Umeå University
Department of Computing Science

14th June 2004

Resource Brokering for Grid Environments

Master's thesis

Johan Tordsson

Supervisors:
Erik Elmroth
Åke Sandgren

Examiner:
Per Lindstöm

Abstract

A resource broker is a central component in a grid environment. The purpose of the broker is to dynamically find, characterize and allocate the resources most suitable to the user's applications.

This thesis describes the development of a broker that bases resource selection on load and performance of the resources as well as characteristics of the application, aiming at minimizing the total time to delivery for each application. The total time to delivery includes the time required for input/output file transfer, batch queue waiting and application execution.

The methods used to minimize the total time to delivery includes the usage of advance reservations, characterization of resource performance by the use of benchmarks and estimation of data transfer times from network performance predictions. Additionally, a basic queue adaptation mechanism is used in order to adapt to changes in load on the grid.

Keywords: Resource broker, grid scheduling, benchmark-based resource selection, network bandwidth predictions, advance reservations, adaptation, Globus toolkit.

Contents

1 Introduction **1**

2 Grid computing **1**
2.1 Background . . . 1
2.2 The Globus toolkit . . . 2
2.3 Resource brokering . . . 6

3 The NorduGrid middleware **7**
3.1 Components and Services . . . 8
3.2 Brokering within NorduGrid . . . 11

4 Measuring brokering performance **12**
4.1 Scenarios . . . 13

5 Algorithms and implementations **14**
5.1 Techniques used to estimate the TTTD . . . 14
5.2 Job submission . . . 16
5.3 Network file transfers . . . 18
5.4 Advance reservations . . . 22
5.5 Benchmark based execution time prediction . . . 26
5.6 Features . . . 28

6 Usage scenarios **30**

7 Discussion **31**
7.1 Related work . . . 31
7.2 Future work . . . 32
7.3 Conclusion . . . 35

8 Acknowledgements **35**

A List of abbreviations **40**

B Example of broker configuration file **41**

List of Figures

1	Chain of trust including CA, user and proxies.	3
2	RSL job request specifying executable, input and output files and number of processes.	4
3	RSL job request including a list of values and the greater-than operator.	5
4	RSL job request using the disjunct-request.	5
5	Job submission using the Globus toolkit 2.	7
6	States and state transitions for a NorduGrid job	11
7	Interaction between NorduGrid components.	12
8	Overview of program flow for job submission.	16
9	High-level job submission algorithm.	17
10	Algorithm for determining TTTD.	19
11	Algorithm for advertising bandwidth predictions.	20
12	Predicting time required for file transfer to or from a given cluster.	21
13	Algorithm for making a reservation.	23
14	Predicting execution time using benchmarks.	27
15	Program flow for queue adaptation.	28
16	Program flow for submitting jobs using I/O-synchronization.	29
17	XRSL job request including benchmarks relevant for the job.	30
18	XRSL job request with estimation of job output size.	30

List of Tables

1	Examples of tools in the NorduGrid user interface.	8
2	Overview of “GERE” command.	25
3	Overview of “RERE” command.	26

1 Introduction

This thesis describes the design and implementation of a resource broker aiming at minimizing the total time to delivery for each application submitted by the user. Methods used to achieve this includes prediction of run time using benchmarks, usage of advance reservations to guarantee job start times, and prediction of file transfer times from network bandwidth forecasts.

The rest of this thesis is organized as follows. Section 2 gives an introduction to grid computing and describes the basic architecture of the Globus toolkit 2. The section also defines the resource brokering problem, both in general terms and implemented using the Globus toolkit. The NorduGrid middleware is described in Section 3. Section 4 discusses performance of resource brokers, introduces the cost model used and lists some representative use cases. The algorithms and implementation of the broker are described in Section 5. Section 6 contains usage scenarios showing possible usage of the broker. Related and future work are discussed in Section 7, which also contains some concluding remarks. Section 8 is devoted to acknowledgements. Last in the thesis, lists of references and abbreviations are found.

2 Grid computing

This section gives a short background to grid computing and introduces concepts, standards and Application Programming Interfaces, APIs, used throughout the thesis.

2.1 Background

Over the last years, network capacity has increased tremendously, bandwidth tends to double every 9 months, to be compared with CPU speed that roughly doubles every 18 months according to Moore's law. During the late 1980's and early 1990's, researches used the emerging high performance networks to interconnect remotely located parallel computers. These proof-of-concept experiments, often referred to as *metacomputing*, showed that it was possible to achieve higher performance than obtainable from one single parallel computer. One of the most successful experiments was I-WAY [7], a project including more than 60 applications using distributed, heterogenous resources. These early applications used custom made protocols for accessing, controlling and integrating resources. The need for standard protocols and tools soon lead to the development of toolkits such as Legion [16] and Globus [13, 41], the latter today a de-facto standard for building grid infrastructures.

In their area founding article, "The Anatomy of the Grid" [12], the authors identify several use cases where "a number of mutually distrustful participants with varying degrees of prior knowledge (perhaps none at all) want to share resources in order to perform some task." This sentence very much captures the reason why a *Virtual Organization*, VO, should be formed. People participating in the virtual organization often belong to different physical organizations. Resources are typically computational resources such as CPUs, disk space, network bandwidth, but the term resource also includes such diverse entities as scientific instruments and software licenses. The term *grid computing* was coined to

describe this broader context of distributed collaboration and resource interconnection, and this term has replaced metacomputing.

2.2 The Globus toolkit

As the Globus toolkit 2 is used extensively in this thesis, the main components of the toolkit are discussed. The Globus alliance web page [41], contains more in-depth information about the toolkit.

GSI

Grid computing, being distributed and heterogeneous in its nature, has high demands on security. The *Grid Security Infrastructure*, GSI [11] specifies a public key infrastructure and SSL (TLS) [17] for authenticated and private communication. For more information about public key encryption, see, e.g., [35].

All users and resources in the grid are identified by a *certificate*, specified in the X.509 format [46]. Certificates can be used to authenticate and identify the owner of the certificate. In order to do this, some information is required. First of all, the certificate contains the *Distinguished Name*, DN, of the owner. This name has a format specified by the hierarchical name space of LDAP, e.g., `/O=Grid/O=NorduGrid/OU=hpc2n.umu.se/CN=Johan Tordsson`. Also included in the certificate is the public key of the owner. The DN and the public key contain enough information to authenticate the user. However, no resource or user will trust a certificate just because it contains a DN and a public key as such information can be generated by anyone. A trusted third-party is required to guarantee the identity of the user and the certificate. A *Certificate Authority*, CA provides this guarantee. The CA signs the certificate by including its DN and digital signature in the certificate. Anyone who trusts the CA can now trust the certificate. The CA has its own certificate, signed by another, even more trusted CA, and a *chain of trust* is formed. The certificate of a user is stored in an encrypted file in the local file system, the user must hence enter a password in order to access the certificate.

Authenticated communication with multiple grid resources would normally require the user to (re)type a password more times than feasible. A *single sign-on* mechanism is implemented using certificates. The user creates a user *proxy*, a short-term certificate. The user acts as a CA for the proxy and signs the proxy using the user's certificate. The proxy contains a new pair of public and private keys. As the lifetime of the proxy is typically limited to a few hours, the private key of the proxy can be stored without encryption in a file accessible only by the user. The proxy and the user's certificate is sent to the remote site during the authentication phase. The proxy authenticates the user, the public key in the user's certificate validates the authenticity of the proxy. As only the proxy's private key is required during the authentication process, the user need not type any password. As the user delegates authority to the proxy, it is possible for the proxy to create new proxies. Figure 1 shows a chain of trust with a CA signing the certificate, the certificate signing the proxy etc.

Which users are granted access to a resource in the grid is determined by the *grid-mapfile* which is stored locally at the resource. On, e.g., a typical UNIX system, this file maps the DN of each authenticated user to a local UNIX account. More than one DN can be mapped to the same account.

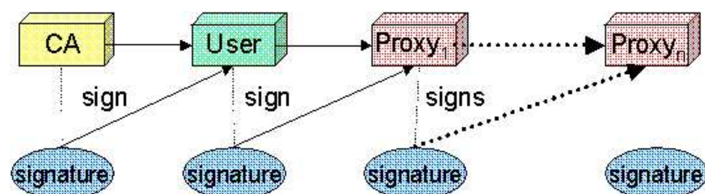

Figure 1: Chain of trust including CA, user and proxies.
Picture from www.globus.org.

GridFTP

The GridFTP protocol [40] is an extension to the well known FTP protocol. Extensions focus, among other things, on security and performance. GSI is used for authentication, and optionally for providing integrity and privacy. One important performance extension is parallel transfer, where multiple data channels can be used between server and client. A related technique is striped transfer, where data is concurrently transferred from more than one server to the client. Automatic negotiation of TCP buffer size and support for partial file transfer are other supported techniques useful for faster transfer of large files. The Globus toolkit 2 implementation of the GridFTP protocol includes APIs for adding plugins to a GridFTP server, enabling more complex services.

Information Services

Directories are used for storing and retrieving information. *Directory services* are directories accessible via a network protocol. The *Lightweight Directory Access Protocol*, LDAP, is a directory service defining an information model and a protocol for querying and manipulating information in the directory. LDAP also includes a hierarchical namespace that defines the organization of the information. LDAP *schemas* specify what *attributes* the directory should contain. Each attribute has one *type* and zero or more *values*. For each such schema, there exists an *information provider* that generates the values for each attribute. These information providers are typically shell scripts. The values provided can either be static, e.g., the number of CPUs, or dynamic, e.g., the number of jobs in the queue of a batch system. Both the protocol and the information model defined in LDAP are extensible. New schemas can be designed to advertise additional information in the directory. The definition of the LDAP protocol is found in RFC 1777 [48].

An information service for a grid environment must be able to handle a wide range of queries and many different types of resources. Furthermore, resource statuses and dynamic changes in VO membership must be handled as well as dynamic addition and deletion of information sources. LDAP, and other existing directory services such as X.500 [18] and *Universal Description, Discovery and Integration*, UDDI [42], do not fully meet these requirements. The design of the *Monitoring and Discovery Service*, MDS, is an attempt to fulfill these requirements [6].

MDS consists of two parts, the *Grid Resource Information Service*, GRIS, and the *Grid Index Information Services*, GIIS. Notable is that MDS relies

heavily on LDAP, both the GRIS and the GIIS are implemented as OpenLDAP [27] server backends. Each grid resource runs a GRIS server that advertises static and dynamic information about the resource. Examples of advertised information include CPU type, current load and available disk space. Incoming queries are dispatched to the appropriate information provider. For improved performance, provider results can be cached in a GRIS. The lifetime of cached information can be configured for each information provider. The GIIS makes it possible to construct an aggregate directory. GRIS servers register themselves to one or more GIIS servers. GIIS servers can either dispatch incoming resource information requests to the appropriate GRIS server or cache information from each GRIS server for faster client access. Scalability may become an issue with GIIS caching. Resource discovery is typically done by contacting a GIIS server and retrieving a list of available resources containing contacts to the GRIS server of each resource. All communication between a client and an MDS server is authenticated using the GSI infrastructure.

