t_l]$ window is moved outside $[T_e, T_l]$. In Step 3 of the algorithm, an initially empty set A is created for jobs for which it is neither possible to create a new reservation starting within $[t_e, t_l]$, nor to modify an existing reservation to start within this window. Step 4 of the algorithm defines T_{best}, where to move the $[t_e, t_l]$ window if an additional iteration of the main loop

Algorithm 1 Coallocation

```
Require: A set of $n \geq 2$ resource requests (job requests) Jobs $= \{J_1, J_2, \dots, J_n\}$.
Require: A set of resources with the capabilities required to fulfill these requests. Resources $=$
    $\{R_1, R_2, \dots, R_n\}$ where $R_1 = \{R_{11}, R_{12}, \dots, R_{1m_1}\}$, $R_2 = \{R_{21}, R_{22}, \dots R_{2m_2}\}$, $\dots R_n =$
    $\{R_{n1}, R_{n2}, \dots, R_{nm_n}\}$ are the resources that can be used by $J_1, J_2, \dots, J_n$, respectively.
Require: A start time window $[T_e, T_l]$ specifying earliest and latest acceptable job start.
Require: A maximum allowed start time deviation $\epsilon$.
Ensure: A set $n$ of simultaneously starting reservations, one for each job in Jobs.
 1: Let $t_e \leftarrow T_e$ and $t_l \leftarrow T_e + \epsilon$.
 2: repeat
 3:     Let $A \leftarrow \emptyset$ be the set of jobs for which path augmentation should be performed.
 4:     Let $T_{\text{best}} \leftarrow \infty$ be the earliest time later than $[t_e, t_l]$ that some reservation can start.
 5:     for each job $J_i \in$ Jobs, $1 \leq i \leq n$, that does not have a reservation starting within $[t_e, t_l]$ do
 6:         if $J$ already has a reservation starting outside (before) $[t_e, t_l]$ then
 7:             Modify the existing reservation to start within $[t_e, t_l]$.
 8:         if Step 7 fails, or if $J_i$ had no reservation then
 9:             Create a new reservation starting within $[t_e, t_l]$ for $J_i$ at some $r \in R_i$.
10:         if Step 9 fails then
11:             Add $J_i$ to $A$.
12:             Let $T_{\text{best}} \leftarrow \min\{T_{\text{best}}$, the earliest T_next value returned from Step 9$\}$.
13:     if each job $J \in A$ may be augmented then
14:         for each job $J \in A$ do
15:             Find an augmenting path $P$ starting at $J$ using breadth-first search.
16:             Update reservations along the path $P$ using Algorithm 2.
17:             if Step 16 fails then
18:                 Let $T_{\text{best}} \leftarrow \min\{T_{\text{best}}$, the earliest T_next value returned from Step 16$\}$.
19:     if some job in $J$ has no reservation starting within $[t_e, t_l]$ then
20:         Let $t_l \leftarrow T_{\text{best}}$ and $t_e \leftarrow (t_l - \epsilon)$.
21:         if $t_e > T_l$ then
22:             The algorithm fails.
23: until all jobs have a reservation starting within $[t_e, t_l]$
24: Return current set of reservations.
```

is required. To ensure termination of the main loop in the case when all reservation requests fail, and no reservation ever will be possible (reply number 2 in the reservation protocol), T_{best} is set to infinity. This variable is assigned a finite value in steps 12 and 18 if any failed reservation request returns a next possible start time (reply number 3 in the reservation protocol).

Step 5 is performed for all jobs that have no reservation within the $[t_e, t_l]$ window. This applies to all jobs unless the window has been moved less than ϵ since the last iteration, in which case some previously created reservations still may be valid. In Step 6, it is tested whether the job already has a reservation from a previous iteration, that starts too early for the current $[t_e, t_l]$ window. If so, this reservation is modified in Step 7 by requesting a later start time (within the new $[t_e, t_l]$ window). The condition in Step 8 ensures that Step 9 is only executed for jobs that have no reservation, either because no reservation could be created in the previous iteration of the algorithm or because the job has lost its reservation due to a reservation modification failure. In Step 9, a new reservation is created for the job by first trying to reserve the resource highest ranked by the broker, and upon failure retry with the second highest ranked resource etc., until either a reservation is created or all requests have failed. In case one or more reservation requests in Step 9 receive reply number 3 (“`<rejected, T_next>`”) the earliest of these `T_next` values is stored for usage in Step 12.

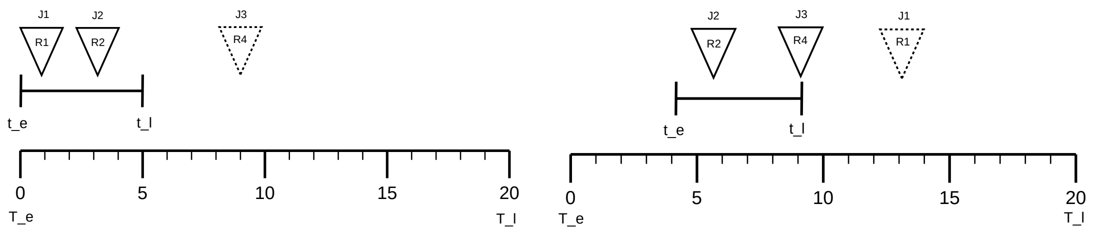
Step 10 tests if all reservation requests have failed for a job J_i, and if so, J_i is included in A in Step 11 to be considered for path augmentation later. Step 12 updates T_{best} with the earliest `T_next` value returned during Step 9, if such a value exists. In Step 13, it is tested whether augmentation can be used for each job in A. This test is done for a job J by ensuring that some other job J' holds a reservation for a resource that J can use. If no such other job J' exists, there is no reservation to modify to suit job J and no augmenting path can hence be found. As the goal of the algorithm is to match all jobs, Step 13 ensures that all jobs in A are eligible for augmentation. It is of no use if the current matching can be extended with some, but not all, unmatched jobs.

If augmentation techniques can be used according to the test in Step 13, the loop in Step 14 is executed for each job in A. In Step 15, an augmenting path of alternating unmatched and matched edges, starting and ending in an unmatched edge, is found using breadth-first search. In Step 16, the reservations (matchings) along this augmented path are updated using Algorithm 2. The path updating algorithm includes both modifications of existing reservations and creation of new ones. If any of these operations fail and return reservation request reply number 3, the earliest `T_next` value is, analogous with Step 9, stored for usage in Step 18. Step 17 tests if the update algorithm failed, and if so, Step 18 updates T_{best}. Step 19 tests whether the algorithm will terminate, or if another iteration is required. In the latter case, the $[t_e, t_l]$ window is updated in Step 20. In order to move the window as little as possible, t_l is set to T_{best} and t_e is updated accordingly. Step 21 ensures that the $[t_e, t_l]$ window is still within the $[T_e, T_l]$ window. Unless this is the case, the algorithm fails (Step 22). Once the loop in Step 2 terminates without failure, the coallocation algorithm is successful and the current set of reservations is returned (Step 24).

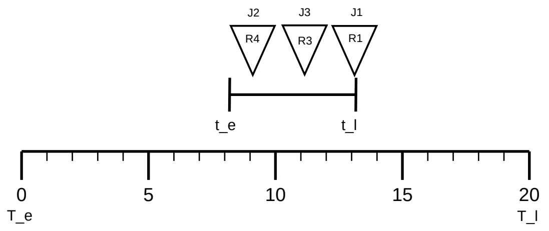
Algorithm 2 Update augmenting path

```
Require: An augmenting path $P = \{J_1, R_1, \ldots, J_n, R_n\}, n \geq 2$ where $R_i$ is reserved for $J_{i+1}$.
Ensure: An augmenting path $P = \{J_1, R_1, \ldots, J_n, R_n\}, n \geq 2$ where $R_i$ is reserved for $J_i$.
1: Create a new reservation for $J_n$ at $R_n$.
2: if Step 1 fails then
3:     The algorithm fails.
4: for $i \leftarrow (n-1)$ downto 1 do
5:     Modify the existing reservation at resource $R_i$ for job $J_{i+1}$ to suit job $J_i$.
6:     if the modification in Step 5 fails then
7:         The algorithm fails.
```

Algorithm 2 is used by Step 16 of Algorithm 1 to modify the reservations along an augmented path. In Step 1 of Algorithm 2, a new reservation is created for job J_n, as the existing reservation for this job will be used by job J_{n-1}. If the creation of the new reservation fails, it is of no use to modify the existing reservations, and the algorithm fails (Step 3). The loop in Step 4 is performed for all



(a) After the first iteration. (b) After the second iteration.



