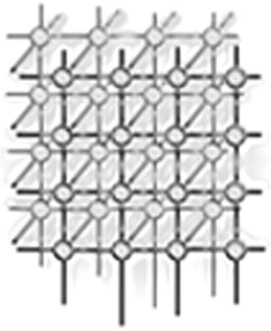
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **00**:1–38 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

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 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 and NorduGrid/ARC. 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 of a particular Grid environment and job submission scenario. By default, the service implements a decentralized brokering policy, striving to optimize the performance for the individual user by minimizing the response time for each job submitted. 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 coordinated use. The performance of the system is analyzed with focus on overall service throughput (up to over 250 jobs per minute) and individual job submission 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

[†]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

Received 14 December 2006
Revised 20 October 2008
Accepted 19 January 2009

1. INTRODUCTION

The emergence of Grid infrastructures facilitates interoperability between heterogeneous resources. Following this development, it is somewhat contradictory that a new level of portability problems has 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 designing and building general and standards-based Grid software.

We argue that the conditions for developing portable Grid applications can be drastically improved 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. 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 describe resources, jobs, requirements, agreements, etc, and for the implementation of the service itself.

The service is also 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) [30] and NorduGrid/ARC [15]. The service itself is designed in compliance with the Web Services Resource Framework (WSRF) [25] and its implementation is based on GT4 Java WS Core.

The architecture of the service includes a set of general components. To emphasize separation of concerns, each component is designed to perform one specific task in the job submission process. 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 plugins at well-defined integration points.

The flexible architecture enables partial or complete replacement of the resource selection algorithms with custom implementations. By default, the service uses a decentralized brokering policy, working on behalf of the user [19, 41]. The existing algorithms strive to optimize the performance for an individual user by minimizing the response time for each job submitted. Resource selection is based on resource information as opposed to resource control, and 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 (QoS), the broker also includes features for performing advance resource reservations and 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 [19]. With the development of the second generation tools, principles of SOAs were adapted. Early work on this WSRF-based job submission tool is presented in [18]. 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.

In summary, our contributions are the following:

- A demonstration of how standard formats and interfaces can be used to support interoperability in terms of all-to-all cross middleware job submission, without restricting functionality to the lowest common denominator. A thorough analysis shows that this can be done with far better performance than required in the envisioned usage scenarios.
- A flexible and portable architecture that allows both customization and replacement of arbitrary components for well-defined subtasks in the the job submission process.
- Resource selection algorithms that can utilize, but do not depend on, sophisticated mechanisms for predicting job and resource performance.
- A standards-based advance reservations framework and its applications in supporting end user QoS.
- Advances to the state-of-the-art in Grid resource coallocation, including the design, implementation, and analysis of an algorithm for arbitrarily coordinated allocations of resources.

The outline of the rest of the paper is organized as follows. The overall system architecture is presented in Section 2. The resource brokering algorithms used in this implementation are described 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 and NorduGrid/ARC. The performance of the system is analyzed in Section 5, followed by a presentation of related work in Section 6. Section 7 contains some concluding remarks, including a discussion of our scientific contributions. Information about how to retrieve the presented software 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 a set of components that each performs a well-defined task 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 resource brokering component.

The service is implemented using the GT4 (Java WS Core) Web service development framework [24]. This framework combines WSRF functionality with the Axis Web service engine [8]. As the service itself is made independent of any particular middleware, all middleware-specific issues are

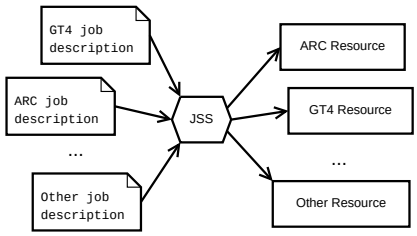


Figure 1. Logical view of interoperability in the job submission service.

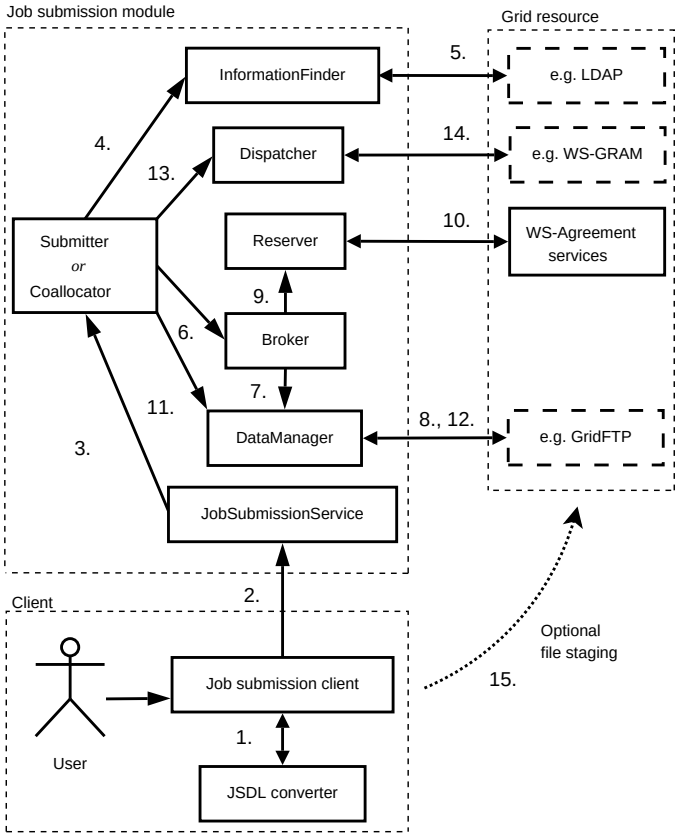
handled by a few, well-defined, plugins. Currently such plugins are available for the GT4 and ARC middlewares. A typical set of middleware plugins constitutes less than ten percent of the general code. Descriptions of the middleware-specific components, including a short discussion about their differences, are found in Section 4.

The service supports job submission to and from any middleware with an X.509 certificate based security framework for which plugins are implemented. This also includes cross-middleware job submission, as illustrated in Figure 1. The figure shows how job requests in the respective job description languages of GT4, ARC or any other supported middleware, are sent to the job submission service (denoted JSS in the figure), which can dispatch the job to a resource that runs any (supported) middleware. This proxy [28] pattern achieves interoperability in the sense of end user transparency, which is in harmony with the ISO definition of interoperability [23]. In contrast to alternative approaches [31, 54], that are based on interoperation through resource side middleware extensions, our solution is non-intrusive as it requires no modifications to the Grid middleware of the resources. However, with our solution, end users must change their job submission software (but can still reuse existing job descriptions), whereas the alternative approaches allow the continued use of middleware-native job submission tools.

In addition to the main job submission and resource brokering service, the framework includes user clients, and an optional advance reservation component that can be installed on the Grid resources for improved QoS. All components are briefly described in sections 2.1–2.3 and their interactions are illustrated in Figure 2.

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) [7], a standardized job description format proposed by the Open Grid Forum (OGF) [55]. Users can of course also specify their job requests directly in JSDL.



1. *Job description translation.*
2. Service invocation.
3. Selection of appropriate submission coordinator.
4. Resource discovery and information retrieval.
5. Information system queries and format conversions.
6. Resource brokering.
7. File transfer time estimation.
8. Job data queries.
9. *Advance reservation request.*
10. *Agreement negotiation.*
11. Input file staging.
12. Data transfers.
13. Job submission, including job description translation.
14. Job request.
15. *Client side file staging.*

Figure 2. An architecture overview that shows components, hosts, and information flow. The dashed lines denote different hosts. The boxes show the components, the solid ones are part of the job submission service, whereas the dashed ones illustrate other services it interacts with. A chronological outline of the job submission processes is shown to the right. In this outline, italic font specifies optional tasks.