RSL

The *Resource Specification Language*, RSL [31], is a language for expressing user resource requests as well as configuration for the application. The basic unit in RSL is an attribute-value pair, which is called a *relation*. The syntax of the relation is *(attribute operator value)*. Examples of operators include '=', '!=', '>' and '<='. The value can either be a single value or a list of values. The syntax of a list is *(name_1 value_1)(name_2 value_2)* etc. A resource request can contain one or more relations. There are several methods to construct a job request that contains more than one relation. The most commonly used way to combine relations is the conjunct-request, `'&'`, which is equivalent to logical 'and'. The disjunct-request, `'|'` can be interpreted as logical 'or'. Users who request more than one resource should combine the requests by using the multi-request operator, `'+'`.

The RSL example in Figure 2 includes four relations combined with the conjunct-request. The attribute *executable* specifies the program to be run. The standard input and output files for the job are specified with the attributes *stdin* and *stdout*. By setting the value of the *count* attribute to 1, the user requests that one instance of `/bin/program` should be started on the resource.

```
&(executable = /bin/program)
 (count = 1)
 (stdin = data.in)
 (stdout = data.out)
```

Figure 2: RSL job request specifying executable, input and output files and number of processes.

In the example in Figure 3, the user requests to run the program `my_app`. The *min_memory* attribute specifies that the resource must have at least 4096 MB of memory available to the job. Similarly, *max_cpu_time* is used to determine the maximum execution time of the job. In the example, the user requests the job to run for 120 minutes. The *environment* attribute is an example of a list

of values. The user specifies values for two environment variables, HOME and DATADIR. These variables are then defined before the job starts to execute. Whereas the example in Figure 2 only included job configuration, the example in Figure 3 contains both configuration, e.g., *executable*, and requirements on the resource, e.g., *min_memory*.

```
&(executable = my_app)
 (min_memory > 4096) (max_cpu_time = 120)
 (environment = (HOME /home/tordsson)
                (DATADIR /home/tordsson/data))
```

Figure 3: RSL job request including a list of values and the greater-than operator.

The usage of the disjunct-request is illustrated in Figure 4. The user requests to run `my_app`. Resources suitable for this application can be of two types. The resources can either allow the job to use 4 processes with 2048 MB of memory available to each process or let the job consist of 8 processes, each using 1024 MB of memory.

```
&(executable = my_app)
 (|(&(min_memory = 2048)(count = 4))
   (&(min_memory = 1024)(count = 8)))
```

Figure 4: RSL job request using the disjunct-request.

GRAM

Resources that are connected to the grid runs the *Grid Resource Allocation Manager*, GRAM. GRAM enables the execution of remotely submitted applications on the resource. Any resource broker, coallocator, and other application can submit jobs using the GRAM API. Furthermore, the local resource manager of the resource can be of any type. Examples include batch systems, such as PBS [29] and LoadLeveler [19], or timesharing systems, e.g., the various dialects of UNIX.

A GRAM client is a program sending RSL job requests to the GRAM *gatekeeper* executing on the resource. The GRAM server consists of two programs, the previously mentioned gatekeeper and one or more *job managers*. The gatekeeper authenticates the incoming request using GSI certificates, and it dispatches the incoming request and starts a new job manager. The job manager parses the resource request and translates it into the language of the local resource manager. Next, the job manager starts a *job* on the resource, i.e., executes the program specified in the resource request. A unique identifier is assigned to the job and this identifier, the *job contact* is returned to the GRAM client enabling later communication between the job manager and the GRAM client. The most commonly used pattern of communication between the job manager and the GRAM client is asynchronous notifications of job state which are sent from the job manager to the client.

GASS

Applications that use traditional file I/O need to be adjusted before they can be executed in a grid environment. The *Globus Access to Secondary Storage*, GASS [4], simplifies porting of such applications to a grid environment. GASS consists of APIs for file access, cache management and data transfer, the latter used along with GRAM in the Globus job submission tools. It is not always possible (or desirable) to port the application to a grid environment. GASS can be used to transfer files required by the job, removing the need to port the application. As file staging imposes an overhead to the job, applications using only small parts of large files may benefit from the use of the remote I/O APIs of GASS as an alternative to staging the complete files.

2.3 Resource brokering

This section discusses the purpose of a resource broker and shows how a basic broker interacts with the components of a grid based on Globus toolkit version 2.

The general problem

The resource broker developed in this thesis differs from traditional batch system schedulers in that every user has their own copy of the broker. The aim of such a personal broker is to find, characterize, select, and possibly reserve and (co)allocate the resources best suited for each job submitted by the user. When selecting the resource, hardware constraints such as architecture, available memory and disk space as well as software constraints, e.g., operating system, software licenses and available runtime environments must be considered. To further complicate the situation, the broker must adhere to the local administration policies of the available resources.

The selected resources should be the ones that results in the earliest job completion. A broker should also try to optimize resource utilization and ensure fairness based on the agreed policies. This must all be accomplished despite the limited and possibly outdated information and the lack of global control. A more thorough discussion about the requirements for a grid resource broker is found in [5].

The Globus perspective

Figure 5 depicts a typical job submission scenario. The client could possibly be a user but is more likely a broker acting on behalf of the user. As a first step, the client contacts a GIIS server to retrieve information about the available resources. Next the client queries the GRIS server of each resource for resource information. The GRIS server of the resource invokes information providers that retrieves information about the resource by querying the local resource manager, typically a batch system. As the next step, the client matches the available resources with the user's RSL job request and selects the "best" resource available. Job submission to this resource is done using the GRAM protocol. The job request is sent to the gatekeeper of the resource, which dispatches the request and starts an instance of the requested job manager. The job manager parses the RSL request, translates it into the language of the local scheduler and sends

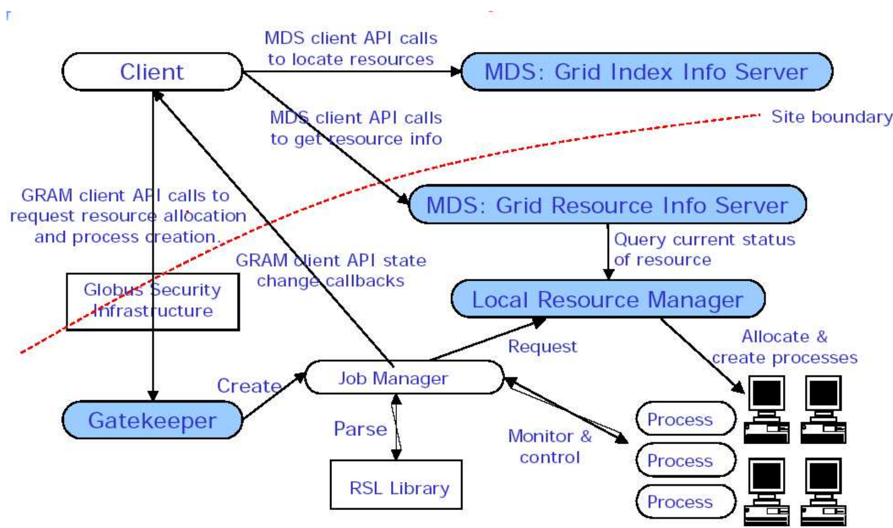

Figure 5: Job submission using the Globus toolkit 2.
Picture from www.globus.org.

a job request to the scheduler. The local resource manager (e.g., the batch system) allocates the requested number of processes. Upon successful process creation, the job manager generates a job contact which is transferred back to the client. Further communication with the job, e.g., retrieval of the job state or cancellation of the job, is done through the job manager.

All communication between the GRAM client and the resource is authenticated using the GSI infrastructure. Although not shown in the figure, the communication between the GIIS server and the client may be authenticated as well. The job submission scenario in Figure 5 includes no file staging. Transfer of job input files would occur before the job request is sent from the client. GridFTP or other protocols could be used for the file transfer.

3 The NorduGrid middleware

The development of the *Nordic Testbed for Wide Area Computing and Data Handling*, more commonly known as NorduGrid [26], started in May 2001. The project targets the development of a data and computational grid. The NorduGrid middleware is based on standard protocols and components such as OpenLDAP [27], OpenSSL [28] and Globus toolkit version 2 [13, 41]. The latter is not used in full, as Globus components such as GASS and GRAM are replaced by custom made components [8]. The NorduGrid middleware is deployed in the NorduGrid production environment, currently consisting of over 2000 CPUs distributed across more than 10 countries.

Table 1: Examples of tools in the NorduGrid user interface.

Program	Purpose
ngcat	Preview output from job.
ngclean	Delete files created by the job on the remote host.
ngget	Download files created by job.
ngstat	Retrieve status of job.

3.1 Components and Services

This section describes important features of the NorduGrid middleware, focusing on the non-Globus parts.

Security

The NorduGrid middleware uses the GSI, defined in the Globus toolkit 2, without modifications. The NorduGrid project operates a CA, issuing certificates to NorduGrid users.

XRSL

As the NorduGrid middleware does not use the Globus GRAM module for job submission, some modifications to the RSL are required. NorduGrid defines the *eXtended Resource Specification Language*, XRSL, a modified version of the RSL. XRSL is based on the same syntax as RSL and the RSL API defined in Globus toolkit version 2 is used within NorduGrid. The languages differs in terms of attributes, although many of the most commonly used attributes, e.g., executable and count, are the same.

User interface

The NorduGrid user interface consists of a set of command line tools that provides the functionality normally found in batch systems. Users can use these tool to, e.g., submit jobs, monitor the execution of their jobs, and cancel jobs. Other tools allow the user to, e.g., retrieve output from jobs, get a peak preview of job output and remove job output files from the remote resource. Communication with the remote resource is handled by a simple GridFTP client module.

Table 1 shows some of the command line tools in NorduGrid. When invoking any of the tools, identifiers for one or more jobs must be passed to the program. Alternatively, the flag `-a` can be passed, to request that the selected action is performed on all jobs submitted by the user. Job identifiers for all jobs belonging to a user are stored locally and can also be retrieved from the information system.

Before any of the job management tools in Table 1 can be used, the jobs must be submitted to the grid. This is done using the job submission tool, *ngsub*, which includes a basic broker personal to each user. One or more XRSL job descriptions must be passed upon invoking ngsub. These descriptions are entered directly on the command line or read from files. Other options include specifying which GIIS server(s) to contact and what resource(s) to submit the

job requests to. The broker developed in this thesis is based on the ngsub tool, which contains a simple broker that bases resource selection on current load. A complete redesign has been done, and as a result, the developed broker and the ngsub tool show very few similarities.

GridFTP server

Each resource in the grid runs one instance of the NorduGrid GridFTP server. Upon job submission, the user invokes the broker which uploads an XRSL job submission request to the GridFTP server.