(c) After the third and final iteration.

Figure 4. Example execution of the coallocation algorithm.

previously existing reservations. In Step 5, the reservation currently created for job J_{i+1} is modified to suit the requirements of job J_i. This modification typically includes changing the number of reserved CPUs and the duration of the reservation, but the reservation start time is never changed. Step 6 tests if the modification fails. If so, it is not meaningful to continue the execution and Step 7 terminates the algorithm (with failure).

3.2.4. Example execution

The following example illustrates the execution of the coallocation algorithm. Let the input to the algorithm be Jobs $= \{J_1, J_2, J_3\}$ and Resources $= \{\{R_1, R_3\}, \{R_2, R_4\}, \{R_3, R_4\}\}$. This scenario corresponds to the bipartite graph shown in Figure 3. Let the start time window $[T_e, T_l]$ be $[0900, 0920]$ (20 minutes) and let the maximum allowed start time deviation, ϵ, be 5 minutes. Notably, we have for clarity kept $[T_e, T_l]$ rather small. In practice, its size may vary from a few minutes to several hours or even days. The size ϵ of the small start time window $[t_e, t_l]$ is however typically only a few minutes.

In the first iteration of the algorithm, a reservation is created for J_1 at R_1 and one for J_2 at R_2. These reservations are shown as solid triangles in Figure 4(a). However, no reservation starting early enough can be created for J_3. The earliest possible reservation (at R_4), which would start a few minutes too late, is shown as a dashed triangle in Figure 4(a) . Path augmentation techniques cannot be used for J_3

as there is no other job holding a reservation for a resource that J_3 can use, i.e., neither J_1 nor J_2 has a reservation at R_3 or R_4, which are the only two resources meeting the requirements of J_3.

In next iteration, the $[t_e, t_l]$ window is moved and aligned with the earliest possible reservation start for J_3. A reservation for J_3 is created at R_4. The reservation for J_2 at R_2 is modified to start within the new $[t_e, t_l]$ window. These two reservations are represented by the solid triangles in Figure 4(b). The reservation for J_1 at R_1 cannot be moved to within $[t_e, t_l]$, and is hence implicitly cancelled. Furthermore, no new reservation can be created for J_1 within $[t_e, t_l]$. Path augmentation techniques cannot be used to create an additional reservation as neither J_2 nor J_3 has reserved one of R_1 and R_3. The earliest possible reservation for J_1 (at R_1) is shown as a dashed triangle in Figure 4(b).

In the next iteration, t_l is set to the earliest possible start of J_1 and t_e is adjusted accordingly. The reservation that in the previous iteration was possible for J_1 at R_1 is created, illustrated by a solid triangle in Figure 4(c). The reservation for J_3 at R_4 already starts within $[t_e, t_l]$ and requires no modification. For job J_2 the existing reservation cannot be moved to within $[t_e, t_l]$ and is hence cancelled. It is furthermore not possible to create a new reservation. The path augmentation algorithm can however be applied. Starting from J_2 in the bipartite graph in Figure 3, a breadth-first search is performed according to Step 15 of Algorithm 1. This search finds a resource that J_2 can use (R_4), which is currently reserved by another job (J_3), which in term can use another resource (R_3). The resulting augmenting path is $\{J_2, R_4, J_3, R_3\}$. Next, Algorithm 2 is invoked with this path as input. The algorithm creates a new reservation for J_3 at R_3, and modifies the existing reservation for J_3 (at R_4) to suit J_2. The resulting reservations (for J_2 and J_3) are shown as solid triangles in Figure 4(c). Since each job has a reservation starting within $[t_e, t_l]$ (and inside $[T_e, T_l]$), the coallocation algorithm terminates and the coallocation request is successful.

3.3. Discussion of Quality of Service issues

We now discuss advance reservations and the properties of the bipartite matching algorithm in more detail, including motivating the usage of path augmentation techniques.

3.3.1. Regarding the use of advance reservations and coallocation

It should be remarked that how and to what extent advance reservations should be used, partly depends on the Grid environment. The current algorithms are designed for use in medium-sized Grids, and with usage patterns where the majority of the jobs do not request advance reservations. In Grids where hundreds or even thousands of resources are suitable candidates for a user's job requests, the algorithms requesting advance reservations should be modified to first select a subset of the resources before requesting the reservations. In order to allow a majority of the Grid jobs to make use of advance reservations, it is probably necessary to have support for, and make effective usage of, the flag "flexible" in all local schedulers, in order to maintain an efficient utilization of the resources.

3.3.2. Modifying Advance Reservations

The coallocation algorithm modifies existing reservations as if the modify operation is atomic, even though the current implementation actually first releases the existing reservation and then creates a new one. The reason is that the Maui scheduler [40], one of the few batch system schedulers that support advance reservations, has no mechanism to modify an existing reservation. This means that the modification operation, in unfortunate situations, may loose the original reservation even if the new one could not be created. This occurs when the scheduler decides to use the released capacity for some other job before it can be reclaimed. However, failures during reservation modifications are non-fatal in our coallocation algorithm, as the path updating algorithm (Algorithm 2) creates the new reservations before modifying the existing ones.

We also remark that the WS-Agreement specification [4] does not specify an operation for renegotiation of an existing agreement (reservation). A protocol for managing advance reservations, including atomic modifications of existing reservations is discussed in [48]. To the best of our knowledge, there exists neither an implementation of this protocol nor a local scheduler with the reservation mechanisms required to implement it. Atomic reservation modifications may very well be included in future versions of the WS-Agreement standard (or defined by a higher level service, such as the currently immature WS-AgreementNegotiation [5]) and supported by new releases of batch system schedulers. If so, then the coallocation algorithm itself needs no modification, and it furthermore becomes more efficient, as failed reservation modifications causes extra iterations of the algorithm to be executed.

3.3.3. Properties of the bipartite matching algorithm.

In the bipartite graph representing jobs and resources, an edge between a job and a resource denotes that the resource has the capabilities required to execute the job. We can however not know a priori that the resource actually can be reserved for the job at the time requested. Seen from a graph theoretic perspective, it is not certain that the edges in the bipartite graph actually exist (e.g., at a particular time) before we try to use them in a matching. Given the above facts, it is not possible to completely solve the coallocation problem using a bipartite matching algorithm that precalculates the matching. Therefore, we use a matching algorithm that gradually increases the size of the current matching (initially containing no matched edges at all), and use path augmentation techniques to resolve conflicts.

3.3.4. Path augmentation considerations.

Path augmentation techniques are used when the coallocation algorithm fails to reserve a resource for a job, but it is possible that this situation can be solved by moving some other reservation (for the same coallocated job) to another resource. In order to reduce the need for path augmentation, we strive to allocate resources in decreasing order of the “size” of their requirements, even though it is in the general case not possible to perfectly define such an ordering. For example, one job may require two CPUs with one GB memory and another job only one CPU but with two GBs memory. To improve the order of the reservation requests, the jobs are sorted based on the requested number of CPUs, required memory, and the requested job runtime before the coallocation algorithm is invoked. Hence, the number of times path augmentation is used is reduced. The usage of breadth-first search when finding augmented paths guarantees that the shortest possible augmented path is found. Both the initial sorting of the job list and the usage of breadth-first search reduces the number of reservation modifications. This both improves the performance of the algorithm as the updating of an augmented path is time-consuming (see Section 5 for more details), and reduces the risk of failures that occur due to the non-atomicity of the update operation as described in Section 3.3.2.

