Master's Thesis

A New Grid Manager for NorduGrid

– A Transitional Path

Thomas Christensen
Rasmus Aslak Kjær

June 2nd, 2005
Aalborg University

Aalborg University

Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

Title:

A New Grid Manager for NorduGrid –
A Transitional Path

Project period:
DAT6, Feb. 1st – Jun. 2nd, 2005

Project group:
D607A / B2-201

Group members:
Thomas Christensen,
thomas@cs.aau.dk

Rasmus Aslak Kjær,
rak@cs.aau.dk

Supervisor:
Gerd Behrmann

Number of copies: 6

Total number of pages: 93

Abstract

This thesis describes the design, implementation and evaluation of a new grid manager for the NorduGrid middleware. NorduGrid is among of the most successful grid computing projects with more than 50 clusters and 5000 CPUs participating. The current NorduGrid grid manager is rigid and difficult to extend with new functionality.

This motivates the design of a new plugin-based event-driven architecture for grid managers. The architecture is implemented in the Python programming language and used as the basis for a new prototype drop-in replacement for the current NorduGrid grid manager. The new grid manager is developed as a drop-in replacement to ease the transition toward supporting novel grid features.

To evaluate the extensibility of the new grid manager, several advanced use cases for the grid manager are examined and approaches for achieving them in the new grid manager are presented. The evaluation shows that the new grid manager is extensible, efficient and customisable. Approaches to improve the fault-tolerance of the new grid manager are presented to improve the resilience to system failures.

Preface

This master's thesis documents the work on improving the NorduGrid middleware done by group D607A during the spring semester of 2005. The thesis is written as part of the DAT6-term at the Department of Computer Science at Aalborg University.

We would like to thank Henrik Thostrup Jensen for assisting us with installing and configuring the NorduGrid ARC software, for him letting us use pyRSL, along with his many bright ideas regarding improvements to NorduGrid. We would also like to thank Josva Kleist for inspiration and response on the new grid manager design.

Aalborg, June 2005.

Thomas Christensen
thomas@cs.aau.dk

Rasmus Aslak Kjær
rak@cs.aau.dk

Contents

1 Introduction

This thesis describes the design, development and evaluation of grid middleware for NorduGrid, a Scandinavian based computational grid. The concept of a computational grid is still rather new and therefore the problem domain addressed by grid middleware is subject to constant changes, due to the emergence of new use cases and experiences with running grids which combined poses new challenges. This has inspired the development of a software architecture for grid middleware capable of evolving and adapting to the changing requirements that novel use cases of computational grids incur.

In this chapter the concept of a computational grid is presented, followed by a generic model for the software used to realise computational grids, namely grid middleware. Finally, the middleware of NorduGrid is outlined in brief followed by a specification of design requirements to the middleware developed in this project.

1.1 Computational Grids

There is some controversy over what constitutes a computational grid, and the term has been much hyped in recent years. In the following, we adopt this definition by Rajkumar Buyya [1]:

> *Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements.*

The term is a reference to the power grid [2]. The users of the power grid are oblivious to the exact nature of the grid and simply plug in their appliances, expecting that the appropriate amount of electrical current is delivered to them. Furthermore, producers in the power grid

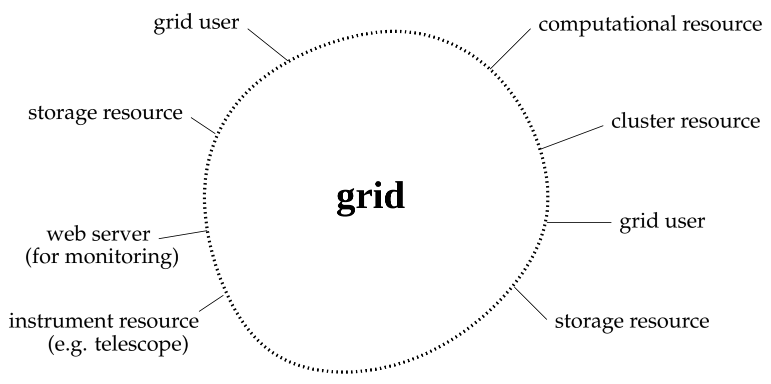


Figure 1.1: A conceptual view of a grid with resources and users.

range from a farmer with a small windmill in his field to large national power plants. Potentially anyone can participate in the power grid as a producer or consumer.

This concept has given birth to the idea of computational grids, i.e. that computers from diverse locations are interconnected to let users share the resources. The idea is illustrated in Figure 1.1. Users should be able to connect to the grid and submit tasks without having to concern themselves with where the tasks are run (unless they want to), and resources should be able to connect to the grid and receive tasks that are appropriate for them. Resources can be computational resources or storage, or even access to specialised equipment such as measurement instruments. The goal is to have better utilisation and wider access to resources, even across traditional organisational boundaries.

Despite the original idea of a single world-wide computational grid, as presented in [2], the present distribution seems to bear similarities to that of power grids. Several computational grids co-exist in diverse communities, covering academic, country level and multi institutional efforts. Currently, most computational grids are either geared towards utilising high-performance computing clusters as resources, or screen saver science where ordinary desktop machines work on a small part of a problem when the processor is idle.

Since a grid by nature is a large distributed system that spans many organisations, there are several challenges that must be overcome in the design of a grid:

- **Security.** The security model must be flexible enough to support modelling the diverse conditions in the various organisations – the local resource policies must be respected – and modelling of virtual organisations [3] spanning multiple organisation for resource sharing. This problem includes authentication and authorisation of users and resources, and enforcement of access control.