Implemented using the Globus GridFTP protocols, the GridFTP server makes use of the Globus toolkit 2 GridFTP plugin API. The GridFTP server contains three plugins. The fileplugin enables grid access to the local file system of the resource and the gaclplugin is used to manage grid access control lists. The most important plugin is the jobplugin, which enables job management. Three FTP commands are used for job management purposes. When submitting jobs, the broker sends the command “CWD” with the argument “new” to the GridFTP server. Upon receiving this call, the job plugin of the GridFTP server creates a new *session directory*. There exists one session directory for each grid job. Files used by the job are stored in this directory along with some additional information about the job. Temporary files created by the job should also be placed in the session directory. The name of the directory is unique and also serves as an identifier for the job.

Cancellation of a job from the user interface is done by sending the command “DELE” with the session directory of the job as argument. The similarly used command “RMD” also takes a session directory as argument. The GridFTP server removes the session directory and all files included in the directory when receiving this command. The GridFTP server with its job plugin conceptually replaces the GASS and the GRAM gatekeeper from Globus toolkit 2.

Grid manager

Each grid resource runs a grid manager in addition to the GridFTP server. The grid manager is responsible for managing each grid job through the various phases involved in the execution of the job. The task of the grid manager is simplified by the existence of the *control directory*. The grid manager stores user proxies, the status of each job and batch system specific information about the job in the control directory.

The grid manager periodically polls the session directories looking for new jobs uploaded from the user interface to the GridFTP server. For each new job found, the following steps are performed:

1. The XRSL job description is analyzed by the grid manager, which looks for, e.g., job input files.

2. Any input files required by the job is transferred to the resource using the GridFTP protocol.

3. The XRSL job description is translated into the language of the local scheduler, and the job is submitted to the local batch system.

4. Upon job completion, the grid manager stages output files to the locations specified in the job description.

Some additional actions are also performed, see the grid manager manual [20] for details. For each action taken by the grid manager, the job is defined to be in a certain state. All possible states and state transitions are shown in Figure 6 and described below.

- A job submitted to the resource has the *ACCEPTED* state before the job description is analyzed. If the Grid manager fails to analyze the description or the user requests to cancel the job, the state changes to *FINISHING*.

- After analyzing the job description, the grid manager downloads the files requested by the job. For the duration of these transfers, the job is in the *PREPARING* state. Upon file transfer failure or user cancellation requests, the job state changes to FINISHING.

- When the preparation phase is over, the grid manager submits the job to the *Local Resource Management System*, LRMS, which typically is the local batch system. During this submission process, the job is in the *SUBMITTING* state. As for the previous states, failures to submit the job or user cancellation requests changes the state of the job to FINISHING.

- After submitting the job to the LRMS, the grid manager awaits the completion of the job. The state of the job is *INLRMS* during this process. The information system (but not the grid manager) defines two substates for the INLRMS state, namely *INLRMS: Q* and *INLRMS: R* denoting a queued job and a running job, respectively.

- If the user requests to cancel a job in the INLRMS state, the state of the job changes to the intermediate state *CANCELING*. Once the grid manager's request to cancel the job in the LRMS is completed, the job leaves the CANCELING state and becomes FINISHING.

- When the job has finished to execute on the LRMS, the state of the job is *FINISHING*. The state remains FINISHING until the grid manager has transferred all job output files to the locations specified in the XRSL job description. Users can download the complete output files from jobs in the FINISHING state using the ngget tool.

- Once the job state changed from FINISHING to *FINISHED*, the grid manager is done with the job. All output files remains in the session directory and can be downloaded by the user. There is no additional state for separating failed jobs from successful ones. The user interface tools can however detect job failure and displays an error message to the user if a job should fail. After a configurable period of time (default 1 week), the job output files are removed and the job state changes from FINISHED to *DELETED*.

- No output files are stored for jobs in the DELETED state. Only a minimal set of information about the job is kept.

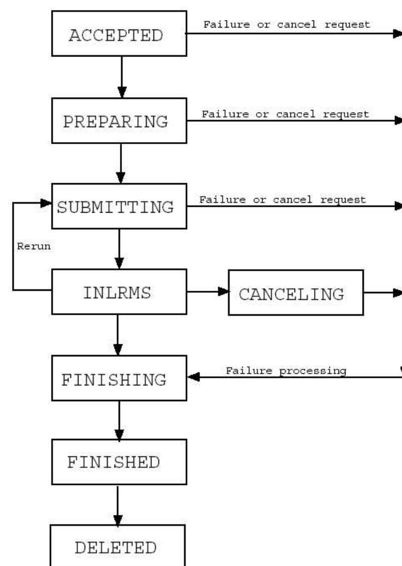

Figure 6: States and state transitions for a NorduGrid job
Picture from www.nordugrid.org.

The grid manager can be configured to invoke system specific *plugin scripts* every time a state transition is about to occur. These scripts determine if the job should change into the new state or fail. Such scripts form a tool for enforcing policies regarding billing, reservations etc. The grid manager basically performs the same tasks as the Globus GRAM job manager, and the job manager is not used in the NorduGrid middleware.

3.2 Brokering within NorduGrid

The task of a general resource broker is to identify, characterize, evaluate, select, allocate and coordinate resources with different characteristics, while at the same time handling constraints of varying nature. Within the NorduGrid environment, the broker's task is simplified by restrictions from the general scenario. Two types of resources are available, computational clusters and storage elements. Of these, the broker only handles the clusters. The brokering problem is further simplified as all clusters are based on the same architecture and run the same operating system, even though this homogeneity is not likely to last in the long term. Jobs using more than one CPU can be submitted, there is however no support for cross-cluster parallelism.

Figure 7 shows the interaction between the components in NorduGrid. The user interface, or more specifically, the ngsub tool, contacts one or more GIIS servers to retrieve a list of available resources. Each resource (cluster) is registered periodically with one or more GIIS servers. The GRIS server of each cluster generates information about the state of the resource. The Information providers of the GRIS server collects static machine information as well as dy-

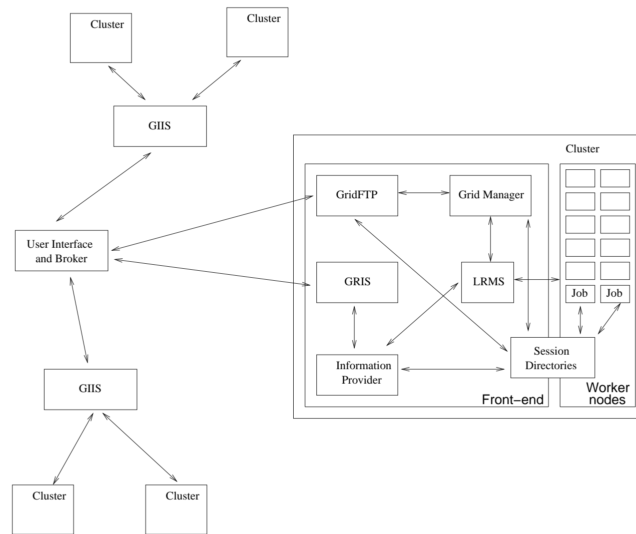

Figure 7: Interaction between NorduGrid components.
Figure based on image from www.nordugrid.org.

namic information, e.g., load from the LRMS and job states from the session directories. Note that information about submitted jobs only can be retrieved from the information system. (In a pure Globus environment, the state of a job can be obtained either from the GRAM job manager or from the MDS). Once a suitable resource is found for the job, the GridFTP server of this resource is contacted. After mutual authentication of client and server, a job request message including the XRSL job description is uploaded. This request is uploaded to a new session directory. The grid manager analyzes the job description, translates it into the language of the LRMS and requests a job from the LRMS. Next, the LRMS allocates the requested number of worker nodes and the job starts to execute. Further job control such as job cancellation or output retrieval is done by communicating with the GridFTP server. Job management that can be performed by using the session directories only is handled by the GridFTP server. If the usage of LRMS tools is required to manage the job, the GridFTP server passes the incoming request to the grid manager, which invokes the appropriate LRMS tools. Currently, the only supported LRMS is the Portable Batch System, PBS [29], although support for Condor [39] is being developed.

4 Measuring brokering performance

The purpose of this thesis is to construct improved algorithms for resource brokering. When doing this, a metric measuring the quality of the work done by the broker is needed. As a grid environment includes many actors, different metrics are possible. Each user within NorduGrid runs a personal broker used by only that user. With this in mind, the aim of the broker should be to enable the best possible utilization of the grid for the user. A good objective for such a broker is to try to minimize the time required for each user-submitted job

to finish, that is, the time elapsed between job submission and the time when the output files are completely stored where requested. We refer to this time as *Total Time To Delivery*, TTTD. A closer inspection of the job execution process shows that the TTTD includes time for executable and input file transfer, batch queue waiting, application execution and output file transfer.

Input file transfer. Before the job can start execution on the resource, the executable and the job input files must most likely be transferred. These files and their respective locations are specified in the XRSL job description. The files can be stored on different machines, and multiple copies of the same file can be replicated across the grid, possibly enabling faster access to the required files.

Batch queue waiting. Once all files are located on the resource, the application can start to execute. As clusters connected to the grid typically are operated by a batch system, the job may have to wait for other jobs to complete before it can start. We refer to this delay as batch queue waiting. If the selected grid resource does not run a batch system, which typically is the case for shared memory machines, workstations and personal computers, no batch queue waiting is required.

Program execution. The time required to execute the application on the resource varies with the characteristics of the job, the performance of the resource and with how apt the resource is for executing the job. For efficient usage of the available grid resources, the time required for program execution should be large compared to the other parts of the TTTD.

Output file transfer. When the job is completed, it has most likely produced some output data. The XRSL job description specifies where each output file should be stored. Possible locations include the resource executing the job, the machine from which the job was submitted, or some other resource in the grid. To store the job output files somewhere else than on the cluster that executed the job imposes an overhead, as the output files must be transferred over the network.

4.1 Scenarios

When considering how the different subtasks involved in the execution of a job on the grid affect performance, some possible scenarios arise. If the job input files are large, the shortest TTTD may be obtained by moving the executable program to a cluster where the input files are located and execute the job on this cluster. Analogously, if the input data is small but the size of the executable is large, moving the data to the machine which stores the executable program may shorten the TTTD. A third option is to move both the input data and the executable from their respective locations to a third machine. This third machine being faster or less loaded than the ones storing job input and executable. Users knowing in advance that the job output will be large benefit if their jobs execute on the cluster where the output files will be stored.

Typical NorduGrid usage

The NorduGrid production environment consists of resources owned and administrated across several domains, resource providers may have different reasons for grid-enabling the resources. Users of the NorduGrid typically use the NorduGrid resources because they have a need for computational power that can not be satisfied using locally available resources. Typical jobs submitted involves lengthy computations and requires the transfer of large input- and output files. The number of jobs submitted by each user is large. Users tend to submit their jobs in batches containing hundreds or even thousands of jobs.

5 Algorithms and implementations

This section describes the general design of the broker, the algorithms and their implementations. Also included in this section is an overview of the program flow.