It should also be remarked that the test performed in Step 13 of Algorithm 1 may cause false positives, since the assumption that there is a possibility for path augmentation is done before actually performing the advance reservations required to augment the path. However, no false negatives are possible, i.e., if the test fails to find a job J' with a reservation that can be used to by job J, then there exist no augmenting paths.

4. CONFIGURATION AND MIDDLEWARE INTEGRATION

This section discusses how to configure the job submission service, with focus on the middleware integration points. Also, the configuration of the WS-Agreement services is briefly covered. We illustrate the middleware integration by describing the custom components required for using the job submission service with three Grid middlewares, GT4, LCG2 and ARC.

Integration of a Grid middleware in the job submission service is handled through the service configuration. This configuration determines which plugin(s) to use for each middleware integration point. Note that the job submission service can have multiple plugins for the same task, enabling it to simultaneously communicate with resources running different Grid middlewares. Using the chain-of-responsibility design pattern, the plugins are tried, one after another, until one plugin succeeds in performing the current task. The configuration file specifies which plugins to use in the InformationFinder and the Dispatcher. This file also specifies connection timeouts, the number of threads to use in the thread pools, and default index services. The client is configured in a separate file, allowing multiple users to share a job submission service while customizing their personal clients. The client configuration file determines which job description translator plugins to use, and also specifies some settings related to client-side file staging.

The configuration of the WS-Agreement services determines which *DecisionMaker(s)* to use. A DecisionMaker is a plugin that grants (or denies) agreement offers of a certain agreement type. A DecisionMaker uses two plugin scripts to perform the actions required to create and destroy agreements. For the advance reservation scenario, these plugin scripts interacts with the local scheduler in order to request and release reservations.

4.1. Integration with Globus Toolkit 4

The GT4 middleware does, among other things, provide Web Service interfaces for fundamental Grid tasks such as job submission (WS-GRAM), monitoring and discovery (WS-MDS), and, data transfer (RFT) [20]. The job submission client plugin for GT4 job description translation is straightforward. The only issue encountered is that job input and output files are specified using the same attribute in JSDL, whereas the GT4 job description format uses two different attributes for this.

There is no fixed information hierarchy in GT4, any type of information can be propagated between a pair of WS-MDS *index services*. A basic setup (also used in our test environment) is to have one index service per cluster, publishing information about the cluster, and one additional index service that aggregates information from the other index services. Thus, the typical GT4 information hierarchy does not really fit the infrastructure envisioned by the job submission service, with one or more index servers storing (only) contact information to clusters. However, by using an XPath query in the GT4 ResourceFinder plugin, it is possible to limit the information returned from the top level

GT4 index service to only cluster contact information. This list of cluster addresses is sent to the GT4 InformationFetcher plugin, that (also using XPath) queries each resource in more detail. Both these plugins communicate with the Grid resource using Web service calls. The InformationConverter plugin for GT4 is trivial as GT4 resource information is described using the GLUE format.

The GT4 Dispatcher plugin converts the job description from JSDL to the job description format used in GT4 and next sends the job request to the GT4 WS-GRAM service running on the resource by invoking the job request operation of the service. This procedure becomes more complicated if the Grid resource is to stage (non-local) job input files, in which case the user's credentials must be delegated from the GT4 Dispatcher plugin to the resource.

The WS-Agreement services themself require no middleware-specific configuration. However, job requests that claim a reservation must be authorized, i.e., it must be established that the user requesting the job is the same as the one that previously created the reservation, e.g., by comparing the distinguish names of the proxies used for the two tasks. In GT4, an Axis request flow chain that intercepts the job request performs this test. Due to current limitations in WS-GRAM, the Globus built-in authorization framework could not be used for the task.

4.2. Integration with LCG2

The LCG2 middleware is based on Globus Toolkit 2 (GT2), and uses additional components, e.g., for resource brokering (Condor-G) and top level index services (BDII).

The integration of our job submission service with LCG2 includes a client plugin for translating the Condor-style *classads* used as job description language in LCG2 to JSDL. This translation is rather tedious as classads use a format where any value-pair expression is a valid part of a job description. Furthermore, classads allow a user not only to specify resource requirements (hardware etc.), but also to express a resource ranking function, i.e., an arithmetic expression over the attributes specified in the job description or gathered from the Grid resource. If a user specifies such a resource ranking expression in a classad, the expression is ignored as the job submission service uses other resource ranking algorithms (described in Section 3.1).

Similarly to GT4, LCG2 uses a centralized information structure where each resource registers all available information in a BDII server. This server stores detailed information about the (thousands of) resources available in the LCG2 Grid. In order to avoid overloading the BDII server, the LCG2 ResourceFinder plugin does, similar to its GT4 counterpart, query only for resource contact information. The LCG2 InformationFetcher contacts the GT2 GRIS running on each resource and sends a query asking for detailed resource information. Both these plugins use LDAP to communicate with the LCG2 resources. Although LCG2 uses the GLUE information model to describe resources, the retrieved information must be translated as it is represented in LDAP-specific data structures, which does not correspond to the GLUE XML schema used by the job submission service. This translation is however straightforward as no mapping of attribute names or similar has to be performed.

LCG2 uses the GT2 GRAM on its *Computing Elements*. Upon submission to an LCG2 resource, the LCG2 Dispatcher plugin translates the JSDL document to the GT2 RSL format and sends it to the GRAM Gatekeeper. There is currently no support in the job submission service for advance reservations (or coallocation) of LCG2 resources.

4.3. Integration with ARC

The ARC middleware is based on GT2, but replaces some GT2 components, including the GRAM which is replaced by a *GridFTP server* that accepts job requests and a *Grid Manager* which manages the accepted Grid jobs through their execution.

The information system in ARC is based on GT2, and uses a hierarchy where a GIIS server keeps a list of available GRIS (and GIIS) servers, which periodically announce themselves to the GIIS. Another configuration, used in some ARC installations, is to aggregate all GRIS information in the GIIS. The ARC ResourceFinder and InformationFetcher plugins use LDAP to retrieve lists of available resources and detailed resource information, from the GIIS and GRIS respectively. The resource information is described using an ARC-specific schema, and must hence be translated to the GLUE format by the ARC InformationConverter plugin. The ARC and GLUE information models are not fully compatible, but most attributes relevant to resource brokering, e.g., hardware configuration and current load, can be translated between the two models.

The ARC Dispatcher plugins converts the JSDL job description to the GT2 RSL-style format (called xRSL) used by ARC and sends the resulting job description to the ARC GridFTP server, i.e., the Dispatcher plugin is a GridFTP client.

Authorization of job requests claiming a reservation is done similarly as in GT4 (by comparing distinguished names). A plugin structure in the ARC Grid Manager enables interception of the job request at a few predefined steps. One such plugin performs the reservation authorization before the job is submitted to the local scheduler.

5. PERFORMANCE EVALUATION

There are several factors that affect the performance of the job submission service, including the Grid middleware deployed on the resources, the number of resources, the local scheduler used by the resources, and whether advance reservations are used or not. In order to evaluate this, the performance analysis include measuring, for varying load, (1) the response time, i.e., the time required for a client to submit a job, and (2) the service throughput, i.e., the number of jobs submitted per minute by the service. Performance results are presented for tests with resources running both GT4 and ARC. In addition, some observations on the performance of the coallocation algorithm are made.