The job description languages used in various Grid middlewares are very similar in terms of job configuration, e.g., executable to run, arguments, input and output files, execution environment etc., and resource requirements, e.g., number of requested CPUs, required disk space, architecture, available disk, memory, etc. These attributes are straight-forward to translate. However, not all attributes can be translated into JSDL. Each Grid job description language typically contains a few attributes custom to a specific Grid middleware or job submission tool, e.g., instructions to a specific Grid resource broker. Such attributes are not translated in the job submission service as it does not implement all the different scheduling policies of existing Grid resource brokers. Furthermore, attributes specific for one Grid middleware would not be useful when submitting the job to a resource that runs some other middleware. An alternative approach, taken by Kertész et al. [39], is to add all custom attributes as

JSDL extensions. This is a reasonable approach for Kertész et al. as their resource broker forwards job requests to middleware-specific resource brokers, whereas the service described in this paper communicates directly with the resources. However, users of the job submission service can specify custom requirements for their jobs, but through the optional job preferences document described in Section 2.4, instead of through middleware-specific mechanisms. A few finer details about the the translations to JSDL from the job description languages of GT4 and ARC are discussed in sections 4.1 and 4.2, respectively.

The job submission clients can be configured to transfer job input files stored locally at the client host. This file staging is required if the local files neither can be accessed directly by the job submission service nor by the Grid resource.

2.2. Job submission service

The clients send their JSDL job requests to the *JobSubmissionService* that exposes a Web service interface to the functionality offered by the broker, namely submission of a single job or coallocation of a set of jobs. As part of this invocation, the clients delegate user security credentials (i.e., proxy certificates) to the JobSubmissionService, for later use in interaction with resources. The service forwards incoming requests either to the *Submitter* or the *Coallocator*, depending on the type of the request. For each successfully submitted (or coallocated) job, the JobSubmissionService creates a WS-Resource, with information about the job exposed as WS-ResourceProperties. This mechanism for storing state information in Web services is specified by the WSRF [25].

The Submitter (or the Coallocator) coordinates the job submission process, 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 allocate multiple resources for coordinated use. The algorithms used by these two components are presented in sections 3.1 and 3.2, respectively.

Resource discovery 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) [4]. The GLUE format defines an information model for describing computational and storage resources in a Grid. To improve the performance of the InformationFinder, threadpools are used for all of these three tasks. For additional performance improvements, the retrieved resource information is cached for a short period of time, which significantly decreases the number of information queries sent to Grid resources during high service load. In order to avoid stale cache entries, resource information is renewed when more recent one becomes available. Metadata included with the resource information specifies how long the retrieved information is valid.

The *Broker* module initially validates incoming job requests, to ensure that a request includes all required attributes, such as the executable to run. Later in the submission process, the Broker is used

by the Submitter (or 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 WS-Agreement [5]. 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 job submission related data management tasks. 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 times. The DataManager can also stage job input files, unless this task is handled by the client or by the Grid resource executing the job.

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, if required, translates the job description from JSDL to a format understood by the Grid middleware of the resource. The Dispatcher also selects the appropriate mechanism to use to contact the resource. A plugin structure that combines the Chain of Responsibility and Adapter design patterns [28] enables the Dispatcher to perform these tasks without any a priori knowledge of the middleware used by the resource.

2.3. Advance reservations with WS-Agreement

As part of the job submission framework, we have made an implementation of the WS-Agreement specification [5], to be used for negotiating and agreeing on resource reservation contracts. The WS-Agreement 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 [18] 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 from a 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 also 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.

An Agreement 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 such a malleable reservation, the local scheduler is given the possibility to rearrange the local schedule. This may improve the resource utilization and partly compensate for the performance penalty imposed by the usage of advance reservations [22].

For advance reservations, two plugin scripts are required at the resource side. The first plugin negotiates reservations with the local scheduler. It is currently implemented for the Maui scheduler [47], but can easily be adopted to work with any local scheduler that supports advance reservations. The second plugin performs admission control of a job that requests to make use of a previously created reservation. This plugin needs to be integrated with the job management mechanism deployed at the Grid resource. The details of the integration of this plugin into existing Grid middlewares are discussed in sections 4.1 and 4.2.

Notably, the job submission service can also handle resources that do not have a reservation capable scheduler and 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 description, specified either in a middleware-specific format or in JSDL, the job submission framework allows a client to include an optional job preferences document in a job request. For example, this document can be used to choose brokering objective. 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 and information about job input and output files. This information is used to improve the resource selection process as described in Section 3.1. Just as it is optional to provide the job preferences document itself, all parameters in the document are optional.

3. ALGORITHMS FOR RESOURCE (CO)ALLOCATION

The general problem of resource brokering is complex, and the design of algorithms is highly dependent on the scheduling objectives, the type of jobs considered, the users' understanding of the application requirements, etc. For a general introduction to these issues, see e.g., [62, 76].

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 predefined 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. Notably, these two objectives give the same result if all resources are identical and jobs do not transfer any output data. This is however an unlikely scenario in Grid environments.

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 [19]. In order to estimate the TTS, the broker needs to, for each resource considered, predict the time required for staging of the executable and the input files, and the time the job must wait in the batch queue. 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*. Notably, by modifying the Selectors and/or Predictors, or by defining new ones, customization of the brokering algorithm is rather straightforward. The existing Predictions have 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 created for the job. The stage out predictor only considers the optional input provided by the user, as it is impossible to predict the size of the job output and hence also the stage out time without user input.

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, a less accurate prediction is made from the information provided by the resource about current load. This rough estimate does however not take into account the actual scheduling algorithm used by the batch system.

Time predictions for application 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 it is the responsibility of the user to ensure that performance characteristics of the selected benchmarks are representative for the application. A benchmark need however not at all 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.

Grid application runtime prediction consists of two separate problems. The first is to predict the performance of an application with fixed parameters on a set of machines, given the performance for the application (with the same set of parameters) on a known machine. The second problem is to, on a known machine, estimate the performance of an application for varying parameters based on knowledge of the performance of the application for a given set of parameters. The TTD and TTS algorithms address the first problem. The second problem is not specific to Grid jobs and have been studied extensively [21, 53, 64].

The TTD (and TTS) application performance models can be used to accurately predict the behavior of a wide range of applications, including compute and/or data intensive jobs. By considering input and output data transfers, resource access wait time and actual application execution, these performance models take into account and combine previous models such as *LeastLoaded* [59] and *DataPresent* [59] as well as the application-specific computational performance of the resource.

The implementations of the TTD and TTS models are carefully designed to predict higher performance for resources for which more detailed information and more accurate performance estimation mechanisms are available. From a resource selection perspective, it is however important to use the same metric (TTD or TTS) for comparison of all the resources of interest. This necessitates the use of coarse-grained prediction methods, e.g., estimating the queue waiting time from current resource load, when no better mechanism is available.

3.2. Coallocation

A coallocation mechanism is required in order to start a Grid job that makes coordinated use of more than one resource. 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 Submitter that allocates resources for single jobs.

The coallocation algorithm presented here share some characteristics with an algorithm by Mateescu [46]. Both these algorithms perform coallocation in an *on-line* style, i.e., the set of resources to coallocate is determined (and reserved) incrementally. The algorithm described in this paper is based on the advance reservation protocol outlined in Section 2.3 and does hence assume that it not known a priori whether a resource can be reserved at a given time or not. An alternative coallocation algorithm suggested by Wäldrich et al. [73] can be classified as *off-line* coallocation. This algorithm assumes preknowledge of when resources are available for reservation, determines the set of resources to use off-line, and creates reservations once a suitable set is found. A more in-depth discussion of the benefits and drawbacks of the respective approaches can be found in Section 6.

The main algorithm for identifying and allocating suitable resources for coordinated use 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.