5.1 Techniques used to estimate the TTTD

Techniques used to estimate each of the four parts included in the TTTD are described in this section. Other possible solutions than the ones implemented are discussed briefly. As local resource owners retain control over grid-connected resources, decisions must be based on information, not on control. The techniques used all adhere to this policy, some of the alternative solutions do not.

Predicting network transfer time

Network transfers are required both for uploading the executable and job input files to the resource and for storing job output where specified. The broker predicts the network performance by using a monitoring approach. With an accurate prediction of the bandwidth available between the computational resource and the hosts storing the input / output files, the broker can approximate the time required for file transfers. For each cluster considered, the broker calculates the time required to transfer the files between the respective locations and the cluster.

An alternative to predicting the network bandwidth would be to reserve network capacity between the cluster and the host(s) storing the required file(s). While this may seem to ensure fast file transfers, it is hard to accomplish. Existing mechanisms for guaranteed bandwidth transfers include intserv and diffserv [3]. The current deployment of those technologies and similar ones is limited to experimental networks. It seems like the Internet of today, running IPv4, is not apt for transfers with bandwidth guarantees.

Waiting time prediction

To estimate the amount of time a job has to wait in a batch queue before the job starts to execute is a difficult task. The algorithms used in the local batch system schedulers are complex and many parameters affect the scheduling policy, making it difficult to predict when a queued job starts to execute. The

broker circumvents this obstacle by using advance reservations. With an advance reservation, the job has a guaranteed start time, which is more accurate than the best prediction. Advance reservations are also useful for coallocation purposes. One problem, however, is that not all batch systems support advance reservations. A survey of the reservation capability of different batch systems can be found in [24].

An alternative to advance reservations is to use the job start time estimation tool that is available in some schedulers, e.g., Maui [36]. However, such tools only provide an estimate, not a guaranteed start time.

Benchmark-based execution time prediction

To predict the execution time in a grid environment is hard due to performance differences between the resources. The fact that the relative performance of resources may vary for different types of applications complicates this issue further. Even though the grid information system advertises an abundance of resource hardware information, e.g., available memory and CPU clock frequency, an execution time can not be accurately predicted from this information.

In order to address this problem, the broker uses a benchmark-based approach for determining the execution time. A user submitting a job specifies a set of benchmarks relevant for the application in the XRSL job description. This benchmark set includes results for the benchmarks on a reference machine and the application's estimated execution time on that machine. These user-estimates are complemented with benchmark results advertised in the MDS by the clusters connected to the grid. The broker can calculate an estimate of the application's execution time using the benchmark information retrieved from the clusters and the benchmark information in the job description. This approach overcomes both the problem with the variation in cluster performance and that the relative performance of applications differs between different types of clusters.

An accurate prediction of the execution time is critical in a heterogenous grid environment. To request a too short execution time results in preemption of the job and job failure. Conversely, the job will have to wait for a long time in the batch queue if a too long execution time is requested.

Queue adaptation

As changes occur all the time on the grid, information is necessarily old. The load on network connections may rise or fall rapidly. Systems with long batch queues may become available faster than predicted if a job using many CPUs completes earlier than estimated. Existing resources may become unavailable due to crashes, and new resources may appear on the grid. Moreover, grid-access to a resource may be granted only during certain hours of the day. In order to take this dynamic resource availability into account, the broker adapts to sudden changes by continuing to search for resources also after the initial job submission is done. If another cluster is found that is likely to result in a shorter TTTD, or actually, result in an earlier job completion than the one the job is currently submitted to, it may be beneficial to move the job.

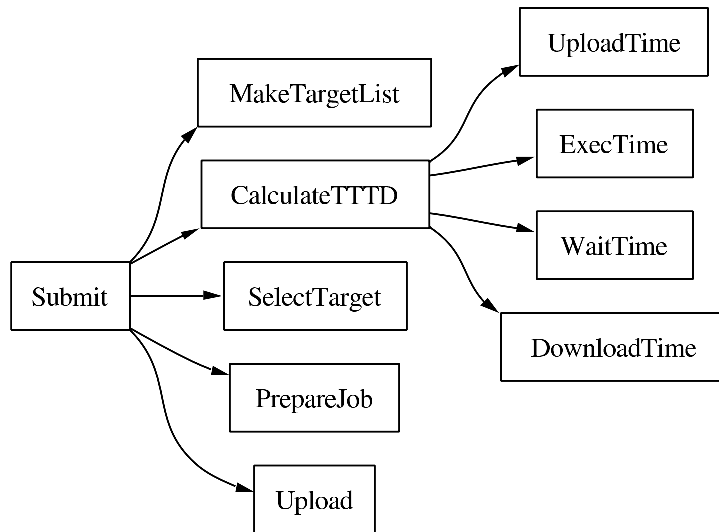

Figure 8: Overview of program flow for job submission.

5.2 Job submission

Figure 8 describes the program flow for job submission. When submitting a job, the broker first identifies possible targets. The term *target* denotes a queue on a cluster. Clusters may have several queues, e.g., each queue dedicated for jobs with certain execution time ranges. Next, the predicted TTTD is calculated for each target, the calculation is based on resource characteristics and the XRSL job description. The best target, i.e., the one with the lowest predicted TTTD, is selected for submission. Next, the broker prepares the XRSL job description for submission by adding some additional information, e.g., the benchmark-predicted execution time. In the final step, the job description is uploaded to the selected cluster.

The algorithm described in Figure 9 shows the steps taken by the broker during job submission. The input to the algorithm is one or more XRSL job requests. These describe the jobs in terms of executable, input and output files etc. The requests may also contain resource requirements such as operating system, disk space and architecture. Optionally, the user includes in the description, job characteristics that may improve the resource selection, e.g., relevant benchmarks and estimations of job output size. Examples of job descriptions are found in Section 6.

In Step 1, all the user's XRSL job requests are organized in a list. Requests containing the multi-request operator, '+', are separated into individual requests. One or more GIIS servers are contacted in Step 2 and a list of available resources is retrieved. Note that no filtering is performed. For a world-wide deployment of the NorduGrid middleware, millions of clusters may be found in Step 2. Hence this step of the current job submission algorithms must be modified if the broker should be used in such environments. In Step 3, the broker queries the GRIS server of each cluster, obtaining information about the software and hardware characteristics of the cluster as well the cluster's current load.

Input: User's XRSL job request(s).
Action: Find, select and allocate the resources most appropriate for each job.
Output: none.

1. Process each XRSL job request and merge all requests into a list.
2. Contact one or more GIIS servers to obtain a list of available clusters.
3. Query the GRIS of each cluster for hardware information as well as the current state of that cluster in terms of queues, jobs and load.
4. For each job:
   (a) Select the cluster to which the job will be submitted:
      i. Filter out clusters that do not fulfill the requirements on memory, disk space, architecture etc, and clusters that the user is not authorized to use.
      ii. Determine the TTTD for each cluster, this includes making reservations if so requested by the user.
      iii. Select the cluster with the shortest predicted TTTD.

   (b) Submit the job to the selected cluster.
   (c) Release any reservations made to non-selected clusters.

Figure 9: High-level job submission algorithm.

The brokering process is performed in Step 4, including resource evaluation and resource selection.

First, the list of all clusters is filtered for those not desirable for the job. This includes clusters failing to fulfill hardware requirements such as architecture, disk space and memory, or software requirements, e.g., operating system and installed software. Furthermore, the cluster is rejected if it does not accept new jobs due to heavy load, or if the user is not authorized to use the cluster. Next, the broker evaluates each remaining cluster by computing the TTTD associated with job execution on the cluster. To predict the TTTD includes making advance reservations if the user requested that to be done. After the prediction step is completed, the broker submits the job to the cluster with the lowest TTTD. Finally, if reservations were made on non-selected clusters, these reservations are released.

Errors occurring in the job submission process can be of two types. First, there are program errors due to incorrect user input. If such errors occur, the job submission process is aborted and the broker terminates. Errors may also occur due to imperfect behavior of the components in the grid and may cause failure in job submission to a certain cluster. Upon these types of errors, the broker retries using another cluster. If further failures should occur, the broker retries as long as there are clusters left to be considered.

The algorithm in Figure 10 gives a more detailed description of how the TTTD is predicted. In Step 1, the upload time is predicted using the algorithm in Section 5.3. The execution time is predicted in Step 2. The method used to determine the execution time depends on the information provided by the user

in the job description and on the information advertised by the GRIS server of the cluster. The following methods are tried in decreasing order of accuracy until the execution time can be determined.

The first three methods are based on information in the user's XRSL job description. The preferred method to use is benchmark-based execution time prediction, which only can be used if the user includes benchmark information in the job description. More details about this method is found in Section 5.5. If the benchmark-based method can not be used, the broker scans the job description for a wallclock time to use as execution time. If no wallclock time is found, the broker instead scans the job description for a normalized execution time. This normalized execution time is linearly scaled with the CPU clock frequency of the cluster, the norm is 1 GHz. If no information that can be used to predict the execution time is included in the job description, the broker uses default values. If a default execution time is advertised by the GRIS server of the resource, this value is used as execution time. If no such information is available, the broker assumes a predefined execution time. Users can configure the predefined execution time in a configuration file. An example of a configuration file for the broker is found in Appendix B.

In Step 3, the queue waiting time is predicted. If the cluster supports advance reservations and the user requests that reservations should be used, the broker makes a reservation and determines the queuing time from the reservation start time and the time required to stage input files. The procedure for requesting reservations is described in Section 5.4. If the cluster does not support advance reservations, if no reservation was requested from the user, or if the reservation request failed, the waiting time is predicted from the current load of the cluster. Parameters used in this prediction are the number of jobs in the queue and the number of CPUs of the cluster. Furthermore, assumptions are made about how many CPUs each job uses and about the duration of a job. Using this model, the broker can predict when all jobs are finished by defining the queue waiting time as the completion time of the last job in the queue.

The broker predicts the time required for transfer of job output files in Step 4. This prediction is only possible if the user specifies estimates of the size of the job output in the XRSL job description. If no job output size predictions is included in the job description, the download time is set to zero. In the 5th and final step, all predictions from the previous steps are added together into the TTTD.

5.3 Network file transfers

Before execution can take place on a cluster, any remote input files must be staged to the cluster.

For the calculation of the TTTD, the broker must predict the time needed for transfer of the files required by the job. In order to do this, the broker must have an estimate of the expected available bandwidth between the resources involved in the job submission. For this purpose, the Network Weather Service, NWS [47], is used. NWS combines bandwidth measurements with statistical methods to make short-term forecasts of the available bandwidth. There exists many other tools for measuring bandwidth, e.g., netperf [25] and ttcp [43], but these do not generate forecasts.

Input: Descriptions of a job and a resource.
Action: Predict the TTTD for the job if executed on the resource.
Output: The predicted TTTD for the job.