5.1. Background and test setup

The performance of the job submission service is evaluated using the DiPerF framework [13]. DiPerF can be used to test various aspects of service performance, including throughput, load and response time. A DiPerF test environment consists of one *controller* host, coordinating and collecting output from a set of *testers* (clients). All testers send requests to the service to be tested and report the measured response times back to the controller. Each tester runs for a fix period of time, and invokes the service as many times it can (submits as many jobs as possible) during the test period.

The response time measured in the client includes the time to establish secure connections to the job submission service, to delegate the user's credential to the service, and to submit the job. On the service side, the time required for the broker's job processing and time required to interact with index servers and resources are also included in the response time. The throughput is computed in DiPerF by counting the number of requests served during each minute. This calculation is done off-line when all testers have finished executing.

GT4 clients developed using Java have an initial overhead in the order of seconds due to the large number of libraries loaded upon start up, affecting the performance of the first job submitted by each client. As a result, a simple request-response Web service call takes approximately five seconds using a Java client (subsequent calls from the same client are however much faster), although a similar call takes less than half a second for a corresponding C client. To overcome this obstacle, a basic C job submission client is used for performance testing.

The performance measurements has been performed in a testing environment with four small clusters, each equipped with a 2 GHz AMD Opteron CPU and 2 GB memory, Ubuntu Linux 2.6, Maui 3.2.6 and Torque 2.1.2. Each cluster is configured with 8 (virtual) backend nodes used by the Torque batch system. The clusters use GT 4.0.3, ARC 0.5.56, or both of these as Grid middleware. For both middleware configurations, one of the clusters also serve as index server for itself and the other clusters. To enable advance reservations, the WS-Agreement services are deployed on each of the four clusters.

Two sets of campus computer laboratories were used as the DiPerF testers (clients), all computers running Debian Linux 3.1 with kernel 2.6. Sixteen of these computers are equipped with AMD Athlon 64 2000 MHz dual core CPUs and 2 GB memory, the other sixteen have 2.8 GHz Pentium 4 CPUs with 1 GB memory each. The job submission service itself was deployed on a computer with a 2 GHz AMD Opteron CPU and 2 GB memory, running Debian Linux 3.1 with the 2.6 kernel. All machines in the test environment are interconnected with a 100 Mbit/s network.

The job submission service was configured with a timeout of 15 seconds for all interactions with the information systems of the resources. The Grid middlewares were configured to generate updated resource information every 60 seconds and the information gathered by the broker was hence cached for this amount of time. Queries about resource information and negotiations of advance reservations were both performed using four parallel threads.

The use of a relatively small but controlled environment for tests, have the advantage that we have full control over the load on the clusters. Hence, the performance of the job submission service can be significantly more accurately analyzed than it could have been on a large production Grid (e.g, as performed in [16]).

5.2. Performance results

Tests have been performed with the number of resources varying between one and four, and the number of clients being $\{3, 5, 7, 10, 15\}$. Each test starts with one client, and then another client is added every 30th second until the selected number of clients is reached. Each client executes for 15 minutes and submits trivial jobs that each outputs a single message and exit. Hence, also tests with large number of clients include time periods where smaller number of clients are used. The reason for this strategy is to better identify the relation between service load and throughput or response time.

In the following presentation, the performance results are grouped by the Grid middleware used, i.e., GT4 och ARC. For each middleware, results are presented for tests using the Torque “PBS” scheduler and POSIX “Fork”.

Our results show that the performance varies very little with the number of Grid resources used. Resource discovery takes longer when more resources are used, but the load distribution of the jobs across more machines does, on the other hand, give faster response time in the dispatch step. These two factors seem to compensate each other rather well for one to four resources. Because of this, we here only present results obtained using four resources. ¿From our tests, we also find it sufficient to present results for tests using 3, 7, and 15 clients.

For tests using the GT4 middleware, Figure 5 shows how the service throughput (lines marked with "$\times$") and response time for the job requests (lines without "$\times$") vary during the tests. The three figures present, from top to bottom, the results obtained using 3, 7, and 15 clients. Solid and dashed lines are used to represent results obtained using Fork and PBS, respectively. Notably, the scale for the response times is on the left-hand side of the figures and the scale for the throughput is found on the right-hand side.

In the results obtained using three clients (the topmost diagram in Figure 5), we do not see any particular trend in the results as the number of clients are increased from one to three (recall that in each test, a new client is started every 30 seconds), which indicates that the service has no problem at all to handle this load. Notably, the response time for individual jobs is as low as down to under one second at best.

As the number of clients increases to seven in the middle graph, we observe that both the response time and the throughput increase as more clients are being started, until it reaches a maximum and then starts to decrease as the clients finish executing. The increase in response time indicate that some bottleneck has been found. As the throughput still increases, our interpretation is that the increase in response time is due to increased waiting for resources to respond, and not due to too high load for the job submission service itself.

We remark, that this is the test for which we see the highest throughput for GT4, with a maximum of just over 250 jobs per minute for Fork and only slightly lower with PBS. Response times for Fork vary between one and two seconds, whereas they fluctuate up to three seconds for PBS. In comparison to the results for three clients, we see that the throughput doubles for Fork, whereas the increase in throughput for PBS is somewhat lower. When further increasing the load to 15 clients, we see that the throughput from the tests with seven clients is maintained also for heavy load, even though we do not reach the same peak result.

In summary, the tests with GT4 resources show that the job submission service is capable of handling throughput up to just over 250 jobs per minute and to give individual job response times down to under one second.

For tests using the ARC middleware, Figure 6 shows the performance using four resources and three, seven and fifteen clients, respectively. Here, the throughput increases from 60-70 jobs/minute with three clients (the top diagram in Figure 6) to approximately 170 jobs per minute with seven clients (the middle plot in Figure 6), while keeping response times between two and three seconds per submitted job.

When further increasing the load to 15 clients, we see in the bottom diagram in Figure 6 a slight increase in throughput, to approximately 200 jobs per minute, whereas the response time increases as well, to approximately four seconds. This suggests that the maximum throughput is around 200 jobs per minute.

Notably, in our tests PBS and Fork perform reasonably equal for both middlewares and for all combinations of different numbers of clients and resources, even though we see slightly more