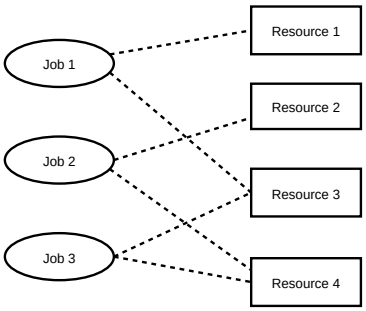


Figure 3. Subjobs and their possible resources viewed as a bipartite graph.

3.2.1. Problem definition

The input to the (on-line) coallocation problem is the following:

1. A set of $n \geq 2$ job requests: Jobs $= \{J_1, J_2, \ldots, J_n\}$.
2. A set of n resource sets, where each of the n sets contains the resources that are identified to have the capabilities required for one of the jobs:
   Resources $= \{R_1, R_2, \ldots, R_n\}$ where $R_1 = \{R_{11}, R_{12}, \ldots, R_{1m_1}\}, R_2 = \{R_{21}, R_{22}, \ldots R_{2m_2}\}$, $\ldots, R_n = \{R_{n1}, R_{n2}, \ldots, R_{nm_n}\}$, $|R_i| = m_i$, 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}.

For clarity, the coallocation algorithms are described for the case when all reservations start simultaneously. The algorithms can however perform any kind of job start time coordination, by allowing individual job start time offsets from a common start time. Although currently only implemented for computational jobs, the coallocation algorithms are general enough to allow coallocation of other resource types, e.g., networks. The only requirement for a resource type to be coallocated is the existence of an advance reservation mechanism that supports the protocol described in Section 2.3. The term *job* in the following descriptions can hence be read as *request for resource* (that supports advance reservations).

The jobs and resources forming the input to the coallocation problem can be expressed as a bipartite graph as illustrated in 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 hence 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 be 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 a 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 of a set of simultaneously starting reservations, one for each job. For practical reasons, a somewhat relaxed notion of simultaneous job start is used, reducing the simultaneous start time constraint to that all jobs must start within the same (short) period of time, expressed as a time window $[t_e, t_l]$. We remark that the coallocation requires the clocks of all resources to be synchronized in the order of $[t_e, t_l]$, using e.g., the Network Time Protocol (NTP) [49]. Typical values of $|[t_e, t_l]|$ is in the order of a half minute to a few minutes, whereas NTP can keep clocks synchronized within milliseconds [49]. Even though poorly synchronized clocks do not cause the coallocation algorithm itself to fail, clock drifts delay the start of the coallocated job, which in turn may cause batch system preemption as the total execution time of the job is increased.

The coallocation 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 increase the number of matched jobs 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, this 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 earliest possible reservation starts too late (failure).

3.2.3. Coallocation algorithms

The main coallocation algorithm is given in Algorithm 1 and the procedure (based on *augmenting paths*) for exchanging reservations between jobs 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 inputs 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 aligned 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 (success) or the $[t_e, t_l]$ window is moved outside $[T_e, T_l]$ (failure). 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 in time to align the $[t_e, t_l]$ window if an additional iteration of the main loop should be 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).

Algorithm 1 Coallocation

Require: A set of $n \geq 2$ resource requests (job requests) Jobs $= \{J_1, J_2, \ldots, J_n\}$.
Require: A set of resources with the capabilities required to fulfill these requests. 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.
Require: A start time window $[T_e, T_l]$ specifying earliest and latest acceptable job start.
Require: A maximum allowed start time deviation ϵ.
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_i 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 the current set of reservations.

Step 5 is performed for each job that has 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 in the reservation protocol ("<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, this job 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 obtained in 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.

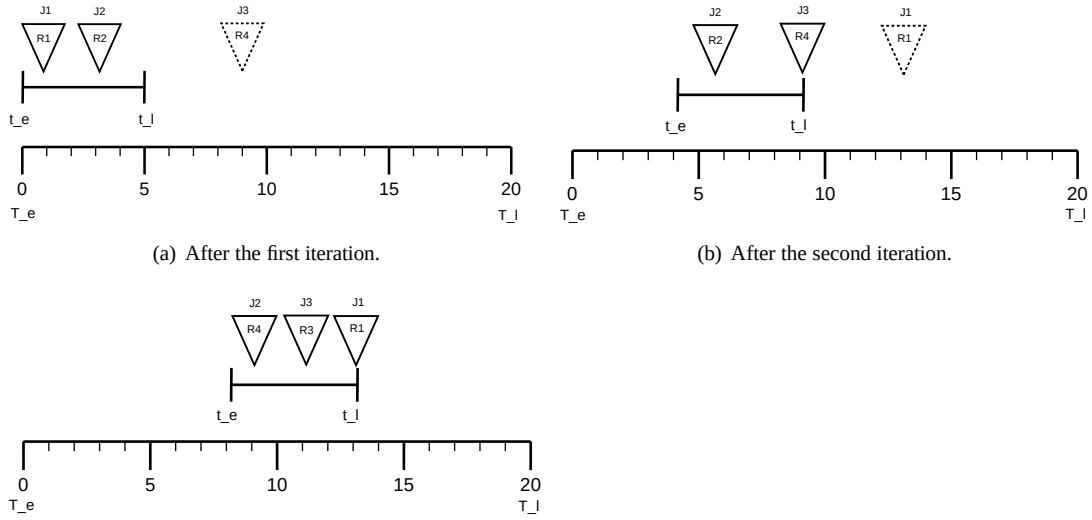
If augmentation techniques can be used according to the test in Step 13, the loop in Step 14 is executed for each job in A. In Step 15, an augmenting path of alternating unmatched and matched edges, starting and ending in an unmatched edge, is found using breadth-first search. In Step 16, the reservations (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, in analogy 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 main coallocation 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, i.e., to ensure the earliest possible job start, t_l is set to T_{best} and t_e is updated accordingly. Step 21 ensures that the $[t_e, t_l]$ window has not moved beyond the $[T_e, T_l]$ window. If 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.
8: Return.

Algorithm 2 is invoked in 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 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



(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.

if the modification fails. If so, it is not meaningful to continue the execution and Step 7 terminates the algorithm (with failure). Once the loop in Step 4 terminates without error and Step 8 is reached, the algorithm is successful.

Algorithms 1 and 2 describe the simplified coallocation scenario where all subjobs are coallocated for a concurrent job start. The actual implemented algorithm is more general as it allows each subjob start time to have an arbitrary offset from a common start time. This general scenario requires two minor extensions to the described algorithms. First, all comparisons with the $[t_e, t_l]$ window (steps 5, 6, 7, 9, 19, and 23 of Algorithm 1) must take into account the individual subjob's offset from this window. Secondly, special care must be taken when exchanging reservations between subjobs, as these need not have a common start time. For clarity, these extensions are left out from the descriptions of algorithms 1 and 2.

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 resources that meet 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 can however not 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 for J_2. 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 turn 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 here discuss advance reservations and the properties of the bipartite matching algorithm in more detail and also motivate 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 "flexible" flag (see Section 2.3) in all local schedulers, in order to maintain an efficient utilization of the resources.

3.3.2. *Modifications of 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 for this is that the Maui scheduler [47], 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 lose 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.

We also remark that the WS-Agreement specification [5] does not include an operation for renegotiation of an existing agreement (reservation). A protocol for managing advance reservations, including atomic modifications of existing reservations is discussed in [61]. 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 [6]) and supported by new releases of batch system schedulers. Should this happen, 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 represents 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 this 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 off-line. Therefore, we use a matching algorithm that (on-line) gradually increases the size of the current matching (initially containing no matched edges at all), and use path augmentation techniques to resolve conflicts. A more in-depth discussion of the difference between on-line and off-line coallocation algorithms is found in Section 6.