1. Predict upload time using the algorithm in Figure 12.
2. Predict execution time, try the following methods in the order listed below until success:
   (a) Benchmark-scaled execution time (Section 5.5).
   (b) User-specified execution time (wall clock time).
   (c) User-specified execution time (CPU frequency scaled time).
   (d) Default execution time for jobs in queue on resource.
   (e) Broker's default execution time.
3. Predict waiting time:
   (a) If reservations are supported by the resource and requested by the user, make a reservation and calculate the waiting time using the upload time and the reservation start time.
   (b) If the reservation request failed, if the resource does not support reservations or if the user chooses not to use reservations, predict the waiting time from the current load of the cluster.
4. Predict download time:
   (a) If the user has provided predictions of job output size, use the algorithm in Figure 12 to predict the download time.
   (b) Else, let the download time be 0.
5. Add the times predicted in the first four steps into the TTTD.

Figure 10: Algorithm for determining TTTD.

Measuring and predicting bandwidth

A setup of the NWS includes at least three processes; a *name server* to which hosts register, a *memory* which stores measurements and generates forecasts, and finally, a *sensor* that performs the measurements. Quantities that can be measured include CPU load, memory availability, network latency and network bandwidth. Typically, one machine runs a name server and a memory. All machines to be monitored, including the one running the name server and the memory, run sensors. These sensors register with the name server and store their measurements on the memory host. An *activity* defines periodic measurements of a quantity with correlated prediction generation. One such activity is called a *clique*, and measures latency and bandwidth between all pairs of sensors in the network. The term clique refers to a complete graph, an appropriate name for the activity as there exists network connections between any pair of sensors.

The algorithm in Figure 11 is implemented by the information provider which is invoked every time the GRIS server receives a request for LDAP information about bandwidths. In Step 1, the name server is contacted and a list of all

Input: Address to NWS name server.
Action: Enter bandwidth information in LDAP directory for resource C.
Output: Bandwidth forecasts of networks connecting this cluster to others.

1. Contact the name server and get a list of all registered clusters.
2. If the cluster C does not run a sensor, terminate the algorithm.
3. For every other cluster M that has a sensor registered at the name server:
   (a) Get the predicted bandwidth B between C and M and add (M, B) as an LDAP attribute to be advertised by C.

Figure 11: Algorithm for advertising bandwidth predictions.

clusters running an NWS sensor is retrieved. If the cluster does not run a sensor itself, no bandwidth predictions can be generated. The existence of a local sensor is controlled in Step 2. In Step 3, the bandwidth from the cluster to all other clusters are predicted, by invoking an NWS tool extracting the latest bandwidth forecasts. The bandwidth, along with the name of the other cluster is advertised in the information system.

The algorithm fails if the NWS name server is not running, or if the cluster does not run a sensor. In these cases, query replies from the GRIS server does not include any bandwidth predictions. This does not affect the operation of the cluster, but it is less likely to be selected by the broker for jobs that require file transfers.

Predicting file transfer times

The times required for input file transfer and output file transfer are both determined using the algorithm in Figure 12. Download of the output files from a job is similar to upload of the input except in one important aspect. The sizes of the output files are not known in advance as these files are created during the execution of the job. Grid users often develop their applications themselves and have some sense of the amounts of output data their grid job generates. An extension to the XRSL job description language allows, but does not require, the user to provide estimates of the sizes of the various output files. Using these file size estimates along with the specified locations of the output files, the broker estimates the transfer time.

The input to the algorithm is the locations of the files required by the job and information about the cluster. The file locations are specified in the XRSL job description. In Step 1, the broker gathers all information available about the network connections by querying the GRIS server of the resource. The broker constructs in Step 2 a set where the time required for the network transfers will be stored. In Step 3, a table with entries for all hosts that stores (or will store) the files is defined. Each entry in the table contains the total size of the files stored at the respective host.

In Step 4, the sizes of the files stored at the same host are added together. As no transfer is required for locally stored files, it is first assured that the files are not stored locally. A file is stored locally if its URL contains the prefix `file://`.

Input: A list of URLs for the files to transfer and description of cluster C. **Action:** Predict time to transfer the files between their respective locations and C. **Output:** The predicted transfer time.

1. Query the GRIS server of cluster C and store information about available network bandwidth between C and all other resources in the grid.
2. Let `transfer_times` be a set of all transfer time predictions. Initiate by setting `transfer_times` $\leftarrow \emptyset$.
3. Let `hosts` be a table of all hosts. Each entry in the table stores the address of the host and total size of the files to be transferred from the host to the cluster. Set `hosts` $\leftarrow \emptyset$.
4. For each input file:
   (a) If the file is stored locally at C, ignore it and continue with the next file.
   (b) Get the size of the file. If the file size could not be determined, ignore the file and continue with the next one.
   (c) Determine, by inspecting the file URL, on which host H the file is stored.
   (d) If there is no entry for H, in `hosts`, add an entry for H with the size of the file as value. If there already exists an entry for H, increase the value in this entry with the size of the file.
5. For each host H in `hosts`:
   (a) Get the predicted bandwidth between H and C. If this value can not be found in the information retrieved from H's GRIS, assume a minimum bandwidth.
   (b) Determine how long the transfer time T will be for the files stored at H and add T to `transfer_times`.
6. Return `max{transfer_times}`.

Figure 12: Predicting time required for file transfer to or from a given cluster.

A file is also local if it is stored at the host `localhost` or at the same host as it should be transferred to. These tests take care of the cases when the user specifies the URLs to local files using prefixes such as `http://` or `gsiftp://`. Next, the size of the file is obtained. For input files, the resource storing the file is queried for the size of the file. For output files, the XRSL job description is scanned for a user-prediction of the size of the file. A file is ignored if the size can not be determined. Next, the name of the host which stores the file is extracted from the file URL. If there exists an entry for this host in the table of all hosts, the size of all files stored at the host in increased with the size of the file. Otherwise, an entry for the host is added to the table and the value of the new entry is set to the size of the file.

In Step 5, the broker predicts the time required to transfer files between their location and the cluster. First, a bandwidth forecast is obtained from the information retrieved from the GRIS of the cluster storing the file(s). If the cluster does not advertise any bandwidth information, the broker assumes a minimum bandwidth, configurable by the user. The sizes of the files and the bandwidth forecast are used to calculate the predicted transfer time and this time is added to the set of all transfer times. The largest value in the set of all predicted transfer times is returned in Step 6.

When configuring the minimum bandwidth, users must consider that it can be fatal to underestimate the input file transfer time. All files required by the job must be present when the job starts to execute, otherwise it may fail. Overestimating the transfer time results in some undue waiting, but it does not cause job failure.

Files simultaneously transferred between the two hosts share the available bandwidth. For this reason, transfer times are calculated on a per-host basis, instead of on a per-file basis. As files transferred to or from different clusters can utilize the full available bandwidth, the maximum and not the sum of the transfer times is used in the prediction. This model assumes that the network connecting the cluster to the rest of the grid is not a bottleneck. Furthermore, it is assumed that the node in the cluster that receives the incoming data can receive all parallel data streams at a full rate. This imposes requirements on the cluster's local storage system and on the network card of the node. A more sophisticated algorithm than the implemented one should take these possible bottlenecks into account. Using the simplified model, it is reasonable to exclude local files and files with unknown sizes from the transfer time prediction. An input file with unknown size may very well be the one requiring the longest transfer time. This file may cause job failure as a result of the underestimate of the transfer time. However, a required file with unknown size is probably broken or unavailable. This will cause problems and most likely failure of the job regardless of the correctness of the prediction of the time required to transfer the file.

5.4 Advance reservations

This section describes the implementation of the advance reservations capability, including how reservations are made and released. The section also includes a description of the additional steps required in the job submission process when submitting a job for which an advance reservation has previously been created. Finally, some further details of the reservation protocol are discussed.

Input: XRSL Job description, requested execution time and requested start time.
Action: Make an advance reservation.
Output: Reservation identifier and start time of reservation.

1. The broker constructs the reservation request message which includes the number of CPUs required for the job, the requested length and start time of the reservation, and optionally, an account to be charged for the job.
2. The broker uploads the reservation request to the GridFTP server.
3. The GridFTP server adds the local identity of the user to the reservation request.
4. A script for making a reservation in the LRMS is invoked by the GridFTP server.
5. If no reservation could be created in the LRMS, the GridFTP server replies with a message telling the broker that the reservation request failed, and the algorithm terminates.
6. The GridFTP server creates a file which associates the user and the reservation.
7. An identifier for the reservation and the reservation start time is returned to the broker by the GridFTP server.

Figure 13: Algorithm for making a reservation.

Making reservations

An outline of the algorithm used for making a reservation is shown in Figure 13. Input to the algorithm is the XRSL job description and the predictions for job start time and execution time. In Step 1, the broker constructs the reservation request message including the number of CPUs, the job start time and execution time, and, if specified in the job description, an account to be charged for the job. The number of CPUs required is read from the job description. If no number is included in the job description, the broker assumes that one CPU is required. The job start time and execution times were determined when predicting the TTTD. A little margin is added to the start time to compensate for overhead in the local scheduling algorithm. Some batch systems, e.g., PBS, requires that the job is present in the batch queue when the reservation starts. In Step 2, the broker uploads the reservation request to the GridFTP server of the cluster. In Step 3, the GridFTP server adds the local identity of the user to the reservation request. The local identity of the user is determined by the grid-mapfile of the cluster. This step can therefore not be done by the broker. The GridFTP server invokes a script creating a reservation in the LRMS in Step 4. This script uses tools specific to each LRMS, and the script must be adapted if some other LRMS than the currently supported PBS should be used. Step 5 is only performed if the local reservation script fails to create a reservation. If this occurs, a message indicating failure is returned to the broker and the algorithm

terminates. In Step 6, which only occurs in the case of a successful reservation, the GridFTP server stores the user's proxy in a file named to associate the user and the reservation. This file is named `reservation.res_id.proxy`, where `res_id` is an identifier for the reservation. Step 6 is performed to ensure that no user tries to spoof a reservation identifier and submit jobs to another user's reservation. The GridFTP server returns the reservation identifier and start time to the broker in Step 7. The returned reserved start time may be later, but can never be earlier, than the one requested by the broker.

Submitting and executing a job with a reservation

Before submitting a job utilizing an earlier reservation request to a cluster, the broker adds the reservation identifier to the XRSL job description. The broker submits the job by uploading the job description to the GridFTP server of the selected cluster. The job description along with files used by the job are stored in the session directory. The control directory stores files controlling job execution, one of these files is the proxy used when submitting the job. This proxy file is named `job.job_id.proxy`, where job_id an unique identifier for the job. If the submission to the preferred cluster is successful, reservations made to other clusters are canceled. When releasing reservations, the broker sends a release message including the reservation identifier to the GridFTP server of the cluster.

