An Interoperable, Standards-based Grid Resource Broker and Job Submission Service

Erik Elmroth and Johan Tordsson
Dept. of Computing Science and HPC2N
Umeå University, SE-901 87 Umeå, Sweden
{elmroth, tordsson}@cs.umu.se

Abstract

We present the architecture and implementation of a Grid resource broker and job submission service, designed to be as independent as possible of the Grid middleware used on the resources. The overall architecture comprises seven general components and a few conversion and integration points where all middleware-specific issues are handled. The implementation is based on state-of-the-art Grid and Web services technology as well as existing and emerging standards (WSRF, JSDL, GLUE, WS-Agreement). Features provided by the service include advance reservations and a resource selection process based on a priori estimations of the total time to delivery for the application, including a benchmark-based prediction of the execution time. The general service implementation is based on the Globus Toolkit 4. For test and evaluation, plugins and format converters are provided for use with the NorduGrid ARC middleware.

1. Introduction

The resource broker and job submission components are vital for any Grid computing infrastructure, as their functionality and performance to a large extent determine the user's experience of the Grid. In all, these components have to identify, characterize, evaluate, select, and allocate the resources best suited for a particular application. The brokering problem is complicated by the heterogeneous and distributed nature of the Grid as well as the differing characteristics of different applications. To further complicate matters, the broker typically lacks total control and even complete knowledge of the state of the resources.

Typically, resource brokers are closely integrated with, or at least heavily dependent on, some particular Grid middleware, with popular solutions ranging from brokering components being part of the job submission client to centralized Grid-schedulers not that different from traditional batch system schedulers [16, 7, 20, 15, 5, 13]. Hence, it is normally non-trivial to migrate a broker from one middleware to another, or to adjust it to simultaneously work with resources running different middlewares.

This contribution presents an architecture and an implementation of a general service for Grid resource brokering and job submission. The service is general in the sense that it can be used with different Grid middlewares, with middleware-specific issues concentrated to minor components. These components are used for format conversions in interactions with clients and information systems as well as for middleware-specific interaction with resources.

The proposed broker and job submission service rely heavily on Grid and Web services standards, including JSDL, WSRF, WS-Agreement, and GLUE (see Section 2), and are implemented using Globus Toolkit 4 (GT4) [10]. Middleware-specific interfaces are provided for the NorduGrid ARC software [8, 14], which is based on the Globus Toolkit 2 (GT2). Tests and evaluation have been performed on SweGrid [18] and NorduGrid [14] resources.

The brokering scenario addressed by our solution is a decentralized broker that acts on behalf of the user in order to allocate the resources that best fulfill the user's request. Hence, the broker does not take any globally controlling role and works independently of any other broker or job submission software interfacing the same resources. All decisions made by the broker are based on the user's requests and the information (including negotiation) it extracts from the resources and information services. Notably, the broker may be used simultaneously by multiple users, but the brokering scenario remains as described above.

The broker aims at identifying the set of resources that minimizes the *Total Time to Delivery (TTD)*, or part thereof, for each individual job submission [9]. In order to do this, the broker makes an a priori estimation of the whole or parts of the TTD for all resources of interest before making the selection. The TTD estimation includes performing a *benchmark-based execution time estimation*, *estimating file* *transfer times*, and performing *advance reservations* of resources in order to obtain a guaranteed batch-queue waiting time. For resources not providing all information required or a reservation capability, less accurate estimations are performed.

The rest of this paper is organized as follows. Section 2 introduces some standards and technologies used, Section 3 gives an in-depth description of the system and Section 4 describes the resource selection algorithms used. Section 5 illustrates the integration of the system with an existing Grid middleware, whereas sections 6 and 7 contain a performance evaluation and conclusions, respectively.

2. Background and standards used

The presented brokering and job submission architecture makes extensive use of existing and proposed Grid and Web service standards not only for interaction with other components but also internally. The most important standards used are briefly presented below.

2.1. JSDL

The Job Submission Description Language (JSDL) proposed by the Global Grid Forum (GGF) describes the configuration of computational jobs and their requirements on the resources that executes them. It is the result of the JSDL working group's attempts to create a standardized job description language, simplifying interoperability between existing resource management systems [3].

In our contribution, the JSDL is used to express job requests sent to the job submission module by clients.

2.2. WSRF

The Web Services Resource Framework (WSRF) [11], defines a relationship between stateful resources and Web services. This relationship is modelled using a construct called a WS-Resource. An endpoint reference addresses a Web service, and may also identify one of the WS-Resources associated with that service.