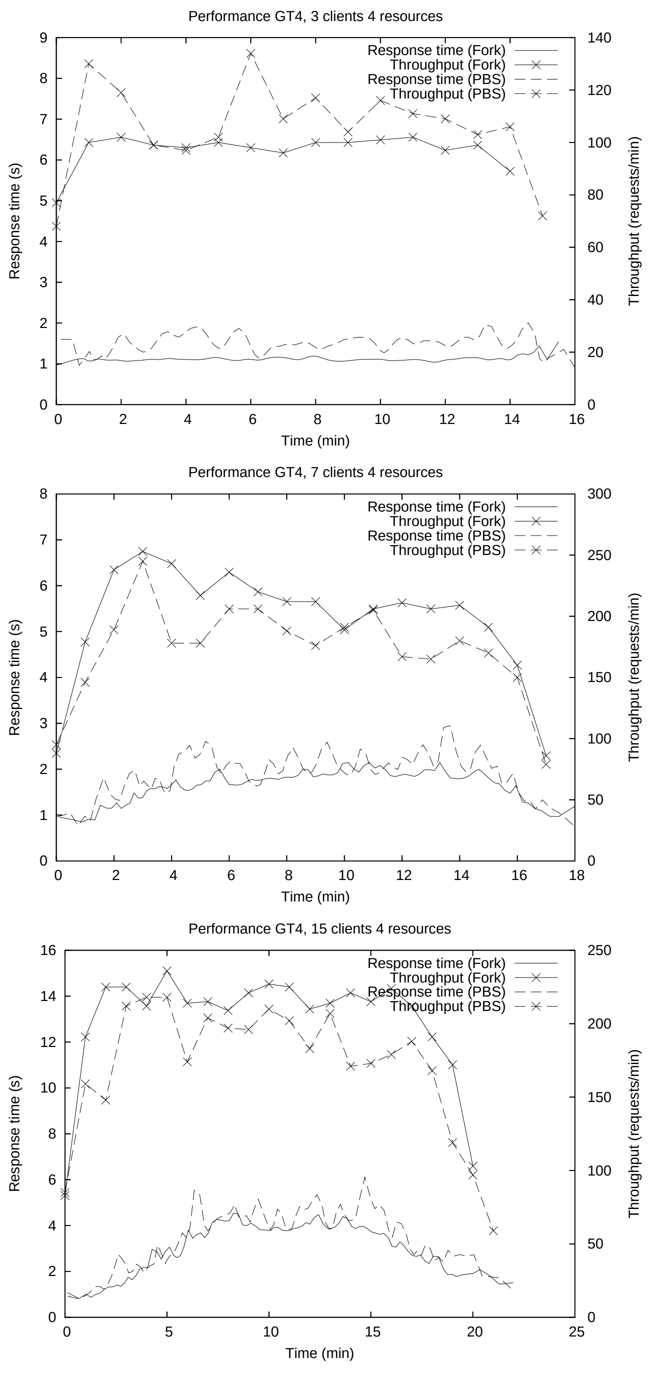


Figure 5. Performance results for GT4 using 4 resources.

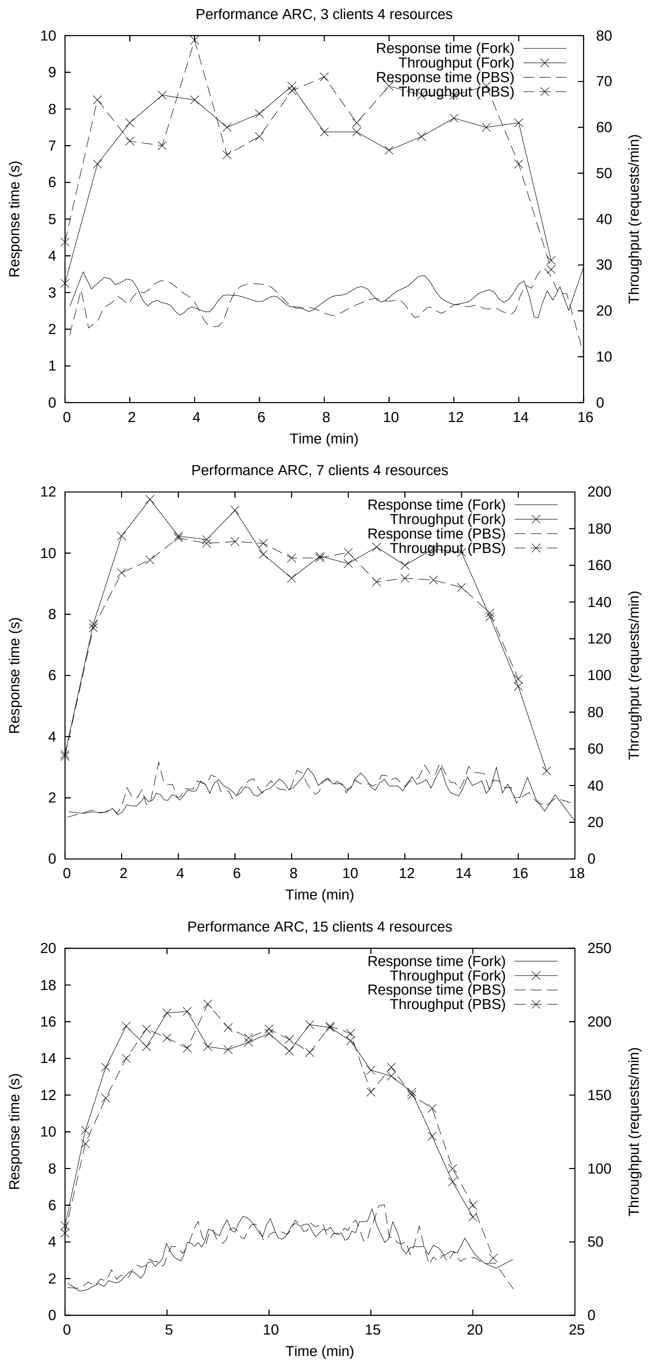


Figure 6. Performance results for ARC using 4 resources.

fluctuating response times using PBS than with Fork. However, if tests are done with jobs requiring substantial computational capacity, the performance obtained using Fork will substantially decrease. For PBS, we expect the results to be similar also for more demanding jobs, if the clusters make use of real (and not virtual) back-end nodes.

The slightly more fluctuating response times obtained using PBS are explained by the fact that the information systems used by ARC and GT4 both perform extensive parsing of PBS log files to determine the current load on the resource. During significant load, this may occasionally lead to slow response times for resource information queries. This does in term result in slower response times for jobs for which the broker can not use cached resource information.

During the period of constant load (while all clients execute), we see a slight decrease in throughput for both ARC and GT4. This decrease, which is most clearly visible in the tests using 7 and 15 clients, can partly be explained by limitations in handling large number of delegated credentials in the GT4 delegation service, a topic further investigated in [24]. The delegation service is for each job invoked by the job submission client to delegate the user’s credential to the job submission service.

5.2.1. Advance reservations

In order to evaluate the performance impact of advance reservations on the job submission service, tests with jobs requesting reservations are compared to the corresponding tests performed without use of reservations. The performance of the job submission service for jobs using reservations are, of course, expected to be lower. A job submitted requesting an advance reservation requires two additional round trips (get agreement template, create agreement) during brokering and one more round trip during job dispatch (confirm temporary reservation). When each job submission request takes longer to serve, fewer jobs can utilize cached resource information before the cache expires, which further decreases performance.

As previous research have demonstrated [19, 52], the usage of advance reservation inflict a performance penalty, and does typically reduce batch system utilization dramatically already when only 20 percent of all jobs use advance reservations. Our resource brokering algorithms described in Section 3.1, are able to create reservations for all resources of interest (or a subset thereof), and upon job submission release all reservations but the one for the selected resource. However, as long as the batch systems do not provide a lightweight reservation mechanism, we think that this feature should be used only when needed.

In order to investigate the performance impact of the advance reservation mechanism, we consider a scenario where exactly one reservation is created for each submitted job.

The performance results for GT4 with reservations (dashed lines) is compared to corresponding results without reservations (solid lines) in Figure 7. We note that the throughput (marked $\times$) with reservations is about 40 submitted jobs per minute for all three tests. In these tests, the response time increases from about five seconds (three clients), to ten seconds (seven clients), and finally to around twenty seconds (fifteen clients). In comparison, for jobs submitted without reservations, throughput increases from around 100 jobs per minute (three clients), to around 210 jobs per minute (seven clients), and finally increases a bit more to around 220 jobs per minute when fifteen clients are used. The response times for these jobs are around two seconds (both three and seven clients) and three seconds for fifteen clients.