The GridFTP server uses two plugin scripts when it manages submission of a job for which a reservation has been created. The first script is invoked when the state of the job changes to `SUBMITTING`. This script is used to confirm the validity and authenticity of the reservation. The script scans the job description, and if no reservation identifier is found, the job is submitted to the LRMS without a reservation. If a reservation identifier is found, the script controls that the identifier is valid. This is done by ensuring that a file named `reservation.res_id.proxy`, where `res_id` matches the reservation identifier, exists in the control directory. If no such proxy file can be found, the job submission fails. If the identifier of an existing reservation is specified in the job description, the script ensures that the user submitting the job is the same one that made the reservation. This is done by comparing the DN of the proxy used for making the reservation (the file `reservation.res_id.proxy`) with the DN of the proxy used for submitting the job (the file `job.job_id.proxy`). Unless these files identify the same user, the job submission fails. The actual comparison is done using OpenSSL tools.

A second plugin script is invoked when the job moves to the `FINISHING` state, that is, the job finished executing in the LRMS. This script reads the job description to determine if the job was submitted with a reservation. If the job used a reservation, the file `reservation.res_id.proxy` is removed and the reservation in the LRMS is released. This ensures that the user making the reservation can not submit more than the requested job to the same reservation. The second script will also be invoked if a job submitted with a reservation is either canceled by the user or fails for some of the reasons discussed in Section 3.1. When doing this, it is guaranteed that no reservations remains for canceled or failed jobs.

The two grid manager plugin scripts depend on the LRMS in only one aspect, the command called to release the local reservation.

Reservation protocol

The protocol used for managing advance reservations supports two operations, requesting and releasing a reservation. As all communication between the broker and the cluster is performed through the GridFTP server, reservation commands are expressed as GridFTP commands. The reservation protocol extends the GridFTP protocol by defining two new messages, "GERE" for requesting a reservation and "RERE" for releasing a reservation.

Table 2: Overview of "GERE" command.

Command:	`GERE`	GEt REservation
Required arguments:	nodes	Number of CPUs requested
	duration	Length of reservation
	start time	Start time of reservation
Optional arguments:	project	Account to be charged for reservation
Server replies:	res_id starttime	Request successful
	`failed` nextstart	Request failed, reservation possible nextstart
	`failed 0`	Request failed, reservation never possible
Server error messages:		Wrong number of arguments

Table 2 gives an overview of the reservation protocol. The broker sends the command "GERE" to the GridFTP server when requesting a reservation. Three arguments are required; the number of CPUs requested, the length of the reservation, and the start time of the reservation. One argument is optional, the account to be charged for the reservation. If a reservation has been created, the GridFTP server replies with the reservation identifier and the start time. Two alternatives exist for failed reservation requests. Instead of the reservation identifier and the reservation start time returned for successful reservations, the GridFTP server replies with "failed" and an integer. This integer can either be 0, meaning that no reservation is possible at all, or a nonzero number indicating the earliest possible start time of a reservation. The latter message would allow the broker to send a new reservation request with the later start time. This functionality is however not implemented. The length of the reservation is specified in seconds. The format of the start time is the time in seconds elapsed since midnight, UTC, January 1, 1970. The usage of these formats simplify both parsing and interaction with batch system tools.

If the wrong number of arguments are sent by the broker, the GridFTP server will return an error message indicating this. The reservation procedure fails if the reservation script did not execute correctly, or if the proxy file for some reason can not be stored on the cluster. If any of these errors should occur, the GridFTP server sends `failed 0` to the broker. The broker can cancel a reservation made to a cluster by sending the "RERE" command to that cluster. An overview of this command is shown in Table 3. The command requires one argument, the identifier of the reservation the broker wants to cancel. The GridFTP server replies with a message indicating that the reservation was released. Two error messages can be sent from the server to the broker, one stating that the number of arguments in the "RERE" message is wrong, the

Table 3: Overview of "RERE" command.

Command:	`RERE`	RElease REservation
Required arguments:	res_id	Reservation identifier
Optional arguments:	-	
Server replies:		Reservation released
Server error messages:		Wrong number of arguments No such reservation

other informing the broker that no reservation with the specified identifier exists on the cluster. As the broker can not handle any of these errors, it ignores the reply from the GridFTP server.

5.5 Benchmark based execution time prediction

Figure 14 describes the algorithm for predicting the execution time. The algorithm is invoked when the broker predicts the TTTD, see Step 2a of the algorithm in Figure 10. The algorithm in Figure 14 takes three inputs. The first input is the set of benchmarks the user specifies in the job description. A specification of a benchmark includes the name of the benchmark, the result of the benchmark on a reference machine and a prediction of the execution time for the job on that reference machine. The second input is the set of all benchmarks advertised in the MDS by the cluster. This set contains the names of the benchmarks and their results on the cluster. The third and final input to the algorithm is a penalty factor. The factor is used when the broker compensates for any user-specified benchmarks not advertised by the cluster.

Step 1 of the algorithm initiates a collection of execution time predictions that will be made from user-specified benchmarks. Multiple instances of the same item (predicted execution time) may exist in this collection. In Step 2, a counter is defined for the number of benchmarks specified by the user but not advertised by the cluster.

An execution time is predicted for each benchmark specified by the user in Step 3. If the cluster advertises the benchmark, a prediction for the execution time is calculated from the clusters benchmark result, and from the benchmark result and corresponding execution time specified by the user. This predicted execution time is added to the collection of all execution times. The algorithm uses the assumption that job execution time decreases linearly with increasing benchmark results. If the cluster does not advertise the benchmark, the number of unknown benchmarks is increased by one.

If no user-specified benchmark is advertised by the cluster, the predicted execution time is set to infinity and the algorithm terminates. This is done in Step 4. If the cluster advertises at least one of the benchmarks specified by the user, the broker compensates for those not advertised. In Step 5, the longest execution time from the predictions is obtained. The broker calculates a worst-case execution time by multiplying the longest predicted execution time with a penalty factor. The penalty factor determines how much slower the broker considers clusters for which only limited information is available. The factor can be configured by the user. If the user does not specify a penalty factor, the

Input:

- A set of k user-specified benchmark triples on the form $(BU_i, RU_i, TU_i), 1 \leq i \leq k$, where BU_i is the name of the benchmark, RU_i is the result of the benchmark and TU_i is the execution time for the application on the machine where the benchmark was run.
- A set of l benchmark values $(BC_j, RC_j), 1 \leq j \leq l$, which represents all benchmarks advertised by the cluster, BC_j being the name of the benchmark and RC_j the result.
- The penalty factor P, which affects the penalty given to clusters for not advertising user-specified benchmarks.

Action: Predict the execution time for a job if submitted to a certain cluster.
Output: The predicted average and worst case execution time.

1. Let `times` be an initially empty collection of all predicted execution times.
2. Let `missing` be the number of user-specified benchmarks the cluster does not advertise a value for. Set `missing` $\leftarrow 0$.
3. For each triple (BU_i, RU_i, TU_i):
   (a) If $BU_i = BC_j$ for some j, let $TC_i \leftarrow TU_i * RC_j / RU_i$ and add TC_i to `times`
   (b) Else, add one to `missing`.
4. If no triple was known by the cluster, that is, `times` is empty, set the average and maximum execution times to ∞. The algorithm terminates.
5. Let $t_{max} \leftarrow \max\{$`times`$\}$
6. In the `missing` number of empty entries in `times`, insert $P * t_{max}$.
7. Calculate the maximum and average execution times using the values in `times`.

Figure 14: Predicting execution time using benchmarks.

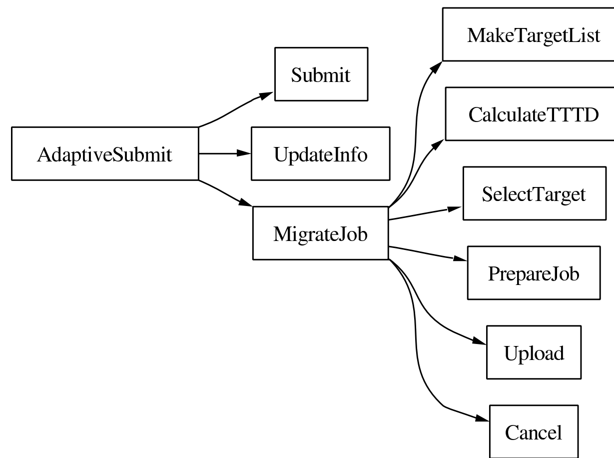

Figure 15: Program flow for queue adaptation.

default value, 1.25, is used.

The worst-case execution time is calculated in Step 6 and added to the collection of execution time predictions as many times as there were user-specified benchmarks not advertised by the cluster. The penalty given to each cluster depends on the results of the benchmarks advertised by the cluster, how many user-specified benchmarks the cluster failed to advertise and on the penalty factor. In Step 7, the maximum and average execution times are calculated from the collection of execution time predictions.

When comparing clusters to each other, the average execution time is used, as the average time reflects the cluster's overall performance better than the maximum time. When the broker adds the predicted execution time to the XRSL job description, the maximum value is used in order to prevent job preemption.

The exact syntax for specifying benchmarks in the job request is shown in an example in Section 6.

Users who configure the penalty factor must understand that a too low penalty factor (below 1) makes clusters for which little is known appear to be faster than those the broker has good knowledge about. As the benchmark-predicted execution time is the wallclock time allocated for the job, an underestimate of the execution time may result in preemption and job failure.

5.6 Features

This section describes the queue adaptation functionality and a feature that enables the submission of I/O-synchronized jobs.

Queue adaptation

Queue adaptation can be used to correct resource selection mistakes caused by the fact that available information always is outdated. Figure 15 gives an overview program flow for queue adaptation. The first step using queue adap-

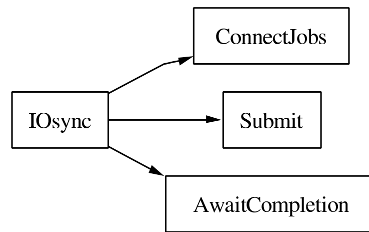

Figure 16: Program flow for submitting jobs using I/O-synchronization.

tation is to submit the job as described in Section 5.2 and depicted in Figure 8. After the job is submitted to the initially selected resource the adaptation phase begins. First, the status of the submitted job is retrieved, the adaptation phase ends once the job started executing on the resource it is currently submitted to. This corresponds to the job state INLRMS with the substate R, which denotes a running job. Queue adaptation includes repeating most of the steps from job submission. The information system is queried to retrieve a list of available submission targets. This list is filtered for targets not suitable for submission, and the TTTD is determined for each remaining target in the list. The best target found is compared to the resource the job is currently submitted to. If the job is likely to complete earlier on the new target, the job is migrated. During job migration, the broker prepares the job descriptions and submits it to the selected cluster. If this submission is successful, the broker cancels the previously submitted job. Otherwise, the job remains on the currently selected resource and no more queue adaptation is performed.