The WSRF consists of five specifications, including the following. WS-ResourceProperties defines the type and value of the WS-Resource's state as viewable through a Web service interface. The WS-ResourceLifetime specification defines lifecycle management of WS-Resources, including creation and destruction (immediate or scheduled for later). WS-BaseFault defines a base type for fault handling in Web services, which increases consistency.

In our work, WSRF, and more specifically, WS-Resources are used to represent jobs and reservations (agreements). Information about submitted jobs and created reservations is modelled using WS-ResourceProperties. The lifetime management mechanisms defined in WS-ResourceLifetime are used to implement soft-state, two-phase reservations. WS-BaseFault is used for error messages.

2.3. WS-Agreement

WS-Agreement is a GGF standard proposal, which makes it possible for an *agreement initiator* and an *agreement provider* to enter an agreement. This agreement specifies service level objectives associated with the use of one or more Web services. The WS-Agreement standard does not specify any domain-specific terms describing the service level objectives, but is rather intended for use with any type of Web service. Service domain-specific terms are expected to be added in extensions for each service domain of interest [2].

The basic operation of WS-Agreement is straightforward. Initially, the agreement initiator retrieves an *agreement template* (prefilled contract) from the agreement provider. The initiator fills out the relevant parameters in the template and sends the resulting *agreement offer* in a request to the agreement provider. Upon granting the offer, the agreement provider creates a WS-Resource representing the agreement, and returns an endpoint reference to this WS-Resource to the agreement initiator.

The AgreementFactory porttype stores the agreement templates and exposes an operation for requesting an agreement. The Agreement porttype exposes no operation, it only holds the WS-Resources modelling created agreements. A third porttype, the AgreementState, is used to monitor the fulfillment of the agreement. Notably, WS-Agreement neither defines a protocol for agreement negotiation, nor states how agreements should be signed.

In our work, WS-Agreement is used to negotiate and represent advance reservations for batch systems.

2.4. GLUE

The Grid Laboratory Uniform Environment (GLUE) project [1] defines an information model for describing Grid resources, targeting core services such as resource discovery and monitoring. Resource discovery services benefit from an extensive list of resource characteristics. For monitoring, state information describing load and availability is defined. The GLUE model (version 1.2) describes computing elements, storage elements, and mappings relating these. The GLUE project targets a model usable by different technologies. Current implementations include LDAP schemas for GT2 and XML schemas for GT4.

In our job submission service, the GLUE format is used to represent resource information gathered during resource discovery.

3. Architecture

The proposed brokering and job submission framework is based on a general architecture with seven components and an implementation of the WS-Agreement specification. The framework is complemented with a job submission client and some middleware-specific components, currently available for the NorduGrid ARC middleware.

The job submission service itself is implemented using GT4 [19, 10], and does, just like GT4, make extensive use of Axis [4]. Presently, plugins for reservations are implemented for the Maui scheduler [12].

Below we give an architecture overview, followed by more detailed descriptions of each of the modules, including some discussions on design considerations.

3.1. Overview

The job submission module consists of seven components: the *InformationFinder* performs resource discovery and retrieves resource information; the *Broker* performs resource selection; the *Reserver* negotiates advance reservations; the *DataManager* handles file transfers; the *Dispatcher* sends job requests to the resource; the *Submitter* coordinates the work of the five first modules and finally the *JobSubmissionService* which stores information about submitted jobs and provides a Web service interface to the job submission module. In addition to these components, there is a user client for sending job requests to the JobSubmissionService. The system also includes an implementation of the WS-Agreement specification, hosted on the Grid resource.

Figure 1 gives an overview of the modules. Their interactions and main operations are the following. Upon receiving a job request from the client, the JobSubmissionService passes the job description along with any optional parameters to the Submitter. The Submitter first invokes the Broker to validate the job description. Then, the InformationFinder is used to retrieve a list of available Grid resources. After receiving this list, the Submitter calls the Broker to filter out unsuitable resources and to rank the suitable ones. The ranking procedure may include creating advance reservations, which is handled by the Reserver. If required, the DataManager is then invoked to stage input files. Then, the Submitter uses the Dispatcher to submit the job to the selected resource and returns the obtained job identifier to the JobSubmissionService.

The JobSubmissionService creates a stateful resource (WS-Resource) storing information associated with the job. In the final step, the job identifier is returned to the client.