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, with the size defined in terms of the requested number of CPUs, required memory, and requested job runtime. We however remark that it is in the general case not possible to perfectly define such an ordering. For example, if one job requires two hours run time and one GB memory, and another only one hour but requires two GBs memory, it is not obvious which of these jobs to first create a reservation for.

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 noted that the test performed in Step 13 of Algorithm 1 may cause false positives, as it is assumed that path augmentation is possible 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 no augmenting path exists.

4. CONFIGURATION AND MIDDLEWARE INTEGRATION

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

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 determines connection timeouts, the number of threads to use in the threadpools, 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) [24]. 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 ones. 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 index service to cluster contact information only. 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 invocations. The InformationConverter plugin for GT4 is trivial as resource information in GT4 is described in 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 themselves 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. This is done by comparing the distinguish names in the X.509 certificate credentials used for the two tasks. In GT4, an Axis request flow chain that intercepts the job request performs this test. Authorization is hence performed in two steps, first by the custom Axis flow chain, and then by the GT4 gridmap authorization mechanism used by WS-GRAM.

4.2. Integration with ARC

The ARC middleware is based on Globus Toolkit 2 (GT2), but replaces some GT2 components, including the GRAM which is substituted by a *GridFTP server* that accepts job requests and a *Grid Manager* that manages accepted Grid jobs through their execution.

The information system in ARC uses GT2 tools, and is organized in 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 used in 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 sent to the local batch queue.

5. PERFORMANCE EVALUATION

The following section evaluates the performance of the job submission service, with an exclusive focus on the service itself, i.e., evaluating how it performs under varying configurations and load.

The resource selection algorithms described in this contribution are based on an estimate of the time required to execute a job in the Grid, either the TTS or the TTD. The accuracy of the algorithms varies with the accuracy of the predictions, which in turn depends both on the mechanisms available, e.g., advance reservations; and input provided by the user, e.g., relevant benchmarks and file transfer time estimates. Given good enough user input and prediction mechanisms as described above, the resource selection algorithms can give arbitrarily good predictions. For this reason, no evaluation of their performance is included in the paper.

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 these factors, the performance analysis includes 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. In addition, the performance of the coallocation algorithm is analyzed.

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 fixed period of time, and invokes the service as many times it can (in this case, submits as many jobs as possible) during the test period.

The response time measured by a tester includes the time required to establish secure connections to the job submission service, to delegate the user's credential to the service, and to submit the job. The response time also includes time for service side tasks such as broker job processing and interactions with index servers and resources. 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), whereas 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 in the performance tests.

The evaluation is performed with resources running either GT4 or ARC. In order to better understand how eventual bottlenecks in the Grid middleware effect the performance of the job submission service, each test uses a single Grid middleware on the resource side. We expect cross-middleware submission and resource brokering in mixed middleware Grids to be slightly slower, as the broker encounters the union of all bottlenecks in the used middlewares in such a scenario.

The performance measurements have been performed in a test 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 either GT 4.0.3 or ARC 0.5.56 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. Sixteen of these computers are equipped with AMD Athlon 64 2 GHz 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. 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 generated 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, gives the advantage that we know that there is no background load on the clusters. Hence, the performance of the job submission service can be significantly more accurately analyzed than it could have been if evaluated in a large production Grid (e.g, as performed in [18]).

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 `/bin/true` jobs, that do not require any input or output file staging. 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 and ARC. For each middleware, results are presented separately for tests using the Torque "PBS" scheduler and POSIX "Fork" as execution backends.

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 left-hand scales in the figures denote response times and the right-hand ones throughputs.

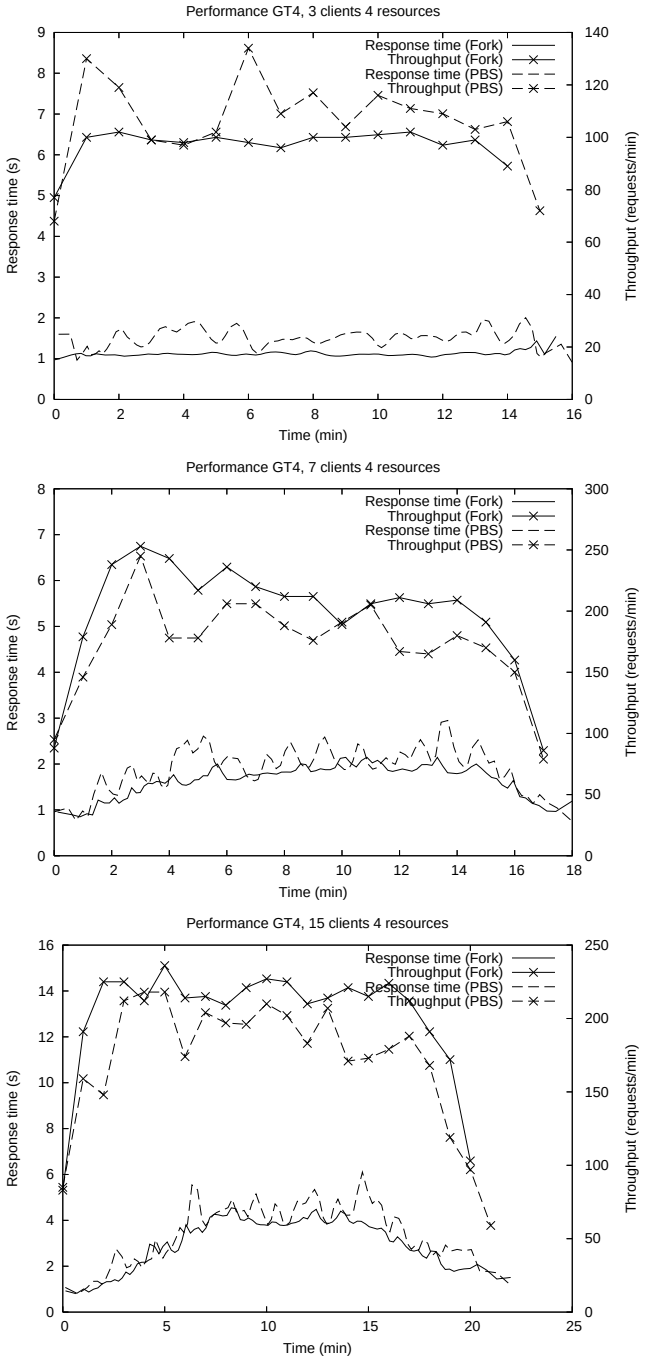


Figure 5. Performance results for GT4 using 4 resources.

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 can handle this load without problems. 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 7 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 7 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 just over 250 jobs per minute and to achieve individual job response times down to under one second.

For tests using the ARC middleware, Figure 6 shows the performance using four resources and 3, 7, and 15 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 7 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 when using ARC.