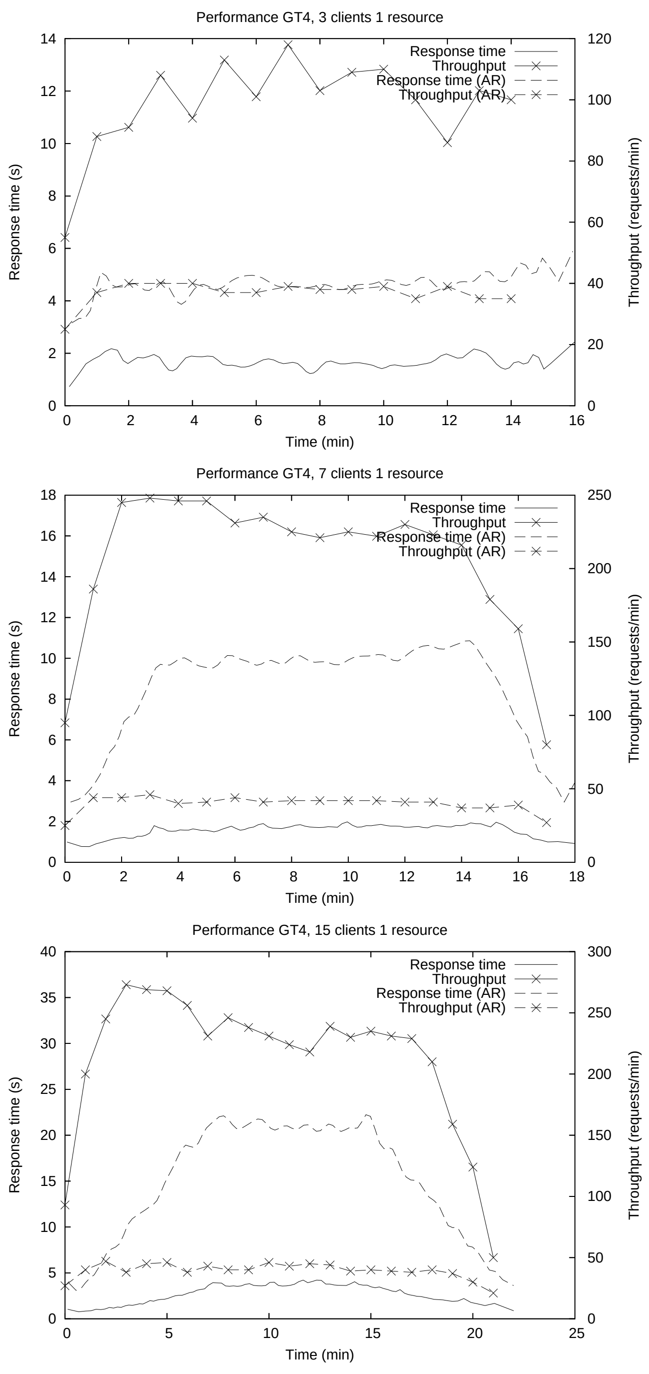


Figure 7. Performance results for advance reservations using GT4 and 1 resource.

The performance results for tests of jobs with advance reservations submitted to ARC are very similar to the corresponding tests using GT4, so therefor we do not include the corresponding graphs for these tests. For jobs submitted with reservations to ARC, the throughput is around 30 jobs per minute for three clients, and 40 jobs per minute for seven and for fifteen clients. The response time varies from around six seconds for three clients, to ten seconds for seven clients and twenty seconds for fifteen clients. In the reference tests where no jobs used reservations, the throughput is around 55 jobs per minute for three clients, 140 jobs per minute with seven clients and 160 jobs per minute with fifteen clients. In these tests, the response time is around 3 seconds for both three and seven clients, and around 4 seconds for fifteen clients.

From the results for GT4 and ARC, we conclude that for jobs submitted with advance reservations, the job submission service and the WSAG services can serve around 40 submitted jobs per minute and that the average response time for these jobs is (at best) below five seconds.

5.2.2. Coallocation

The performance of the coallocation algorithm depends on the current status of the batch system queues of the considered Grid resources, which directly affects how many iterations of the main loop in the coallocation algorithm that have to be executed.

The job request validation, resource discovery and information retrieval performed by the Coallocator are similar to the initial steps executed during submission of ordinary jobs and takes approximately 1-3 seconds to execute if new Grid resource information must be retrieved and a less than a tenth of that time if cached resource information is available. We note that for a coallocated job request with three jobs that each can use (the same) four resources, each iteration of the coallocation algorithm, including creation of new reservations and modification of existing ones, takes around 3.5 seconds to execute. To find an augmenting path of length three and to update the reservations along the path takes around two seconds, almost entirely spent in the reservation update procedure. Moreover, it follows from the design of the algorithm that the execution time of the path updating algorithm increases linearly with the length of the path.

6. RELATED WORK

We have identified a number of contributions related to our work on Grid resource brokering, including performance prediction for Grid jobs, the usage of advance reservations in Grids and resource coallocation. In the following, we make a brief review of these.

6.1. General resource brokering

The compositionable ICENI Grid scheduling architecture is presented in [62], together with a performance comparison between four Grid scheduling algorithms; random, simulated annealing, best of n random, and a game theoretic approach.

The eNANOS Grid resource broker [47] supports submission and monitoring of Grid jobs. Features include usage of the GLUE information model [3] and a mechanism where users can control the resource selection by weighting the importance of attributes such as CPU speed and RAM size.

There are a number of projects that investigate market-based resource brokering approaches. These approaches may typically have a starting point in bartering agreements, in pre-allocations of artificial "Grid credits" or be based on real economical compensation. In such a Grid marketplace, resources can be sold either at fixed or dynamic prices, e.g., in a strive for a supply and demand equilibrium [60]. Claimed advantages of the economic scheduling paradigm include load balancing and increased resource utilization, both a result of good balance between supply and demand for resources [60]. Examples of work on economic brokering include [44, 9, 18, 12]. An alternative to market-based economies is the Grid-wide fairshare scheduling approach [15], which can be viewed as a planned economy.

6.2. Performance prediction

One method for determining which resource to submit a computational job to is to predict the performance of the job if executing on each resource of interest. These predictions can include the job start time as well as the job execution time. Techniques for such predictions include (i) applying statistical models to previous executions [51, 1, 54, 33, 35] and (ii), heuristics based on job and resource characteristics [59, 29, 36].

In our previous work [17], we use a hybrid approach. The performance characteristics of an application is classified using computer benchmarks relevant for the application, as in method (ii). When predicting the performance for a Grid resource, the benchmark results for this machine is compared with those of a reference machine where the application has executed previously. This comparison with earlier execution of the application reuses techniques from method (i).

6.3. Interoperability efforts

There are several resource brokering projects which target resources running different Grid middlewares, e.g., Gridbus [57], which can schedule jobs on resources running, e.g., Globus [25], Unicore [55] and Condor [37]. The GridWay project [31] targets resources running both protocol oriented (GT2) and service-based versions (GT4) of the Globus toolkit as well as LCG [10]. One difference between our contribution and these projects is that we target the use of any Grid middleware both on the resource and client side by allowing clients to express their jobs in the native job description language of their middleware, whereas the job description language of Gridbus and GridWay is fixed on the client side.

The UniGrids project [56] specially targets interoperability between the Globus [25] and Unicore [55] middlewares. The Grid Interoperability Now (GIN) [26] initiative focuses on establishing islands of interoperation between existing Grid resources, and growing those islands to achieve an increasing set of interoperable Grid middlewares.

There are a few projects that have adopted JSDL to describe jobs, e.g., [27] and [42].

6.4. Advance Reservations

Several projects conclude that an advance reservation feature is required to meet QoS guarantees in a Grid environment [22, 50, 30]. However, the support for reservations in the underlying infrastructure is currently limited [38]. The performance penalty imposed by the usage of advance reservations (typically decreased resource utilization) has been studied in [52, 53]. The work in [19] investigates how performance improvements can be can be achieved by allowing laxity (flexibility) in advance reservation start times.

Standardization attempts include [48], which defines a protocol for management of advance reservations. The more recent WS-Agreement [4] standard proposal defines a general architecture that enables two parties, the agreement provider and the agreement consumer, to enter an agreement. Although not specifically targeting advance reservations, WS-Agreement can be used to implement these, demonstrated by e.g., [41, 58, 16].

6.5. Coallocation

The work by Czajkowski et.al. [11] describes a library for initiating and controlling coallocation requests and an application library for synchronization. By compiling an application requiring coallocation with the application library, the subjob instances can wait for each other at a barrier prior to commencing execution. This is typically required when setting up an MPI environment distributed across several machines. The work in [11] does not contain any algorithm for the actual coallocation of the subjobs.