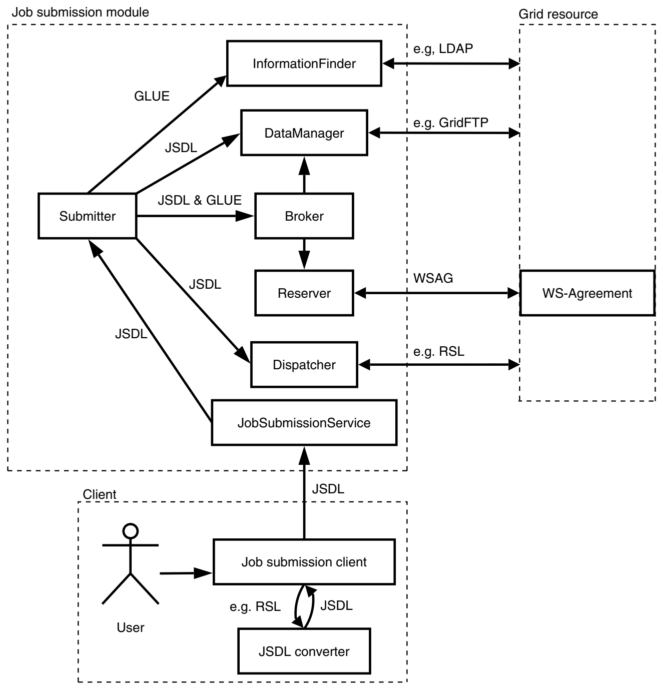

Figure 1. Architecture overview showing components, hosts, and information flow. The boxes show the modules and the dashed lines denote the different hosts.

3.2. Modules

This section presents the finer details of the modules in the system.

JobSubmissionService. The JobSubmissionService is the only component in the job submission module accessible by clients. It exposes one operation, *SubmitJob*, through its Web service interface. The parameters for this operation are the JSDL job description, and optionally, a document describing the user's job preferences and/or a list of URLs of index servers, e.g., GT2 GIISes or GT4 Index Services, to contact during resource discovery.

Notably, the JobSubmissionService does not expose any other operation than job submission. Further job management, such as job monitoring and control are beyond the scope of this service. The JobSubmissionService stores stateful information about each submitted job, including the job identifier, the job description and, if applicable, information about an advance reservation created for the job.

A WS-Resource is created for each successfully submitted job, with job information stored as resource properties. By querying the resource properties, other job management tools can retrieve the job identifier in order to monitor and control the execution of the job.

As the JobSubmissionService typically is invoked by a user, a job submission client is provided. The arguments to the client include those passed in a job request to the JobSubmissionService, i.e., the mandatory job description and the optional job preferences document and index server URLs. In addition to these parameters, a user can specify which JobSubmissionService to contact. The client supports a plugin structure enabling translations to JSDL from any native job description language preferred by the user.

InformationFinder. The purpose of the InformationFinder is to discover what Grid resources are available and to retrieve more detailed information about each resource. The input to this module is a list of index server URLs.

The InformationFinder first performs resource discovery by querying each index server, which due to their hierarchical organization may require some recursive invocations. Then, each identified resource is queried in more detail. Both static and dynamic information about the resource is retrieved, including hardware and software configuration and current load. The InformationFinder also retrieves usage policies, allowing it to discard resources where the user is not authorized to submit jobs.

Unless the information retrieved in the detailed queries follows the GLUE format, it is converted by the InformationFinder before being returned to the Submitter. To improve performance, the resource discovery, the information retrieval, and the conversion to GLUE format are each performed in parallel. A fixed size thread pool is used to control the degree of parallelism, thus avoiding overloading the hosting environment. In order to reduce the resource discovery overhead in situations where the same resources are repeatedly queried during a short period of time, the retrieved resource information is stored in a time-limited cache.

Broker. The Broker strives to select the best resource for each incoming job request. What makes a resource the "best resource" depends on the characteristics of the job and the resources, as well as on the user's preferences.

The Broker provides three operations. Validation of job description ensures that the description contains all required attributes, e.g., the application to run. Resource filtering guarantees that only resources fulfilling the job's requirements on architecture, disk, memory etc. are considered for submission. The most complex operation, resource ranking, ranks the resources according to their suitability for executing the job and reorders the resource list accordingly. Two interfaces are defined to facilitate this operation.

The *predictor* interface is used for the estimation of how long time a certain task associated with the job would require if performed by a certain resource. The four tasks considered for time estimation are: staging input files to the resource; waiting for resource access, e.g., in a batch queue; executing the job on the resource; and staging output files. These four tasks make up the Total Time to Delivery (TTD) for the Grid job. The algorithms used for TTD estimations are presented in [9] and reviewed in Section 4.

The *selector* interface is used for the actual ranking of the resources. The ranking is done using some, possibly all, of the predictors. Currently, the Broker contains two selectors. The earliest start selector ranks resources based on the stage in and wait predictors, in order to achieve an as early job start as possible. In contrast, the earliest completion selector tries to minimize the TTD, and hence achieve the earliest possible job completion.