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 fluctuating response times using PBS than with Fork. However, if tests are done with jobs that require 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 with PBS can also be 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 turn 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 over time for both ARC and GT4. This decrease is most clearly visible in the tests using 7 and 15 clients. We initially suspected that this performance decline was due to scalability issues in the GT4 delegation service, investigated in [29]. During our performance tests of the delegation service (with a test setup similar to the job submission service tests) we observed a performance decline during heavy load. However, as the throughput of the delegation service is around three times higher than that of the

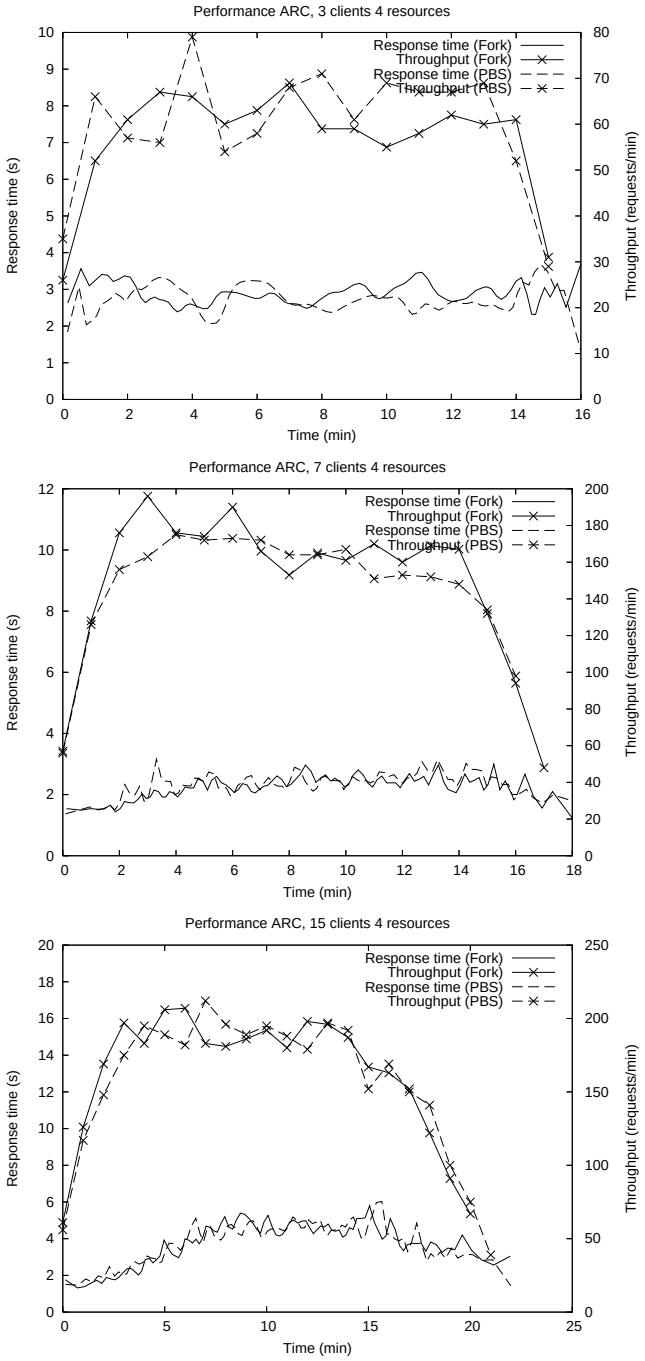


Figure 6. Performance results for ARC using 4 resources.

job submission service for similar loads, this performance decline is negligible. We also noted that the delegation service response time typically is between 0.3 and 0.8 seconds. This is a substantial part of the job submission time, especially considering that the job submission service can submit a job in less than one second, credential delegation included.

5.2.1. Advance reservations

In order to evaluate the impact of advance reservations on the job submission service performance, 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 with 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 has demonstrated [22, 66], the usage of advance reservations imposes 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 batch systems do not provide a lightweight reservation mechanism, we argue 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 with "$\times$") with reservations is about 40 submitted jobs per minute for all three tests. In these tests, the response time increases from about five seconds (3 clients), to ten seconds (7 clients), and finally to around 20 seconds (15 clients). In comparison, for jobs submitted without reservations, the throughput increases from around 100 jobs per minute (3 clients), to around 210 jobs per minute (7 clients), and finally increases a bit more to around 220 jobs per minute when 15 clients are used. The response times for these jobs are around two seconds (both 3 and 7 clients) and three seconds for 15 clients.

The performance results for tests of jobs with advance reservations submitted to ARC are very similar to the corresponding tests with GT4, and graphs for these tests are hence omitted. For jobs with reservations submitted to ARC, the throughput is around 30 jobs per minute for 3 clients, and 40 jobs per minute for 7 and 15 clients. The response time varies from around six seconds for 3 clients, to ten seconds for 7 clients and 20 seconds for 15 clients. In the reference tests where no jobs used reservations, the throughput is around 55 jobs per minute for 3 clients, 140 jobs per minute with 7 clients and 160 jobs per minute with 15 clients. In these tests, the response time is around three seconds for both 3 and 7 clients, and around four seconds for 15 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.

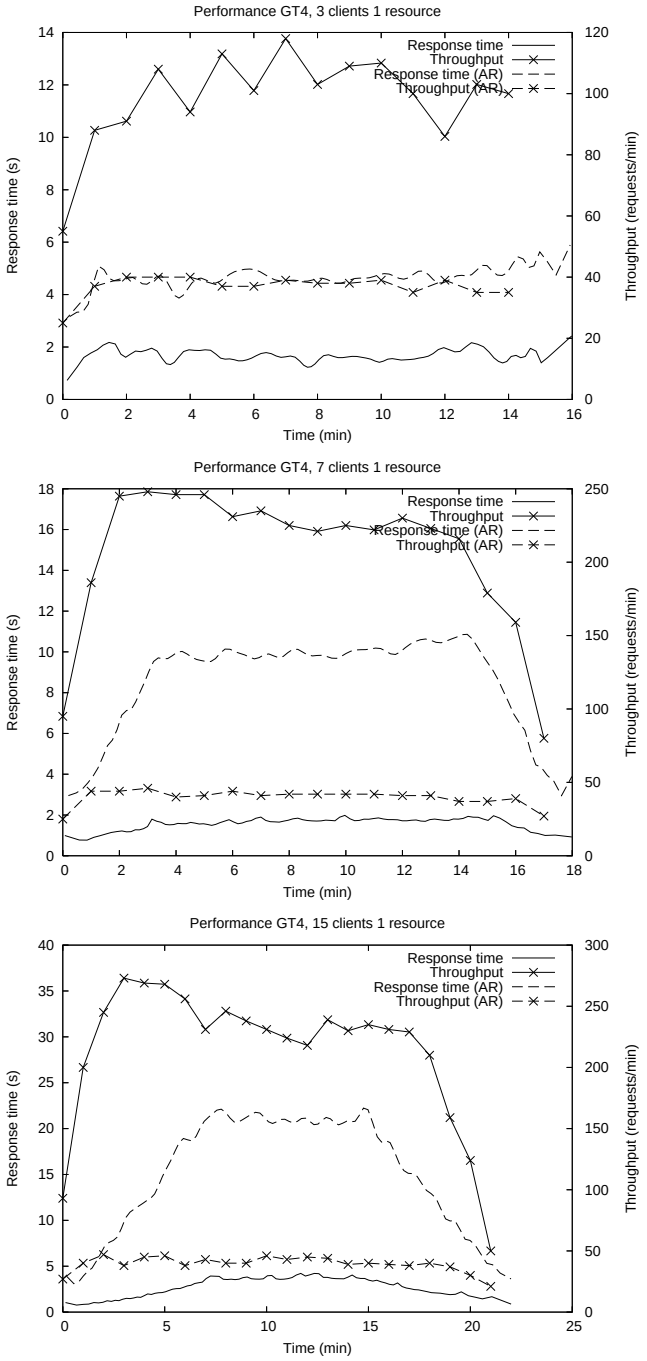


Figure 7. Performance results for advance reservations using GT4 and 1 resource.

5.2.2. Coallocation

The most time consuming parts of Algorithm 1 are creation of new reservations and modifications of existing ones, i.e., steps 9 and 7. The update procedure for augmenting paths (Algorithm 2) performs a series of reservation modifications and it follows from the design that its execution time increases linearly with the length of the augmenting path. The overall performance of the coallocation algorithm thus depends on how often these three mechanisms (create new reservation, modify existing reservation, path augmentation) are used. These numbers increase with the number of iterations of the algorithm that are executed. The number of iterations in turn depends on multiple factors, including the number of requested jobs, how many resources that are capable of executing each job, the degree of overlap in these sets of resources, the current load of each resource (most notably, the fragmentation of the backfill windows of the local schedulers), etc.