The Globus Architecture for Reservation and Allocation (GARA) [22] provides a programming interface to simplify the construction of application-level coallocators. GARA supports both immediate reservations (allocations) and advance reservations. The system furthermore supports several resource types, including networks, computers and storage. GARA focuses on the development of a library for coallocation agents and only outlines one possible coallocation agent [22], targeting the allocation of two computer systems and an interconnection network at a fixed time. The focus of our work is the implementation of a more general coallocation service able to allocate an arbitrary number of computational resources. Our coallocation algorithms also differs from GARA as they allow for a flexible reservation start within a given interval of time.

The authors of the KOALA system [43] propose a mechanism for implementing coallocation that does not use advance reservations. Their approach is to request longer execution times than required by the jobs, and delay the start of the each job until all allocated jobs are ready to start executing.

The work by Matescuu [39] defines an architecture for coallocation based on GT2. The described coallocation algorithm shares some concepts with our algorithm, including the use of a window of acceptable job start times and iterations in which reservations for all job requests are created. Differences include that the algorithm by Matescuu only attempts to reserve resources at a few predefined positions in the start time window, whereas our algorithm uses information included in rejection messages to dynamically determine where in the start time window to retry to create reservations. Our algorithm also tries to modify existing reservations when considering a new start time window. Furthermore, our algorithm uses a mechanism to exchange reservations between jobs in the coallocated job, which can resolve conflicts if more than one job requests the same resource(s).

The coallocation algorithm developed by Wäldrich et al [58] uses the concept of coallocation iterations, and models reservations using the WS-Agreement framework. In each iteration of the algorithm by Wäldrich et al, a list of free time slots is requested from each local scheduler. Then, an off-line matching of the time slots with the coallocation request is performed. If the request can be mapped onto some set of resources, reservations are requested for the selected slots.

Although being similar at first glance, our coallocation algorithm has some fundamental differences from the one described by Wäldrich et al. Our algorithm selects which resources to coallocate incrementally by matching one resource at the time, whereas the algorithm by Wäldrich et al is based on an off-line calculation of which resources to use. We argue in Section 3.3.3 that, due to incomplete information and lack of central control, selection of which resources to use should be performed in an on-line manner. Further differences include that our coallocation algorithm allows a user-specified fluctuation in reservation start times, while the algorithm described in [58] uses a fixed notion of reservation start time. If allowing fluctuations in reservation start times, our algorithm is more likely to succeed in coallocating a suitable set of resources than the algorithm by Wäldrich et al. Our algorithm is also more efficient, as existing reservations may be reused in subsequent iterations.

The work described in [2] reuses the concept of barriers from [11]. In [2], the coallocator architecture consisting of a selection agent, a request agent and a barrier agent. A model for multistage coallocation is developed, where one coallocation service passes a subset of the coallocation request to another coallocation service, thus forming hierarchy of coallocators. The barrier functionality developed in [2] also supports the synchronization of hierarchically coallocated jobs. Our work differs from [2], e.g., by using a flat model where a broker negotiates directly with the resources.

Deadlocks and deadlock prevention techniques in a coallocation context are described by Park et al. [46] whereas other work [8] suggests performance improvements for these deadlock prevention techniques. We however argue that the coallocation algorithm described in this paper does not cause deadlocks. Deadlocks can only occur [32] when the following four conditions hold simultaneously: (i) mutual exclusion, (ii), hold and wait, (iii) no preemption, and (iv), circular wait. Our algorithm modifies (or releases) reservations for resources whenever the algorithm fails to acquire an additional required resource. Condition (ii) does hence not hold and no deadlock can occur.

7. CONCLUSIONS

We have demonstrated how a general Grid job submission service can be designed to enable all-to-all cross-middleware job submission, by leveraging emerging Grid and Web services standards and technology. The architecture's ability to manage different middlewares have been demonstrated by providing plugins for GT4, LCG2, and NorduGrid/ARC. Hence, job and resource requests can be specified in any of these three input formats, and independently, the jobs can be submitted to resources running any of these three middlewares.

A modular design facilitates the customizability of the architecture, e.g., for tuning the resource selection process to a particular set of Grid resources or for a specific resource brokering scenario. The current implementation includes resource selection algorithms that can make use of, but do not depend on, rather sophisticated features for predicting individual job performance on individual resources. It also provides support for advance resource reservations and coallocation of multiple resources.

Even though the design of the job submission service is for decentralized use, i.e., typically to be used by a single user or a small group of users, the performance analysis give at hand that it can handle a quite significant load. In fact, the job submission service itself appears not to be the bottleneck as times waiting for resources becomes dominating. At best, the job submission service is able to give individual job response times below one second and to provide a total throughput of up to over 250 jobs per minute.

Future directions in this work include adaptation of the current architecture and interfaces to follow more recent emerging standards such as the Basic Execution Service [28] and the OGSA Execution Management Services [23]. We also plan to develop a library for job coordination for coallocated jobs, allowing the jobs to coordinate themselves prior to execution at their respective cluster. This is required, e.g., for setting up MPI environments for jobs using cross-cluster communication. This work will build on our experiences from job coallocation and previous work, such as [11].

The current coallocation algorithm reserves jobs for simultaneous job start. The algorithm can be extended to allow arbitrary coordination of the jobs, which would be useful e.g., for workflow purposes where there is a specific order in which jobs are to be executed. The current algorithm would only require minor modifications to allow a per job offset from a simultaneous start time.

8. SOFTWARE AVAILABILITY

The software described in this paper is available at `www.cs.umu.se/research/grid/jss`. This web page contains the job submission service software, installation instructions and a user's guide.

ACKNOWLEDGEMENTS

The authors are grateful to Catalin Dumitrescu, Peter Gardfjäll, Klas Markström, Ioan Raicu, Åke Sandgren and Björn Torkelsson. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

REFERENCES

1. A. Ali, A. Anjum, J. Bunn, R. Cavanaugh, F. van Lingen, R. McClatchey, M. A. Mehmood, H. Newman, C. Steenberg, M. Thomas, and I. Willers. Predicting resource requirements of a job submission. In *Proceedings of the Conference on Computing in High Energy and Nuclear Physics (CHEP 2004), Interlaken, Switzerland*, September 2004.
2. S. Ananad, S. Yoginath, G. von Laszewski, and B. Alunkal. Flow-based Multistage Co-allocation Service. In Braan J d'Auriol, editor, *Proceedings of the International Conference on Communications in Computing*, pages 24–30, Las Vegas, 2003. CSREA Press.
3. S. Andreozzi, S. Burke, L. Field, S. Fisher, B. Kónya, M. Mambelli, J. M. Schopf, M. Viljoen, and A. Wilson. Glue schema specification version 1.2 draft 7. http://glueschema.forge.cnaf.infn.it/uploads/ Spec/GLUEInfoModel_1_2_final.pdf, November 2006.
4. A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (WS-Agreement). https://forge.gridforum.org/sf/docman/do/downloadDocument/projects.graap-wg/docman.root.current_drafts/doc6090, November 2006.
5. A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Negotiation Specification (WS-AgreementNegotiation). https://forge.gridforum.org/sf/go/doc6092?nav=1, November 2006.
6. A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, A. S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) specification, version 1.0. http://www.ogf.org/documents/GFD.56.pdf, November 2006.
7. Apache Web Services Project - Axis. http://ws.apache.org/axis, September 2006.
8. D. Azougagh, J-L. Yu, J-S. Kim, and S-R. Maeng. Resource co-allocation: a complementary technique that enhances performance in grid computing environment. In *Proceedings of the 2005 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 36–42, 2005.