Resource ranking may include the time-consuming task of negotiating advance reservations. To improve performance, resources are ranked in parallel, using a thread pool similar to the one in the InformationFinder.

DataManager. The DataManager is responsible for all data management tasks that relate to job submission. The module defines an interface for staging of files to and from Grid resources. This interface also defines the resolution of physical location(s) of replicated files, which is useful if the Grid middleware supports file replication. Another operation defines the prediction of the duration of file transfers, which can be implemented if the underlying infrastructure supports either network reservations or bandwidth performance predictions. The DataManager also implements an operation for determining the sizes of physical files. This operation is used as a last resort for predicting the duration of file transfers when none of the more sophisticated mechanisms mentioned above are available.

Reserver. The Reserver includes a client API for reserving CPUs at computational resources in advance. These CPUs are reserved for a certain duration and with either a fixed start time or an interval of allowed start times.

The three operations defined in the Reserver API are creation of a temporary reservation, confirmation of a temporary reservation, and cancellation of a reservation (temporary or confirmed). Temporary reservations are released shortly after their creation unless they are confirmed. These operations are implemented using calls to the WS-Agreement module.

The Reserver also contains a repository of created reservations. After a job is successfully submitted to a resource, the Submitter examines the repository and confirms the reservation on the selected resource and cancels any other reservation created for the job.

WS-Agreement. This module includes implementations of the AgreementFactory and Agreement porttypes defined by the WS-Agreement specification. Unlike the other modules, which are hosted on the machine running the JobSubmissionService, WS-Agreement is deployed on the Grid resource.

Note that our current implementation does not include the AgreementState porttype. For our target domain, advance reservations of CPUs, monitoring the state makes little sense. A created reservation will, short of resource failure, fulfill the guarantees specified in the agreement.

Two interfaces are defined in order to guarantee that the WS-Agreement implementation is agnostic of the service domain for which the agreements are created. Both interfaces must be implemented when using WS-Agreement for a specific service, and any service-specific operation has to be placed within the implementations of these interfaces.

The AgreementDecisionMaker interface determines whether to grant or deny an agreement offer, returning an AgreementDecision, which, in addition to the actual decision, contains any domain specific context associated with the created agreement. The AgreementDecisionMaker concept first appeared in Cremona [6].

The AgreementResourceHelper constructs domain specific agreement terms for inclusion in the resource property document of the WS-Resource representing the agreement.

To the best of our knowledge, there exists no earlier work using WS-Agreement to model batch queue reservations. For this reason, a language for describing reservations against computational resources is defined. Each agreement offer (reservation request) contains the duration of the reservation and the requested number of CPUs. Furthermore, a start time window specifying earliest and latest allowed job start is included. A flag named flexible specifies whether the start time of the reservation may be moved within the start time window by the local batch system. If allowing this, backfilling can be performed more efficiently, resulting in increased utilization [17].

The AgreementFactoryService passes an incoming agreement offer to the ReservationDecisionMaker, which executes a plugin creating the earliest possible reservation within the start time window specified in the offer. If no reservation can be created within this window, the offer is denied. After the ReservationDecisionMaker has granted the offer, the AgreementFactoryService creates a WS-Resource and invokes the ReservationResourceHelper to create resource properties for this WS-Resource. The resource properties include an identifier for the reservation, the exact reservation start time and the parameters in the agreement offer. Finally, the endpoint reference to the WS-Resource is returned to the agreement initiator (the Reserver). Figure 2 shows the interactions between the Reserver and the WS-Agreement services.

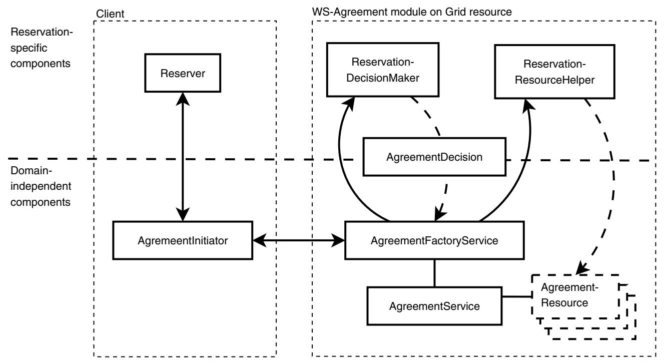
Client
WS-Agreement module on Grid resource
Reservation-specific components
Reserver
Reservation-DecisionMaker
Reservation-ResourceHelper
AgreementDecision
Domain-independent components
AgreementInitiator
AgreementFactoryService
AgreementService
Agreement-Resource

Figure 2. Interactions between general WS-Agreement components and the reservation specific modules.