We have performed a series of tests to demonstrate the robustness of the coallocation algorithm, and illustrate how it performs for a given combination of coallocation request type and Grid infrastructure configuration. The results from these tests increase the understanding of the characteristics of the algorithm, but cannot, due to the intrinsic performance dependencies discussed above, be generalized beyond the particular configuration used in the tests.

The coallocation tests are performed in the test environment described in Section 5.1. In each test, background loads are created on all machines, a coallocation request is issued, and results from the coallocation algorithm execution are gathered. In all tests, four jobs are requested to be coallocated over four machines. The length of the job start time window is four hours, the maximum allowed job start time deviation (ϵ) is five minutes and the requested length of each job is one hour. Each machine has a 20 minutes long reservation as background load that is randomly placed within the four hour job start time window. Notably, this setup means that it is not always possible, even in theory, to fulfill the coallocation request.

As the machines in the test environment are identical, the path augmentation procedure is per default never required for successful coallocation. For this reason, the tests are divided into two sets: tests where path augmentation is not required, and tests where heterogeneity in the test environment is simulated to necessitate path augmentation. We refer to these as *homogeneous tests* and *heterogeneous tests*, respectively. In the homogeneous tests, each of the four requested jobs can make use of all four resources. The same hold for three of the jobs in the heterogeneous tests, whereas the fourth job in these tests can execute on only one of the machines.

5.2.2.1. Homogeneous tests. A summary of 1000 homogeneous tests, divided into successful and failed coallocation requests, is shown in Table I. This table shows the average, minimum and maximum value for the following metrics: execution time of the coallocation algorithm, the number of iterations performed, and the usage frequency of the three most time consuming operations (create new reservation, update existing reservation, and path augmentation). The table is divided into results for successful invocations of the coallocation algorithm and results for cases where the algorithm does not manage to obtain a coallocation.

Table I shows that the successful coallocation attempts are faster to execute than the failed ones. This is due to the fact that the algorithm stops when a coallocation is found, whereas the failed attempts have to scan the complete job start time window before concluding that no coallocation is possible. From the average values of the execution time and the operation counts, we conclude that the coallocation

Table I. Results for 1000 homogeneous tests of the coallocation algorithm. The term AR denotes advance reservation.

Successful coallocations (86.0%)			
	average	min	max
execution time (s)	15.02	3.30	38.65
# iterations	3.02	1	7
# new ARs	4.79	4	10
# failed new ARs	19.89	0	59
# modified ARs	4.09	0	12
# failed modified ARs	0.79	0	6
# path augmentations	0	0	0
# failed path augmentations	2.01	0	6
Failed coallocations (14.0%)			
	average	min	max
execution time (s)	27.91	13.77	39.92
# iterations	5.04	4	7
# new ARs	6.64	2	10
# failed new ARs	40.64	24	59
# modified ARs	6.71	3	11
# failed modified ARs	3.75	0	7
# path augmentations	0	0	0
# failed path augmentations	5.02	1	7

algorithm is able to perform slightly more than two advance reservation operations (create or modify) per second. This observation takes into account that a modify request performs two service invocations (removal of an old reservation and creation of a new one, see Section 3.3.2), and holds both for the failed and the successful coallocation attempts.

The successful attempts range from trivial solutions where all jobs are reserved in the first iteration and in less than four seconds, to complex scenarios where up to seven iterations with more than 60 advance reservation operations performed during 40 seconds are required to coallocate the job. The failed attempts search through the whole coallocation window, and hence show much less deviations in performance. The observable deviations are due to differences in the distribution of the background load.

5.2.2.2. Heterogeneous tests. A summary of 1000 heterogeneous tests is shown in Table II. This table has both the same division into successful and failed coallocation attempts and the same metrics as Table I. We note that also for the heterogeneous tests, the successful attempts are faster and the coallocation algorithm carries out just above two advance reservation operations per second. It follows from the construction of the coallocation algorithm that, as only one job in the heterogeneous tests may require path augmentation, the maximum number of successful path augmentations is one. For the successful attempts, the average number of successful path augmentations is 0.81. The reason for this

Table II. Results for 1000 heterogeneous tests of the coallocation algorithm. The term AR denotes advance reservation.

Successful coallocations (66.3%)			
	average	min	max
execution time (s)	9.21	3.32	22.56
# iterations	1.97	1	4
# new ARs	3.65	3	7
# failed new ARs	7.44	0	27
# modified ARs	1.82	0	7
# failed modified ARs	0.46	0	4
# path augmentations	0.81	0	1
# failed path augmentations	0.67	0	3
Failed coallocations (33.7%)			
	average	min	max
execution time (s)	15.71	7.62	23.40
# iterations	3.30	2	4
# new ARs	5.42	1	8
# failed new ARs	16.32	3	32
# modified ARs	3.28	0	7
# failed modified ARs	2.77	0	5
# path augmentations	0	0	0
# failed path augmentations	2.59	0	4

number being less than one is that in some tests, none of the first three jobs reserve the only resource that the forth job can use, and path augmentation is hence not required. Failed path augmentations occur, when an augmenting path is found and augmentation hence appears possible (Step 15 of Algorithm 1), but a background load job prevents the new reservation (Step 1 of Algorithm 2) from being created. We note that all failed coallocation attempts have in common that no path augmentation operation is successful.

Although results for the homogeneous and the heterogeneous tests are not directly comparable with each other due to the intrinsic performance dependencies of the coallocation algorithm, we observe that, as the number of potential matchings between jobs and resources are fewer in the heterogeneous tests, these tests are faster to execute than the homogeneous ones.

Tables I and II list the execution time of the coallocation procedure as described in Algorithm 1. In a complete job coallocation scenario, tasks such as job submission service invocation, job request validation, resource discovery and information retrieval must also be performed. These tasks are similar to the initial steps executed during submission of individual jobs and, as discussed earlier, take approximately one to three 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.

In addition to the above performance observations, the coallocation evaluation also demonstrates that the implementation of the coallocation algorithm is robust, as no errors (except for failed coallocation

attempts, that are not errors per se) occurred during the 2000 tests, that had a total execution time of more than seven hours.

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 and coallocation in Grids, and Grid interoperability efforts. In the following, we make a brief review of these.

6.1. General resource brokering

The composable ICENI Grid scheduling architecture is presented in [77], 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 [60] supports submission and monitoring of Grid jobs. Features include usage of the GLUE information model [4] and a mechanism where users can control the resource selection by weighting the importance of attributes such as CPU frequency 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 [75]. 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 [75]. Examples of work on economic brokering include [10, 12, 20, 52]. An alternative to market-based economies is Grid-wide fairshare scheduling [16] that can be viewed as a planned economy.

6.2. Performance prediction

One method for selecting the submission target for a computational job is to predict the performance of the job 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 [2, 38, 43, 65, 70] and (ii), heuristics based on job and resource characteristics [34, 44, 74].

In our previous work [19], 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).

A Grid application performance model similar to TTD is described by Ali et al. [1]. In this work, application execution time and batch queue waiting time are both predicted using method (i), whereas file transfer times are estimated from file size and bandwidth information.

6.3. Interoperability efforts

There are several resource brokering projects which target resources running different Grid middlewares, e.g., Gridbus [72], which can schedule jobs on resources running, e.g., Globus [30], UNICORE [68], and Condor [45]. The GridWay project [36] targets resources running both protocol oriented (GT2) and service-based versions (GT4) of the Globus toolkit as well as LCG [40]. 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 (or directly in JSDL), whereas the respective job description languages of Gridbus and GridWay are fixed on the client side.