I/O-synchronization

Standard UNIX shells support the useful pipe tool, which lets the output of one process serve as the input to another. This functionality is useful in a grid environment as well. The broker implements I/O-synchronization, a feature similar to pipes, except in one aspect. As the NorduGrid middleware does not support streaming of job output, the preceding job must finish executing before the next one can start. Just like pipes, I/O synchronization can be used to connect two or more jobs in a series. Figure 16 shows the program flow of I/O-synchronization. When using this feature, jobs are submitted one at a time. Before submitting the next job, the broker ensures that the job description contains the correct input and output files. This is done by modifying the XRSL relations *stdin* and *stdout*. If the stdin relation is missing, or mismatching with the stdout relation of the previous job, the stdin relation is inserted / changed. Similar modifications ensure that the stdout relation matches the stdin relation of the next job. Next, the job is submitted, which can be done with or without queue adaptation. After the submission, the broker awaits the completion of the job by polling the MDS for changes in the job state. As the GRAM protocol is not used within NorduGrid, there is no support for notifications of job state changes. The broker must therefor poll the information system for changes in job state.

```
&(executable = my_app)(stdin = my_app.in)(stdout = my_app.out)
 (ngb-nas-lu-c = 350:65)(ngb-nas-bt-c = 275:65)(ngb-nas-cg-c = 120:75)
```

Figure 17: XRSL job request including benchmarks relevant for the job.

```
&(executable = my_program)(arguments = params input)(stdout = logfile)
(inputfiles= (params gsiftp://host1/file1)
             (input http://host2/file2))
(outputfiles= (results gsiftp://host3/my_program.results)
              (data gsiftp://host3/my_program.data)
              (logfile gsiftp://host4/my_program.log))
(outputfilessizes= (results 230MB)
                   (data  5GB))
```

Figure 18: XRSL job request with estimation of job output size.

6 Usage scenarios

This section contains examples of XRSL job descriptions where users include information used to improve the resource selection process. Command line options activating various features of the broker are also described.

Benchmark-based resource selection

Figure 17 shows a job request where the user includes benchmarks relevant to the application in order to improve resource selection. Note that it is necessary to include the execution time for the application along with each benchmark, as it is not certain that all benchmark results come from the same machine. The user specifies three NAS benchmarks. NAS is a benchmark suite consisting of problems from computational fluid dynamics. The classes A, B, C, and recently introduced, D, defines the problem size used in the benchmark. For the class BT and LU benchmarks class C, the results were 350 and 275 for a machine where the user's application completed in 65 minutes. For the CG benchmark of class C, the result was 120 on another, possibly slightly slower machine, on which user's application needed 75 minutes to finish. The reference machine(s) used for determining the runtime of the application need not be connected to the grid. Any machine available to the user can be used. There is no way to explicitly weight benchmarks, all benchmarks specified in the job description are considered to be of equal importance. As clusters can advertise any benchmark, the broker can not know in advance whether a user-specified benchmark exists or not. To avoid mistaking benchmarks for invalid XRSL attributes, the user must enter all benchmarks with "ngb-" as prefix. The broker ignores benchmarks that no machine advertises and benchmarks specified with an illegal format. Warnings are displayed to the user if such benchmarks are specified in the job description.

Network file transfers

The example in Figure 18 shows how a job request for a job for which large input and output files must be transferred. The job uses two input files and generates three output files. The XRSL relations *inputfiles* and *outputfiles* specifies files that should be transferred before and after the job execution. When predicting the time required for these transfers, the broker determines the actual size of the input files and uses user-estimates of the size of the output. The user is not required to specify the size of all output files. In the figure, the user knows that the logfile will be small and does not specify a size estimation for this file. Output size estimations are specified using the new XRSL attribute *outputfilessizes*. Estimated data sizes can specified in bytes or with any of the suffixes kB, MB, GB.

Command line options

As some features in the broker may not always be desirable to use, these are activated using command line options. When a user specifies the option ”-A”, the broker performs queue adaptation. If the advance reservations utility should be used, the ”-R” option must be specified. The option ”-S” activates I/O-synchronization. The usage of I/O-synchronization is independent of the values for the *stdin* and *stdout* XRSL relations, i.e., these relations can even be omitted. It is also possible to use any combination of two or more of the command line options.

7 Discussion

This section first discusses work by others that relate to this thesis. Next, possible improvements and alternative solutions are discussed, and finally, some conclusions are drawn.

7.1 Related work

There exist other attempts to estimate queue waiting time and application execution time in a grid environment. The authors of [32] try to find similarities between the submitted jobs and previously executed ones in order to predict the run time. These run time predictions are also used to predict queue waiting times [33]. The queue time for a job can be predicted if the execution times of all jobs in the batch queue are known in advance.

Alternatives to the TTTD performance metric exists, that may be useful for grid systems with a central broker. From a resource provider point of view, a balanced load for the provided resource may provide a good performance metric. Consider a scenario where the resource owner desiring a low load experiences unfairness on the grid, the own provided resources become heavily loaded while other grid resources remain idle. A broker always submitting a new job to the machine with the lowest load uses the *job_least_loaded* heuristic. To avoid congested networks, the broker can submit jobs to a machine where the required input data is present whenever this is possible. Using this *job_data_present* approach reduces network transfer overhead. This heuristic does most likely not achieve load balance, as data files typically are not

evenly distributed across the grid. Simulation studies of scheduling using the job_least_loaded and job_data_present algorithms as well as other resource brokering heuristics are presented in [30].

A framework for advance reservations and coallocation is defined in [10], and the impact of advance reservation on local scheduling performance is studied in [34].

Job adaptation, including the use of service level agreements, contract monitoring and the migration of executing jobs upon contract violation, has been done by the GrADS project [1]. In addition, the GrADS project uses the NWS for network bandwidth and latency predictions. The Cactus toolkit [15] implements a job migration mechanism similar to the one found in GrADS.

The project (apart from Globus) that has had perhaps the highest impact on grid usage is the Condor project [22, 39] for *High Throughput Computing*, HTC. The aim of HTC is to complete as many tasks as possible within the period of a week, a month or a year. Each task executed in a HTC environment typically uses only one CPU, and possibly has modest requirements for memory but has a long execution time. The Condor installation, the *pool*, consists of worker nodes and job submission nodes. Typically, all available resources, including desktop machines, are added to the pool as worker nodes. Individual machine owners decide how and when their resources may be used for Condor jobs. Although not originally designed for grid usage, later extensions to Condor allows the usage of remotely located worker nodes, and makes it possible to temporarily bring in Globus-enabled resources as worker nodes in the pool [38]. A Condor pool differs from typical Globus-based grids in that each pool has a central resource broker. Furthermore, Globus RSL is a pure job submission language while the *classads* used within Condor has a broader use. Both job requests and resource information are advertised within the pool using classads. The Condor resource broker uses classads to match resources (worker nodes) with job requests. Initial work in this thesis project included setting up a local Condor pool.

7.2 Future work

This section discusses alternative ideas and possible improvements. Some of the future work will be completed shortly, as the broker is about to be installed on Swegrid [37] resources.

Cost models

Tests of the broker confirms that it makes reasonable resource selection decisions when basing these decisions on the TTTD. However, further tests and a thorough evaluation is required. More specifically, resource utilization versus individual job turnaround time should be investigated. This should preferably be done using simulation tools. Other possible simulations includes investigating how these issues are affected if all users in the grid use the new broker compared to the case when only some use it. Also, resource utilization and job turnaround time may change if not all jobs executed on the grid resources are submitted by the grid broker. This occurs on machines that execute both grid jobs and jobs submitted by local users.

Alternatives and complements to using TTTD as a basis for resource selection should also be investigated. Statistical methods for predicting application

execution times and queue waiting times are studied in [32, 33]. Such methods would provide more accurate estimates of queue waiting time in the case when advance reservations can not be used. A pure statistical approach may also be useful for predicting network bandwidth for resources that do not run a NWS sensor. The current implementation makes rough guesses of queue waiting time and network bandwidth when required information is not available.

Configuration

Another remaining issue is configuration. The integration of the new broker into an existing NorduGrid installation is cumbersome. It must also be easier to configure locations of, e.g., the NWS name server host, the NWS memory host and the reservation scripts. Furthermore, the NWS clique must be restarted every time a new resource is added to the grid, a non-scalable solution. NWS sensors has shown to be error-prone, the restart of such sensors must be handled automatically in a production system. How to install and configure the system should be included in an installation manual, which remains to be written.

Network file transfers

There exists two obvious improvements related to networks transfers. First of all, a file replication system, such as the currently developing NorduGrid Smart Storage Element [21] should be utilized. This allows the broker to transfer the replica of the requested file stored closest to the computational resource, keeping network traffic at a minimum. Another possible solution is to integrate the broker with a storage broker. Examples of storage brokers includes the Storage Resource Broker, SRB [2]. Secondly, the model used to estimate transfer times needs to be revised. Grouping together files stored at the same host should improve accuracy in network transfer time predictions. However, this model assumes that the network connecting the cluster to the rest of the grid is fast enough to transfer all required job files in parallel using the full bandwidth, which is not always true. Care must be taken in the predictions when the network connecting the cluster to the rest of the grid is the bottleneck. However, it is difficult to detect bandwidth bottlenecks.

Benchmarks

The infrastructure required for benchmark-based resource selection is a general implementation. Neither the broker nor the LDAP schema need to be modified if new benchmarks are to be added. The only required modification is to append the new benchmark and its result to the file from which the information provider reads benchmark data. Widespread adaptation of the benchmark schema is required for achieving good performance of the broker. This gives rise to a "political" issue. Resource owners must to some extent agree on what benchmarks all resources should advertise. Research communities and other groups with specific requirement may later add benchmarks suitable for their applications. Furthermore, benchmark naming standardization is vital. Another issue related to naming is the magnitude of the units used. Exemplifying using DGEMM, does 4000 mean 4000 Gflop/s or 4000 Mflop/s? One way of circumventing this problem is to allow suffixes such as 'k', 'M' and 'G' in attribute values.

Usability concerns

The usage of the broker puts some demands on the user. Estimating the size of the job output may sometimes be difficult. However, a user rerunning the same application quickly learns how much output it generates and can in later submissions include estimates of the job output size in the job descriptions. To fully utilize the broker, the user must know what benchmarks are relevant for the performance of the submitted application. Users developing their own applications usually has some sense of this. For legacy applications and for large, complex programs it may be hard to determine the benchmarks that characterizes performance.

Design issues

The advance reservation protocol works well enough for the purpose of minimizing TTTD. In more complex job submission scenarios such as coallocation, negotiation between the broker and the resource(s) is required. The current implementation uses a simple request-response protocol. Even though the broker specifies a start time for the reservation in the request, the resource may create a reservation with a later start time. Using the current protocol, coallocation would involve repeatedly requesting and releasing reservations. With this in mind, it would be convenient with a two-phase protocol where the broker requests a reservation and the resource returns a reservation offer. This offer is either accepted or declined by the broker. A coallocating broker could then consider offers from the involved resources, accept those forming the coallocated job, and decline the others. Unanswered offers will time out in a soft-state fashion. This approach has the same advantages and caveats as a transaction system. An outline of a two-phase protocol for reservation is discussed in [23].