9. R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in Grid computing. *Concurrency Computat. Pract. Exper.*, 14:1507–1542, 2002.
10. J. Knobloch (Chair) and L. Robertson (Project Leader). LHC computing Grid technical design report. Technical report, CERN, 2005. http://lcg.web.cern.ch/LCG/tdr/.
11. K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational Grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.
12. M. Dalheimer, F-J. Pfreundt, and P. Merz. Agent-based grid scheduling with Calana, 2005.
13. C. Dumitrescu, I. Raicu, M. Ripeanu, and I. Foster. Diperf: An automated distributed performance testing framework. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 289–296, Washington, DC, USA, 2004. IEEE Computer Society.
14. M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M. Niinimäki, O. Smirnova, and A. Wäänänen. Advanced resource connector middleware for lightweight computational Grids. *Future Generation Computer Systems*, 27:219–240, 2007.
15. E. Elmroth and P. Gardfjäll. Design and evaluation of a decentralized system for Grid-wide fairshare scheduling. In H. Stockinger, R. Buyya, and R. Perrott, editors, *First International Conference on e-Science and Grid Computing*, pages 221–229. IEEE CS Press, 2005.
16. E. Elmroth and J. Tordsson. An interoperable, standards-based Grid resource broker and job submission service. In H. Stockinger, R. Buyya, and R. Perrott, editors, *First International Conference on e-Science and Grid Computing*, pages 212–220. IEEE CS Press, 2005.
17. E. Elmroth and J. Tordsson. A Grid resource broker supporting advance reservations and benchmark-based resource selection. In J. Dongarra, K. Madsen, and J. Waśniewski, editors, *PARA04, LNCS 3732*, pages 1061–1070, 2006.
18. C. Ernemann, V. Hamscher, and R. Yahyapour. Economic scheduling in Grid computing. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 2537*, pages 128–152, 2002.
19. U. Farooq, S. Majumdar, and E. W. Parsons. Impact of laxity on scheduling with advance reservations in Grids. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 319–324, Washington, DC, USA, 2005. IEEE Computer Society.
20. I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, LNCS 3779*, pages 2–13. Springer-Verlag, 2005.
21. I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modeling stateful resources with Web services. http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf, November 2006.
22. I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *7th International Workshop on Quality of Service*, pages 27–36. IEEE, 1999.
23. I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, version 1.5. http://www.ogf.org/documents/GFD.80.pdf, November 2006.
24. P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, and T. Sandholm. Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS). *Submitted for Journal Publication*, October 2006.
25. Globus. http://www.globus.org. September 2006.
26. Grid Interoperability Now. http://wiki.nesc.ac.uk/read/gin-jobs. September 2006.
27. GridSAM. http://gridsam.sourceforge.net. September 2006.
28. A. Grimshaw, S. Newhouse, D. Pulsipher, and M. Morgan. OGSA Basic Execution Service version 1.0. https://forge.gridforum.org/sf/go/doc13793, November 2006.
29. R. Gruber, V. Keller, P. Kuonen, M-C. Sawley, B. Schaeli, A. Tolou, M. Torruella, and T-M. Tran. Towards an intelligent grid scheduling system. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 751–757. Springer Verlag, 2005.
30. M. H. Haji, I. Gourlay, K. Djemame, and P. M. Dew. A snap-based community resource broker using a three-phase commit protocol: A performance study. *The Computer Journal*, 48(3):333–346, 2005.
31. E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution on grids. *Software - Practice and Experience*, 34:631–651, 2004.
32. E. G. Coffman Jr, M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):68–78, 1971.
33. N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *In Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1999.
34. K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software - Practice and Experience*, 15(32):135–164, 2002.
35. H. Li, J. Chen, Y. Tao, D. Gro, and L. Wolters. Improving a local learning technique for queue wait time predictions. *ccgrid*, 0:335–342, 2006.

36. H. Li, D. Groep, and L. Wolters. Efficient response time predictions by exploiting application and resource state similarities. In *6TH INTERNATIONAL WORKSHOP ON GRID COMPUTING (GRID 2005)*, pages 234–241, 2005.
37. M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.
38. J. MacLaren. Advance reservations state of the art, 2003. http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html, September 2006.
39. G. Mateescu. Quality of service on the Grid via metascheduling with resource co-scheduling and co-reservation. *Int. J. High Perf. Comput. Appl.*, 17(3):209–218, Fall 2003.
40. Maui Cluster Scheduler. http://www.clusterresources.com/products/maui/, September 2006.
41. A. S. McGough, A. Afzal, J. Darlington, N. Furmento, A. Mayer, and L. Young. Making the grid predictable through reservations and performance modelling. *The Computer Journal*, 48(3):358–368, 2005.
42. K. Miura. Overview of japanese science Grid project NAREGI. *Progress in Informatics*, 1(3):67–75, 2006.
43. H. H. Mohamed and D. H. J. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. In *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID2005)*, pages 784–791. IEEE Computer Society, 2005.
44. R. Moreno and A. B. Alonso-Conde. Job scheduling and resource management techniques in economic Grid environments. In *Lecture Notes in Computer Science*, volume 2970/2004, pages 25–32. Springer Berlin / Heidelberg, 2004.
45. Open Grid Forum. www.ogf.org. October 2006.
46. J. Park. A scalable protocol for deadlock and livelock free co-allocation of resources in internet computing. In *Proceedings of the 2003 Symposium on Applications and the Internet*, pages 66–73. IEEE Computer Society, 2003.
47. I. Rodero, J. Corbalán, R. M. Badia, and J. Labarta. eNANOS Grid Resource Broker. In P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005, LNCS 3470*, pages 111–121, 2005.
48. A. Roy and V. Sander. Advance reservation API. http://www.gridforum.org/documents/GFD.5.pdf, May 2006.
49. J. Schopf. Ten actions when Grid scheduling. In J. Nabrzyski, J. Schopf, and J. Węglarz, editors, *Grid Resource Management State of the art and future trends*, chapter 2. Kluwer Academic Publishers, 2004.
50. M. Siddiqui, A. Villazon, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized qos. In *the 2006 ACM/IEEE Conference on Supercomputing SC—06*, 2006. To appear.
51. W. Smith, I. Foster, and V. Taylor. Using run-time predictions to estimate queue wait times and improve scheduler performance. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 202–219, Berlin, 1999. Springer-Verlag.
52. W. Smith, I. Foster, and V. Taylor. Scheduling with advance reservations. In *14th International Parallel and Distributed Processing Symposium*, pages 127–132, Washington - Brussels - Tokyo, 2000. IEEE.
53. Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing: IPDPS 2000 Workshop, JSSPP 2000, LNCS 1911*, pages 137–153. Springer Berlin / Heidelberg, 2000.
54. D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling user runtime estimates. *The 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS 3834*, pages 1–35, 2005.
55. Unicore. http://www.unicore.org. September 2006.
56. UniGrids. www.unigrids.org. September 2006.
57. S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling e-science applications on global data grids. *Concurrency Computat. Pract. Exper.*, 18(6):685–699, May 2006.
58. O. Wäldrich, P. Wieder, and W. Ziegler. A meta-scheduling service for co-allocating arbitrary types of resources. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 782–791. Springer Verlag, 2005.
59. J. Wang, L-Y. Zhang, and Y-B. Han. Client-centric adaptive scheduling for service-oriented applications. *J. Comput. Sci. and Technol.*, 21(4):537–546, 2006.
60. R. Wolski, J. Brevik, J. S. Plank, and T. Bryan. Grid Resource Allocation and Control Using Computational Economies. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making The Global Infrastructure a Reality*, chapter 32. John Wiley & Sons, 2003.
61. R. Yahyapour. Considerations for resource brokerage and scheduling in grids. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, pages 627–634, 2004.
62. L. Young, S. McGough, S. Newhouse, and J. Darlington. Scheduling architecture and algorithms within the ICENI Grid middleware. In Simon Cox, editor, *Proceedings of the UK e-Science All Hands Meeting*, pages 5 – 12, 2003.