In the contribution by Pierantoni et al. [57], Metagrid Services are used as a bridge between users and different Grids. In this architecture, condensed graphs are used to express workflows of jobs. Interoperability is demonstrated among WebCom (a workflow engine), GT4, and LCG2 [57]. Similar ideas are explored by Kertész et al. [39], who define an architecture for a meta-broker and a language for communicating broker requirements in addition to job requirements. Instead of performing resource selection, such a meta-broker selects the best Grid resource broker and hence creates a hierarchy of Grid brokers. Common features in our contribution and the work by Kertész et al. is the use of JSDL and a plugin-based architecture for interaction with specific middlewares. The UniGrids project [71] specially targets interoperability between the Globus [30] and UNICORE [68] middlewares. The Grid Interoperability Now (GIN) [31] initiative focuses on establishing islands of interoperation between existing Grid resources, and growing those islands to achieve an increasing set of interoperable Grids. The goal of the Open Middleware Infrastructure Institute (OMII) Europe [54] is to make components for job management, e.g., the OGSA Basic Execution Service [33]; data integration; and accounting available for multiple platforms, including gLite [14], Globus [24], and UNICORE [68].

There are some projects that have adopted JSDL to describe jobs, e.g., [32, 39, 50].

6.4. Advance reservations

Several contributions conclude that an advance reservation feature is required to meet QoS guarantees in Grid environments [26, 35, 63]. Unfortunately, the support for reservations in the underlying infrastructure is currently limited. Qu describes a method to overcome this shortcoming by adding a Grid advance reservation manager on top of the local scheduler(s) [58]. Advance reservations can hence be provided regardless of whether the local scheduler supports them. This reservation approach however requires that all job requests are passed through the Grid advance reservation manager.

The performance penalty imposed by the usage of advance reservations (typically decreased resource utilization) has been studied [66, 67]. The work in [22] investigates how performance improvements can be can be achieved by allowing laxity (flexibility) in advance reservation start times.

Standardization attempts include [61], which defines a protocol for management of advance reservations. The more recent WS-Agreement [5] standard proposal defines a general architecture that enables two parties, the agreement provider and the agreement initiator, to enter an agreement. Although not specifically targeting advance reservations, WS-Agreement can be used to implement these, as demonstrated e.g. by [18, 48, 73].

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 that requires 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 selection of which resources to coallocate.

The Globus Architecture for Reservation and Allocation (GARA) [26] provides a programming interface to simplify the construction of application-level coallocators. GARA can perform both immediate reservations (allocations) and advance reservations. The system furthermore supports several resource types, including networks, computers and storage. The GARA project is focused on the development of a library for coallocation agents and only outlines one possible coallocation agent [26], 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 resources. Our coallocation algorithm also differs from GARA as our algorithm allows for a flexible reservation start within a given interval of time.

The authors of the KOALA system [51] propose a mechanism for implementing coallocation without using 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 jobs are ready to start executing.

The work by Mateescu [46] defines an architecture for coallocation based on GT2. Mateescu's 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. One difference is that the algorithm by Mateescu 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 and 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. [73] models reservations using the WS-Agreement framework and uses the concept of coallocation iterations. 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.

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 (on-line), whereas the algorithm by Wäldrich et al. is based on an off-line calculation of which resources to use. Furthermore, our coallocation algorithm allows a user-specified fluctuation in reservation start times, while the algorithm described in [73] uses a fixed notion of reservation start time.

The advantages of on-line coallocation include fewer requirements on the local scheduler, as on-line algorithms need not know all available time slots in advance. Not all local schedulers allow users to see the current backfill-window. Furthermore, in order to be accessible by a Grid coallocation service, the backfill-window must be included in the information advertised by the Grid information system.

Information retrieved from such a system can be both incomplete and outdated. Even if the backfill-window is available and up-to-date there are possible complications. The existence of a free slot in the backfill window is not a guarantee that a certain user may reserve this slot. Any reservation request may, e.g., due to policy reasons be denied. Furthermore, nodes in a cluster can be heterogeneous in the number of cores, available memory etc. This implies that information about the backfill window is not enough to (off-line) match a job with specific requirements to some time slot. These complicating factors suggest, as argued in Section 2.3, that reservations should be created on a trial-and-error basis. The main benefit of off-line coallocation algorithms is that they use fewer reservations and can hence be seen as more efficient.

The off-line coallocation algorithms use methods resembling optimistic concurrency control for transactions [42], whereas on-line algorithms can be described as a more pessimistic locking approach. In the coallocation context, a lock is equivalent to a reservation. Which of the two approaches that is better from a transaction perspective much depends on the likelihood of (resource reservation) failure. Further work in the coallocation area should investigate the likelihood of failures and also leverage the theory developed for distributed transactions.

The GridARS project defines a protocol for advance reservations and coallocation of computational and network resources [69]. The protocol specifies a two-phase commit and does hence provide safe transactions for coallocation. There is however no description of the actual algorithm used to select and coreserve the resources.

The work described in [3] reuses the concept of barriers from [11]. In [3], the coallocator architecture consists 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 a hierarchy of coallocators. The barrier functionality developed in [3] also supports the synchronization of hierarchically coallocated jobs. Our work differs from [3], 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. [56] whereas other work [9] 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 when the following four conditions hold simultaneously: (i) mutual exclusion, (ii), hold and wait, (iii) no preemption, and (iv), circular wait [37]. Our algorithm modifies (or releases) reservations for resources whenever it 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 and NorduGrid/ARC. Hence, job and resource requests can be specified in any of these two input formats, and independently, the jobs can be submitted to resources running any of these 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 job submission service is designed for decentralized use, i.e., typically to be used by a single user or a small group of users, the performance analysis demonstrates 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 during high load. At best, the job submission service is able to give individual job response times below one second and to provide a total throughput of over 250 jobs per minute.

The scientific contributions in this work are mainly in two directions. We conclude that the current set of Grid and Web service standards enables interoperability between different Grid middlewares, although only at a fundamental level. We have however demonstrated that cross-middleware interoperability need not be restricted to the least common denominator of the used middlewares. Even though middleware specific job description attributes are lost in translation, other mechanisms, e.g., the job preferences document used in the job submission service, allow users to express QoS requirements for their jobs. Use of proper infrastructure extensions, e.g., the WS-Agreement services, enable such requirements to be fulfilled across different middlewares.

The other direction of contributions of this work is the proposed coallocation algorithm that allows users to perform arbitrarily coordinated allocations of multiple resources. Even though currently only implemented for computational resources, the algorithm is general enough to be used to coordinate use of any reservable resource, e.g., network bandwidth. The discussion of the differences between the on-line and off-line coallocation approaches adds to the understanding of the various problems that must be addressed in a coallocation scenario.

Future directions for this work include adaptation of the current architecture and interfaces to adhere to more recent emerging standards such as the OGSA Basic Execution Service [33] and the OGSA Execution Management Services [27]. An ongoing effort is the integration of the job submission service with the LCG2/gLite middleware. As the job submission service replaces the LCG2/gLite Resource Broker component in this scenario, the only involved components are the GT2 GRAM computing element and the information system, the latter based on the Berkeley Database Information Index (BDII). The translation of the Condor-style *classads* used as job description in LCG2/gLite is rather tedious as classads do not define a schema of valid attributes but rather allow any value-pair expression.

We also plan to develop a library for job coordination of 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]. Based on our earlier experiences with Grid workflows [17], we currently investigate how the coallocation algorithm can be used to improve QoS for job pipelines in a Grid data-flow scenario.