Dispatcher. The Dispatcher is responsible for sending the job request to the selected resource. While this task may seem trivial, the job description may first have to be translated (back) from JSDL to the job description language understood by the resource. Furthermore, the mechanism used for sending the job request to the resource depends on the Grid middleware used on the resource. Possible approaches include invocation of a Web service, as used by Globus WS-GRAM, and interaction with a GridFTP server, which is the mechanism used by NorduGrid ARC. Due to these significant differences, the Dispatcher contains no code common for all middlewares, but rather defines a general interface for dispatching jobs. This interface includes the job dispatch operation, which takes three arguments, the GLUE information about the selected resource, the job description (in JSDL format), and optionally, information about an advance reservation created for the job. The interface also defines the translation of job descriptions from JSDL to the native job description language of the used Grid middleware.

Submitter. The Submitter coordinates the job submission process. When a job request is passed from the JobSubmissionService, the Submitter invokes the Broker to validate the job description. If the description is valid, the Submitter calls the InformationFinder to retrieve an updated list of resource information. Once the resource list is updated, the Broker is invoked twice by the Submitter. First for filtering out inadequate resources, then for reordering the remaining ones after their suitability for executing the job. This second step may include requesting advance reservations, a task handled by the Reserver. Unless the Grid middleware performs file staging, the Submitter next invokes the DataManager to stage input files. It then calls the Dispatcher to send the job request to the most suitable resource. If one of these two operations fails, the Submitter retries with the second most suitable resource etc., until the job either is successfully submitted or all submission attempts fail, the latter causing an error message to be returned to the client. After completing these tasks, the Submitter returns a middleware-specific job identifier to the JobSubmissionService.

It may seem superfluous to separate the Submitter and the JobSubmissionService into two layers. We do however believe that the separation of the job submission process handled by the Submitter and the management of stateful resources performed by the JobSubmissionService is beneficial. This allows the construction of alternative Submitters, e.g., for coallocation or workflow scheduling.

4. Resource Brokering Algorithms

The resource broker selects the resources that gives the minimum predicted TTD (or part thereof) for the application. The algorithms for predicting the TTD depend on the support provided by the resources and the optional information supplied by the user. Below, we review the algorithms used for predicting the TTD, originally presented in [9], and describe what optional information a user can provide in order to improve the brokering process.

4.1. A priori estimation of TTD

The TTD for a Grid job includes the times for (1) staging in the executable and the input files to the resource, (2) waiting in the batch queue, (3) executing the application, and (4) staging output files to their requested location(s). The Broker presented in Section 3 makes use of one predictor for each of these tasks. The two provided selectors make use of the first two and all four predictors, respectively.

If the DataManager provides support for predicting file transfer times or for resolving physical file locations and determining file sizes (see Section 3), these features are used by the predictors of (1) and (4) for estimating file staging times. If no such support is available, e.g., depending on the information provided by the Grid middleware used, these predictors make use of the file transfer times optionally provided by the user. Notably, the time estimate for stage in is important not only for predicting the TTD but also for coordinating the start of the execution with the arrival of the executable and the input files if an advance reservation is performed.

The most accurate prediction of the batch queue waiting time (2) 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 to deactivate this feature, less accurate estimates are made from the information provided by the resource about current load. This estimation, however, does not take into account the actual scheduling algorithm used by the batch system.

The prediction of (3) is performed through a benchmark-based estimation of execution time 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 benchmarks. For more information, e.g., how information about multiple benchmarks is used, see [9].

4.2. Optional input for brokering

In addition to the JSDL document describing the job, a user may include a job preferences document in the job request sent to the JobSubmissionService. This document contains both job requirements and additional information that may improve the resource selection process.

The job requirements are the preferred job objective, which can be either earliest job completion or earliest job start. The job start offset enables users to request that the job starts after a certain time, which can be expressed either as an absolute time or a relative offset from now. This feature can be used e.g., for debugging, demonstrations and coallocation purposes. Users can also specify a latest allowed job start, ensuring that the job either starts in time or is not submitted at all. Users with no strong requirements on the start or completion times of their jobs can specify that no reservation should be created for the job. Such jobs receive best-effort job start times, which facilitate improved resource utilization [17].

The job preferences document is also where a user may provide the additional information mentioned above, that allows the broker to improve the resource selection. This includes the predicted times for file staging and information used for benchmark-based execution time estimation.

Just as the job preferences document itself is an optional parameter in the job request message, all parts of the job preferences document are optional. However, by making use of this feature, expert users can benefit from their knowledge of the job characteristics.