Another improvement of the reservation protocol would be to allow a range of possible reservation start times. The broker requests a latest start time, but also specifies how long before this start time the job is allowed to start. Allowing the job to start earlier than the latest start time is not useful for coallocation purposes. This flexibility does however allow the local scheduler to start the job whenever best suitable for the scheduler, which results in a higher resource utilization. Job mean wait time and resource utilization for systems where the scheduler allows advance reservations are studied in [34].

The support for I/O-synchronization of jobs may be useful for some users. Ideally, I/O-synchronization and similar features should not be part of the broker itself, they should rather be part of a workflow system. Workflow systems could use an API for the broker to submit each individual job. Construction of a grid workflow system is however far beyond the scope of this thesis.

The Globus toolkit 2 used in NorduGrid is becoming an outdated technology. Since the NorduGrid project started, new technologies and standards such as the Open Grid Services Architecture, OGSA [14], the Open Grid Services Infrastructure, OGSI [44], and most recently, the Web Services Resource Framework, WSRF [45] has emerged. Redesigning and reimplementing the broker using such web service-based architectures has the advantage that the tight coupling to NorduGrid disappears. The broker could then be integrated to any grid system built upon web services. It could still be possible to use the broker within NorduGrid, e.g., the SGAS project [9] demonstrates that OGSA-based services

can be integrated into NorduGrid. Deciding which new technologies to use can be a difficult issue. The Globus toolkit 3, the reference implementation of OGSI is not even a year old, but its replacement, the WSRF-based Globus toolkit 4 will be released during the third quarter of 2004. Choosing the wrong architectures, one can soon find oneself stuck with deprecated technologies. A future broker implementation based on web service technologies should preferably use some other job description language than XRSL. The Global Grid Forum will within shortly standardize the Job Submission Description Language, JSDL, which will be the preferred job description language to use in a web service-based resource broker.

Policy implications

When using advance reservations, the broker makes reservations and releases those not used, assuming that no cost is associated with making a reservation and later releasing it. Possible resource usage policies may associate a cost with each created reservation, making it more expensive to activate the reservation capability of the broker than running without it. The discussed soft-state reservation protocol could possibly be a solution to this, given that no cost is associated with making a soft-state reservation.

When a job submitted with an advance reservation finished, the grid manager removes the reservation in the LRMS. One possible policy would be to charge users for the reserved time and not the time actually consumed by the job. In such a scenario, a user who finds out that a job submitted with a reservation finished much earlier than predicted may want to submit more jobs to the same reservation. For this to be possible, users must be able to specify the reservation id directly in the XRSL job description. The grid manager must also be configured not to run the script that releases reservations upon job completion.

Minimizing the TTTD for each submitted job may not be the desired result even though this approach lowers the turn-around time in the system. When considering economic aspects of grid usage, the broker may need to treat different parts of the TTTD differently. One possible scenario is where the user is charged for the consumed CPU time, and the usage rates are the same for all resources. For this case, it may be desirable for the broker to minimize the execution time by finding the fastest machine, even though using this machine does not result in the shortest possible TTTD.

7.3 Conclusion

A user-owned broker that minimizes the TTTD predicted for each job gives the user a high utilization of the available grid resources. Given reasonably correct predictions, such a broker can be used for all kinds of applications. The implementation shows that it is feasible to use the TTTD as performance metric, improvements of the used predictions are however possible.

8 Acknowledgements

First of all, I would like to thank my supervisors, Erik Elmroth and Åke Sandgren. Erik suggested, inspired and encouraged this work. He also proofread this

thesis and other grid-related documents appearing from my keyboard. Åke resolved technical issues, co-developed parts of the reservation system, and showed great patience over my poor knowledge of system administration.

Past and present people in "amanuenskorridoren" kept the spirit up and answered questions regarding exotic programming details.

Peter Gardfjäll provided good company on the thesis-related travels and is always keen on discussing various aspects of grid computing.

I would also like to thank my family and Johanna for their support and interest in this work, even though the technical details might have bored them.

References

[1] D. Angulo, R. Aydt, F. Berman, A. Chien, K. Cooper, H. Dail, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, Kesselman C, M. Mazina, J. Mellor-Crummey, D. Reed, O. Sievert, L. Torczon, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive grid programs. In *Proceedings of the IPDPS Conference*, 2002.

[2] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC storage resource broker. In Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of CASCON'98*, 1998.

[3] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, B. Davie, J. Wroclawski, and E. Felstaine. A framework for integrated services operation over diffserv networks. Internet, May 2004. http://www.ietf.org/rfc/rfc2998.txt.

[4] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. Gass: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems.*, 1999.

[5] John Brooke and Donal Fellows. Draft discussion focument for gpa-wg - abstraction of functions for resource brokers. Technical report, University of Manchester, 2003. http://grid.lbl.gov/GPA.

[6] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10).* IEEE CS Press, Aug. 2001.

[7] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T.Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *Int. J. Supercomput. Appl.*, 10(2):123–130, 1996.

[8] P. Eerola, B. Kónya, O. Smirnova, T. Ekelöf, M. Ellert, J.R. Hansen, J.L. 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, P. Gardfjäll, L. Johnsson, O. Mulmo, and T. Sandholm. An OGSA-based Accounting System for Allocation Enforcement Across HPC Centers. Submitted, February 2004.

[10] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of ACM/IEEE Intl Workshop on Quality of Service*, 1999.

[11] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[12] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. Supercomput. Appl.*, 15(3), 2001.

[13] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputer Applications*, 2:115–128, 11 1997.

[14] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid. Internet, May 2004. http://www.globus.org/research/papers/ogsa.pdf.

[15] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Masso, Thomas Radke, Edward Seidel, and John Shalf. The cactus framework and toolkit: Design and applications. In V. Hernandez J.M.L.M. Palma, J. Dongarra and A. A. Sousa, editors, *Proceedings of VECPAR'2002*, 2002.

[16] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The next logital step toward a nationwide virtual computer. Technical report, University of Virginia, 1994.

[17] F.J. Hirsch. Introducing SSL and certificates using SSLeay. *World Wide Web Journal*, 2(3 Summer), 1997.

[18] International Telecommunication Union. Recommendation x.500 (08/97): Open systems interconnectoin - the directory: Overview of concepts, models and services.

[19] Subramanian Kannan, Peter Mayes, Mark Roberts, Dave Brelsford, and Joseph F Skovira. *Workload Management with LoadLeveler*. IBM Corp., 2001.

[20] Aleksandr Konstantinov. The nordugrid grid manager and gridftp server. Internet, Mar. 2004. http://www.nordugrid.org/documents/GM.pdf.

[21] Aleksandr Konstantinov. The nordugrid "smart" storage element. Internet, Apr. 2004. http://www.nordugrid.org/documents/SE.pdf.

[22] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[23] J. MacLaren. Advance reservations state of the art. Technical report, Global Grid Forum, 2003. http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html.

[24] Jon MacLaren. Advance reservations state of the art. Technical report, Global Grid Forum, 2003. http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html.

[25] netperf. http://www.netperf.org. Internet, May 2004.

[26] NorduGrid. http://www.nordugrid.org. Internet, May 2004.

[27] OpenLDAP. http://www.openldap.org. Internet, May 2004.

[28] OpenSSL. http://www.openssl.org. Internet, May 2004.

[29] Portable Batch System. http://www.openpbs.org. Internet, May 2004.

[30] Kavitha Ranganathan and Ian Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing*, 1:53–62, 2003.

[31] The globus resource specification language rsl v1.0. Internet, May 2004. http://www-fp.globus.org/gram/rsl_spec1.html.

[32] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In D. G. Feitelson and L. Rudolph, editors, *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[33] 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, *Proceedings of the IPPS/SPDP '99 Workshop on Job Scheduling Strategies for Parallel Processing*, 1999.

[34] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In Jose Rolim et al., editors, *Proceedings of the IPDPS Conference*, 2000.

[35] Douglas R. Stinson. *Cryptography, theory and practice.* CRC press, New York, 1995.

[36] Supercluster.org. Center for HPC Cluster Resource Management and Scheduling. http://www.supercluster.org. Internet, May 2004.

[37] Swegrid. http://www.swegrid.se. Internet, May 2004.

[38] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons Inc., December 2002.

[39] The Condor Project. http://www.cs.wisc.edu/condor. Internet, May 2004.

[40] The Globus Alliance. Gridftp universal data transfer for the grid. Internet, May 2004. http://www-fp.globus.org/datagrid/deliverables/C2WPdraft3.pdf.

[41] The Globus Alliance. http://www.globus.org. Internet, May 2004.

[42] Inc. The Stencil Group. The evolution of UDDI. Internet, May 2004. http://www.stencilgroup.com/ideas_scope_200207uddiv3.pdf.

[43] ttcp. http://www.pcausa.com/utilities/pcattcp.htm. Internet, May 2004.

[44] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open grid services infrastructure (ogsi) version 1.0. Internet, May 2004. http://www.globus.org/research/papers/Final_OGSI_Specification_V1.0.pdf.

[45] Web Service Resource Framework. http://www.globus.org/wsrf/. Internet, May 2004.

[46] R. Butler Von Welch, D. Engbert, I. Foster, S. Tuecke, J. Volmer, and C. Kesselman. A National-scale Authentication Infrastructure. *IEEE Computer Society Press*, 2000.

[47] Rich Wolski. Dynamically forecasting network performance using the network weather service. *Journal of Cluster computing*, 1:119–132, 1998.

[48] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol. Internet, May 2004. http://www.ietf.org/rfc/rfc1777.txt.

A List of abbreviations

DN	Distinguished Name
GIIS	Grid Index Information Service
GRAM	Grid Resource Allocation Manager
GRIS	Grid Resource Information Service
GSI	Grid Security Infrastructure
JSDL	Job Submission Description Language
LDAP	Lightweight Directory Access Protocol
LRMS	Local Resource Management System
MDS	Monitoring and Discovery Service
NAS	NASA Advanced Supercomputing Division
NWS	Network Weather Service
OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure
RSL	Resource Specification Language
TTTD	Total Time To Delivery
VO	Virtual Organization
WSRF	Web Services Resource Framework
UDDI	Universal Description, Discovery and Integration
XRSL	eXtended Resource Specification Language

B Example of broker configuration file

```
# This is a sample .ngrc file
# As you already guessed, a line starting with '#' is a comment

# Minimum bandwidth between any two clusters
# Used to predict file transfer time when no bandwidth
# forecasts are advertised by the cluster.
# N.B. A too high value may cause job failure.
# 1 Mbit/s:
NGMIN_BANDWIDTH=132322

# Average execution time for any job (grid and non-grid)
# submitted to a cluster. This value is used to determine queue
# waiting time when advance reservation are not used.
# 6 hours:
NGAVG_JOBTIME=3600

# Max execution time in minutes for user's jobs.
# The value will be used when no other execution time is available.
# N.B. A too low value may cause job failure.
# 24 hours:
NGMAX_EXECTIME=21600

# Penalty factor used by broker to compensate
# for user-specified benchmarks not advertised by cluster.
# N.B. a value below 1 may to cause job failure.
NGBENCHMARK_PENALTY=1.45
```