8. SOFTWARE AVAILABILITY

The software described in this paper is available at `www.gird.se/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. We also acknowledge the three anonymous referees for constructive comments. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

REFERENCES

1. A. Ali, A. Anjum, T. Azim, J. Bunn, A. Mehmood, R. McClatchey, H. Newman, W. ur Rehman, C. Steenberg, M. Thomas, F. van Lingen, I. Willers, and M.A. Zafar. Resource management services for a Grid analysis environment. In W-C. Chun and J. Duato, editors, *Proceedings of the 2005 International Conference on Parallel Processing*, pages 53–60. IEEE Computer Society, 2005.
2. 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*, 2004.
3. S. Ananad, S. Yoginath, G. von Laszewski, and B. Alunkal. Flow-based multistage co-allocation service. In B.J. d'Auriol, editor, *Proceedings of the International Conference on Communications in Computing*, pages 24–30, Las Vegas, 2003. CSREA Press.
4. S. Andreozzi, S. Burke, L. Field, S. Fisher, B. Kónya, M. Mambelli, J. M. Schopf, M. Viljoen, and A. Wilson. Glue schema specification version 1.3. http://glueschema.forge.cnaf.infn.it/Spec/V13, January 2009.
5. 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.
6. 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.
7. 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.
8. Apache Web Services Project - Axis. http://ws.apache.org/axis, January 2009.
9. 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 L. Barolli, editor, *Proceedings of the eleventh International Conference on Parallel and Distributed Systems*, pages 36–42, 2005.
10. 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.
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. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 741–750. Springer Verlag, 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. IEEE Computer Society, 2004.
14. EGEE. gLite - lightweight middleware for grid computing. www.cern.ch/glite, August 2008.
15. 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.
16. 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.
17. E. Elmroth, F. Hernández, and J. Tordsson. A light-weight Grid workflow execution engine enabling client and middleware independence. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 4967*, pages 754–761. Springer Verlag, 2007.

18. 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.
19. E. Elmroth and J. Tordsson. A Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, 24(6):585–593, 2008.
20. 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.
21. Thomas Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, 1996.
22. 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. IEEE Computer Society, 2005.
23. International Organization for Standardization. ISO/IEC 2382-1 information technology - vocabulary - part 1: Fundamental terms, 1993.
24. I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin, D. Reed, and W. Jiang, editors, *IFIP International Conference on Network and Parallel Computing, LNCS 3779*, pages 2–13. Springer Verlag, 2005.
25. 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. www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf, January 2009.
26. 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 M. Zitterbart and G. Carle, editors, *7th International Workshop on Quality of Service*, pages 27–36. IEEE, 1999.
27. 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, January 2009.
28. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
29. P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, and T. Sandholm. Scalable grid-wide capacity allocation with the SweGrid Accounting System (SGAS). *Concurrency Computat.: Pract. Exper.*, 20(18):2089–2122, 2008.
30. Globus. http://www.globus.org. January 2009.
31. Grid Interoperability Now. http://wiki.nesc.ac.uk/read/gin-jobs. January 2009.
32. GridSAM. http://gridsam.sourceforge.net. January 2009.
33. A. Grimshaw, S. Newhouse, D. Pulsipher, and M. Morgan. OGSA Basic Execution Service version 1.0. https://forge.gridforum.org/sf/go/doc13793, January 2009.
34. 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.
35. 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.
36. E. Huedo, R.S. Montero, and I.M. Llorente. A framework for adaptive execution on Grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004.
37. E. G. Coffman Jr, M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):68–78, 1971.
38. N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational Grid environment. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 71–80, 1999.
39. A. Kertész and P. Kacsuk. Meta-broker for future generation Grids: A new approach for a high-level interoperable resource management. In *CoreGRID Workshop on Grid Middleware in conjunction with ISC'07 conference, Dresden, Germany*, 2007.
40. J. Knobloch and L. Robertson. LHC computing Grid technical design report. http://lcg.web.cern.ch/LCG/tdr/, January 2009.
41. K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of Grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 15(32):135–164, 2002.
42. H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.
43. H. Li, J. Chen, Y. Tao, D. Gro, and L. Wolters. Improving a local learning technique for queue wait time predictions. In S.J. Turner, B.S. Lee, and W. Cai, editors, *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 335–342. IEEE Computer Society, 2006.

44. 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.
45. 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.
46. 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, 2003.
47. Maui Cluster Scheduler. http://www.clusterresources.com/products/maui/, January 2009.
48. 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.
49. D. Mills. Network time protocol (version 3) specification, implementation and analysis. http://www.ietf.org/rfc/rfc1305.txt, January 2009.
50. K. Miura. Overview of japanese science Grid project NAREGI. *Progress in Informatics*, 1(3):67–75, 2006.
51. 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.
52. R. Moreno and A. B. Alonso-Conde. Job scheduling and resource management techniques in economic Grid environments. In F. Fernández Rivera, M. Bubak, A. Gómez Tato, and R. Doallo, editors, *Grid Computing - First European Across Grids Conference, LNCS 2970*, pages 25–32, 2004.
53. F. Nadeem, M. M. Yousaf, R. Prodan, and T. Fahringer. Soft benchmarks-based application performance prediction using a minimum training set. In *In Second International Conference on e-Science and Grid computing, Amsterdam, Netherlands*. IEEE press, 2006.
54. OMII Europe. http://omii-europe.org, October 2008.
55. Open Grid Forum. www.ogf.org. January 2009.
56. J. Park. A scalable protocol for deadlock and livelock free co-allocation of resources in internet computing. In S. Helal, Y. Oie, C. Chang, and J. Murai, editors, *Symposium on Applications and the Internet, SAINT 2003*, pages 66–73. IEEE Computer Society, 2003.
57. G. Pierantoni, B. Coghlan, E. Kenny, O. Lyttleton, D. O'Callaghan, and G. Quigley. Interoperability using a Metagrid Architecture. In *ExpGrid workshop at HPDC2006 The 15th IEEE International Symposium on High Performance Distributed Computing*, 2006.
58. C. Qu. A Grid advance reservation framework for co-allocation and co-reservation across heterogeneous local resource management systems. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 4967*, pages 770–779. Springer Verlag, 2007.
59. K. Ranganathan and I. Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid computing*, 1(1):53–62, 2003.
60. 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.
61. A. Roy and V. Sander. Advance reservation API. http://www.ogf.org/documents/GFD.5.pdf, January 2009.
62. J.M. Schopf. Ten actions when Grid scheduling. In J. Nabrzyski, J.M. Schopf, and J. Węglarz, editors, *Grid Resource Management State of the art and future trends*, chapter 2. Kluwer Academic Publishers, 2004.
63. M. Siddiqui, A. Villazon, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized QoS. In *Proceedings of the 2006 ACM/IEEE SC—06 Conference (SC'06)*, 2006.
64. W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 1459*, pages 122–142, 1999.
65. 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, LNCS 1459*, pages 202–219, 1999.
66. W. Smith, I. Foster, and V. Taylor. Scheduling with advance reservations. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 127–132. IEEE, 2000.
67. 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 Verlag, 2000.
68. A. Streit, D. Erwin, Th. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and Ph. Wieder. UNICORE - from project results to production grids. In L. Grandinetti, editor, *Grid Computing: The New Frontiers of High Performance Processing, Advances in Parallel Computing 14*, pages 357–376. Elsevier, 2005.
69. A. Takefusa, H. Nakada, T. Kudoh, Y. Tanaka, and S. Sekiguchi. GridARS: An advance reservation-based Grid co-allocation framework for distributed computing and network resources. In E. Frachtenberg and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 4942*, pages 152–168. Springer Verlag, 2007.

70. D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling user runtime estimates. In D.G. Feitelson and. E. Frachtenberg, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 3834*, pages 1–35, 2005.
71. UniGrids. www.unigrids.org. January 2009.
72. 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, 2006.
73. 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.
74. 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.
75. 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.
76. 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.
77. L. Young, S. McGough, S. Newhouse, and J. Darlington. Scheduling architecture and algorithms within the ICENI Grid middleware. In S. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting*, pages 5–12, 2003.