An example of a user preference document is shown in Figure 3. In this document, a user specifies two benchmarks, nas-lu-c and specFP2000, as characteristic for the performance of the job. The user specifies the (possibly predicted) application execution times 60 and 45 minutes on machines where the results of these benchmarks are 450 and 750, respectively. The file stage out time is estimated to 10 minutes. Notably, file staging predictions are normally very inexact as the resource which to stage files to and from actually is unknown. Still, a rough approximation may often provide additional value in situations where the broker has no other information about network performance.

In the job requirement part, the user specifies the earliest allowed job start, and also states that the job may start no more than 30 minutes later than the earliest allowed start time. The job objective is an as early job completion as possible.

```
<JobPreferences>
    <SchedulingHints>
        <Benchmark name="nas-lu-c"
            result="450" time="60"/>
        <Benchmark name="specFP2000"
            result="1750" time="45"/>
        <FileTransfers>
            <StageOutTime>10</StageOutTime>
        </FileTransfers>
    </SchedulingHints>
    <JobRequirement>
        <EarliestAbsoluteStart>
            2005-08-11T10:00:00Z
        </EarliestAbsoluteStart>
        <LatestRelativeStart>
            30
        </LatestRelativeStart>
        <JobObjective>
            EarliestJobCompletion
        </JobObjective>
    </JobRequirement>
</JobPreferences>
```

Figure 3. Example of a user job preferences document.

5 Configuration and Middleware Integration

The integration of the general job submission module with a particular Grid middleware requires some configuration and some customized components. Below we summarize what needs to be done, including the set up of the Submitter to interact with a specific middleware and how to configure the WS-Agreement module to support reservations of computational resources. Finally, we illustrate this by the steps taken in our integration with the NorduGrid ARC [14, 8].

The basic operation of the JobSubmissionService is controlled by the configuration of the Submitter. This configuration decides which plugin modules are used by the InformationFinder for discovering resources, for querying resources, and if needed, for converting the information retrieved to the GLUE format. The configuration file also controls which Dispatcher to use. These settings do, i.e., determine which Grid middleware to use when communicating with the Grid resources. As resource discovery can be rather time-consuming, it is possible to specify a timeout to use when discovering and querying resources. Further tuning of the performance of the InformationFinder can be done by adjusting the maximum number of threads to use in the thread pool. The configuration file also includes URLs to one or more default index servers, which are used in the resource discovery phase unless the user includes URL(s) to index server(s) in the job request. A configuration file containing the above described parameters is passed to the JobSubmissionService upon service startup.

The configuration of the WS-Agreement services determines which agreement template(s) to store in the AgreementFactoryService. Also included in the configuration is the DecisionMaker to use when determining whether to grant an agreement or not. The configuration file may also include a list of DecisionMaker initialization parameters. In the reservation scenario, these parameters are used to specify plugin scripts that invoke the local scheduler when creating and cancelling reservations. Support for other schedulers than the currently supported Maui scheduler can easily be implemented by creating new reservation plugin scripts and reconfiguring the AgreementFactoryService. The configuration file also specifies which ResourceHelper to use.

5.1. Integration with NorduGrid ARC

Here, we illustrate the integration of the brokering and job submission service with NorduGrid ARC, a middleware based on GT2 with some GT2-components replaced or modified.

A major difference between ARC and GT2 is the job management. In ARC, the server-side GT2 GRAM components (gatekeeper and job manager) are replaced by custom components, a GridFTP server and a Grid Manager, respectively [8]. Another difference is that even though ARC uses the GT2 MDS, this is done with customized schemas. Information advertised by a ARC GRIS describes users, jobs and resources (clusters and their job queues).

For integrating the job submission module with ARC, plugins are required for the Dispatcher and the InformationFinder. Furthermore, a few server-side scripts are required for managing the advance reservations created by the WS-Agreement components.

The hierarchal MDS structure used in ARC makes the resource discovery plugin in the InformationFinder straightforward. Starting from the list of GIISes provided on input, all GIISes are recursively queried for resources, each by a separate thread, without calling any GIIS more than once. Then, the resource query plugin requests information about the cluster and its queues, using LDAP. The result of these queries is objects providing an ARC-specific description of the resources, which are converted to the GLUE format by the converter plugin.

An ARC job submission client procedure includes uploading any locally stored input files to the resource (the Grid Manager handles stage in of non-local input files) before sending the job description to the GridFTP server of the resource. The ARC dispatcher plugin converts the JSDL job description to the GT2-style RSL used by ARC, and then modifies the job description to ensure that also local input files are staged by the Grid Manager. If an advance reservation is created for the job, the identifier of the reservation is added to the job description before it is uploaded.

The implementation of WS-Agreement and the local scripts used by the ReservationDecisionMaker operates independently of ARC. However, a mechanism is required to associate the user creating the reservation with the one submitting the job. This is done by ensuring that the job submission and the creation of the reservation both are performed using proxies that originate from the same certificate.

A general infrastructure such as the job submission module can normally not capture all the features available in every single middleware, e.g., ARC. However, for use with ARC, the only noticeable shortcoming is that in ARC, files accessible through the Globus Replica Location Service (RLS) can serve as job input and output, whereas the job submission module currently only handles locally stored files and files stored at (Grid)FTP servers. Support for RLS may however be added in future versions of the job submission module.

6. Performance evaluation

The job submission module has been evaluated with respect to the service response time, i.e., the time required to submit a job, and the service throughput, i.e., number of jobs submitted to resources per minute.

The evaluation has been performed with the client and job submission module each running on a system equipped with one 2.8 GHz Intel P4 processor with 1 GB memory, Debian Linux Sarge and Globus Toolkit version 4.0.0. The WS-Agreement services were deployed on a system with a 667 MHz Intel P3 processor, 384 MB memory, Debian Linux Sarge, Globus Toolkit 4.0.0, Maui 3.2.6 and Torque 1.1.0. The Grid infrastructure used is a subset of NorduGrid and SweGrid, with in total 12 resources ranging from four to 388 CPUs (including the six 100 CPU clusters in SweGrid). These resources used the NorduGrid ARC middleware and were indexed by four GIISes, with one serving as higher level GIIS for the others. The LDAP information gathered from the resources was valid for 30 seconds, which hence became the cache expiration time. A timeout of 15 seconds was used for all InformationFinder connections and a maximum of eight threads were used in all threadpools.

In order to evaluate the service response time, a client submitted a series of jobs, waiting with the submission of the next one until the submission of the previous job was completed. Hence, the time measured includes the broker's processing of one single job and all waiting times associated with the resource selection and job submission of that job.

As negotiation of advance reservations is rather time-consuming, tests were performed both with and without reservations. The job response times were grouped into five classes depending on the time required, as shown in tables 1 and 2. The tables summarize five sets of 200 jobs each, showing the average, the minimum and the maximum percentage of job submissions in each interval for each set.

Table 1. Job run time distribution without reservations.

	<2 s	2-5 s	5-8 s	8-11 s	>=11 s
average	9.4%	66.1%	15.4%	3.9%	5.2%
min	7.5%	64%	12%	2%	2.5%
max	12.5%	68.5%	19.5%	7%	8.5%

Table 2. Job run time distribution with reservations.

	<7 s	7-10 s	10-13 s	13-16 s	>=16 s
average	11.4%	37.4%	33.3%	12.4%	5.5%
min	8.5%	32.5%	30%	6.5%	2.5%
max	16.5%	51%	38%	15.5%	10.5%

For jobs submitted without reservations (Table 1), the majority of the submissions take 2–5 seconds, and around 75% of them take less than 5 seconds. This corresponds to jobs where the broker can take advantage of cached resource information and no resource has any exceptionally long response time for job submission. Around 15% of the jobs take 5–8 seconds, corresponding to jobs where the cached information has expired and additional resource queries therefor are performed. The last two categories, i.e., jobs that take more than 8 seconds, include jobs for which additional delays were encountered. These delays were due to an overloaded Grid infrastructure and resulted in two issues: increased time for resource discovery and, more time-consuming, slower dispatch of the job to the selected resource. The latter operation was particularly slow if many jobs recently had been submitted to the same resource.

In the results for submissions performed with advance reservations in Table 2, we basically see the same pattern but with around 5 seconds longer times. One side effect of these longer times is that fewer jobs can benefit from the cached resource information, explaining the smaller number of submissions falling into categories 1–2.

For the throughput tests, jobs were submitted concurrently from many clients, in order to put a high load on the job submission module. To reduce the overhead associated with the first invocation of a Web service, each client submitted a number of jobs. The results show up to 40 successfully handled job submissions per minute, even though the performance varied due to load variations on the Grid infrastructure, resulting in the same anomalies as for the first test. We conclude, however, that for submissions up to at least 40 jobs per minute by the job submission module, it is rather the resources than the broker and job submission service that are the bottleneck.

7 Conclusions

We have presented the design and implementation of a general framework for Grid job submission and resource brokering. The framework is intended to be as middleware independent as possible, and it is therefore to a large extent based on (proposed) Grid and Web services standards. We have demonstrated how the general framework can be integrated in the NorduGrid ARC middleware. In an evaluation of the framework integrated in an environment with ARC resources, we conclude that most jobs can be submitted in less than 5 seconds (less than 13 seconds if using advance reservations), and that the job submission module is capable of achieving a throughput of at least 40 submitted jobs per minute.

The integration requires some effort, in particular when translating the job description language used in ARC back and forth to JSDL, and when converting information retrieved from ARC resources to GLUE. The time spent developing the ARC plugins was however only a fraction of the time required to implement the complete framework, illustrating that implementing plugins for an additional middleware is a feasible method of constructing a feature-rich job submission client for that middleware.

The strive for interoperability often boils down to finding the lowest common denominator between the various systems. In this case, we have been able to adopt the ARC formats to the various standard formats used in our system without losing much of the original ARC features.

Future directions in this work include implementing plugins for additional Grid middlewares. Current efforts focus on support for GT4 and gLite. The future plans also include extending the framework to support coallocation of resources. This will mainly require the development of an alternative submitter module as all the other modules basically have the functionality required.

Acknowledgement

The authors are grateful to Peter Gardfjäll, Aleksandr Konstantinov, and Åke Sandgren. We also acknowledge the three anonymous referees for constructive comments. This work was funded by The Swedish Research Council (VR) under contract 343-2004-953.

References

[1] S. Andreozzi, S. Burke, L. Field, S. Fisher, B. Kónya, M. Mambelli, J. M. Schopf, M. Viljoen, and A. Wilson. Glue schema specification version 1.2 draft 7. Internet, 2005. http://infnforge.cnaf.infn.it/docman/view.php/9/90/GLUEInfoModel_1_2_draft_7.pdf.

[2] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Internet, 2004. https://forge.gridforum.org/projects/graap-wg/document/WS-AgreementSpecification/en/7.

[3] 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. Internet, 2005. https://forge.gridforum.org/projects/jsdl-wg/document/draft-ggf-jsdl-spec/en/21.

[4] Apache Web Services - Axis. http://ws.apache.org/axis.

[5] M. Dalheimer, F.-J. Pfreundt, and P. Merz. Calana: A General-purpose Agent-based Grid Scheduler. Proceedings of Parallel Computing 2005. To be published.

[6] A. Dan, H. Ludwig, and R. Kearney. Cremona: An architecture and library for creation and monitoring of ws-agreements. In *ICSOC'04*, USA, 2004. ACM.

[7] C. Dumitrescu, I. Raicu, and I. Foster. DI-GRUBER: A Distributed Approach to Grid Resource Brokering. SC'05 2005.

[8] P. Eerola, B. Kónya, O. Smirnova, T. Ekelöf, M. Ellert, J. Hansen, J. Nielsen, A. Wäänänen, A. Konstantinov, J. Herrala, M. Tuisku, T. Myklebust, F. Ould-Saada, and B. Vinter. The NorduGrid production Grid infrastructure, status and plans. In *Proc. 4th International Workshop on Grid Computing*, pages 158–165. IEEE CS Press, 2003.

[9] E. Elmroth and J. Tordsson. A Grid resource broker supporting advance reservations and benchmark-based resource selection. In *State-of-the-art in Scientific Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[10] I. Foster. A Globus toolkit primer. http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf.

[11] 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. Internet, 2005. http://www-106.ibm.com/developerworks/library/specification/ws-resource/ws-modelingresources.pdf.

[12] Maui Cluster Scheduler. http://www.clusterresources.com/products/maui.

[13] J. Nabrzyski, J. M. Schopf, and J. Węglarz, editors. *Grid Resource Management*. Kluwer, 2003.

[14] NorduGrid. http://www.nordugrid.org.

[15] D. Ouelhadj, J. Garibaldi, J. MacLaren, R. Sakellariou, and K. Krishnakumar. A multi-agent infrastructure and a service level agreement negotiation protocol for robust scheduling in Grid Computing. In Peter M. A. Sloot et al., editor, *Advances in Grid Computing - EGC 2005*, 2005.

[16] I. Rodero, J. Corbalán, R. M. Badia, and J. Labarta. eNANOS Grid Resource Broker. In Peter M. A. Sloot et al., editors, *Advances in Grid Computing - EGC 2005*, 2005.

[17] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In J. Rolim et al., editors, *IPDPS*, 2000.

[18] SweGrid. http://www.swegrid.se.

[19] The Globus Alliance. http://www.globus.org.

[20] S. Venugopal, R. Buyya, and L. Winton. A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids. Technical Report GRIDS-TR-2004-1, University of Melbourne, Australia, Feb. 2004.