- **Fault-tolerance.** A large grid will continually suffer hardware failures and must be able to cope gracefully with them and at the same time stay up. It should also take precautions against losing jobs and data.

- **Flexibility.** Much is expected of grid technology, so a grid design should preferably be flexible enough to be able to support future use cases.

- **Ease of use.** A successful grid relies on many parties adopting the idea, reaching a critical mass, so it must not be too difficult or cumbersome to use or setup a resource for the grid.
- **Scalability.** A grid must be able to scale to a large number of users and resources. This affects both administration of the grid and the operation of the software itself – e.g. it is not possible for a design that relies on a central authority to scale to something the size of the Internet.
- **Privacy.** Protecting the privacy of information when submitting jobs to the grid. Jobs can work on valuable data sets, and situations can arise where it is unacceptable that these data can be intercepted and read by anyone other than the job itself and the submitter.
- **Accounting.** High performance computing resources are not cheap to acquire and maintain. So as it is the case with the power grid, accounting of resources spent by consumers is needed if the grid is not to be operated gratis. This can be imagined to evolve into grid economy where resources receive micro-payments from job submitters in return for running jobs.
- **Heterogeneity.** The portability of the grid software is also important. Ideally there should be no technical boundary preventing resources from participating in the grid. But even if the grid software itself is portable, the problem of making the job software run on diverse architectures remains.

None of these issues are trivially solved. The current grid designs have mostly dealt successfully with only a subset of them. A single design that is flexible enough to cover all grid use cases and at the same time proves to solve the above issues has yet to reveal itself.

1.2 A Generic Grid Model

At a conceptual level the responsibility of a computational grid is to tie users, jobs and resources seamlessly together. The glue that ties these entities together is called middleware; a software layer between client and server processes that provides translation, conversion or extra functionality. At present, several computational grids are running their own grid middleware. These grids vary greatly with respect to purpose, architecture, design and implementation. Even though different in many respects, practically all grids share a common foundation of components necessary to realise their middleware. [4]

- **Grid Fabric.** This consists of all the globally distributed resources that are accessible from anywhere on the Internet. These resources could be computers running a variety of operating systems, storage devices, databases and special scientific instruments.
- **Core grid middleware.** This offers core services such as remote process management, co-allocation of resources, storage access, information registration and discovery and aspects of Quality of Service (QoS) such as resource reservation and trading.
- **User-level grid middleware.** This includes application development environments, programming tools and resource brokers for managing resources and scheduling application tasks for execution on global resources.

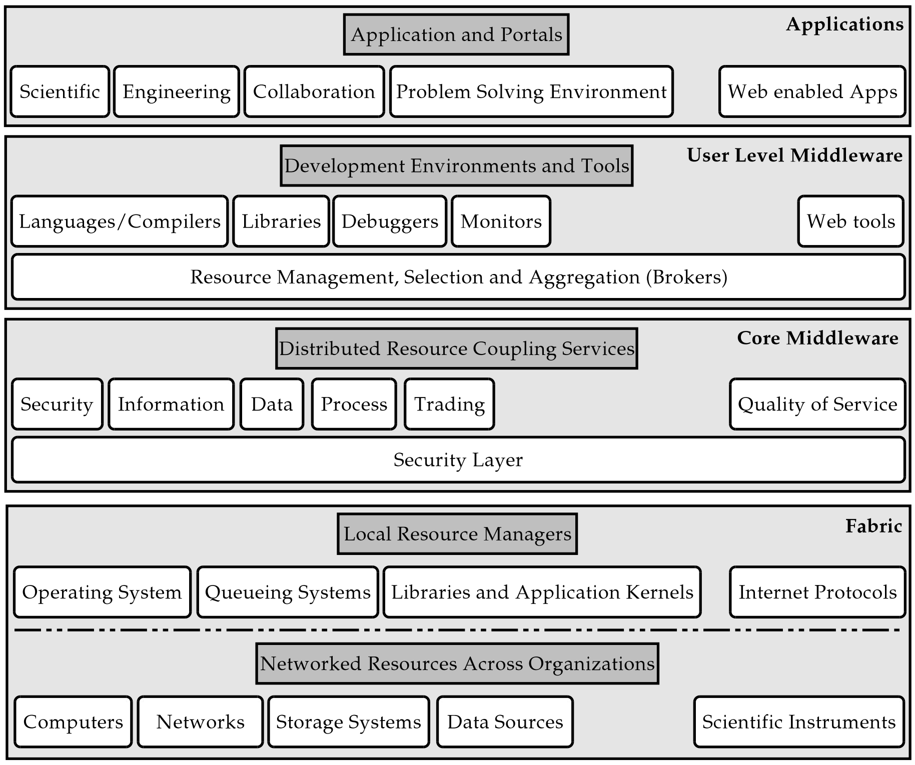


Figure 1.2: A generic model for a grid architecture. The depicted components are necessary in some for in order to create a computational grid. [4]

- **Grid applications and portals.** Grid applications are often developed using grid-enabled languages and utilities such as HPC++ or MPI. An example application, e.g. a parameter simulation, would require computational power, access to remote data sets and may need to interact with scientific instruments. Grid portals offer web enabled application services, where users can submit and collect results for their jobs on remote resource.

A generic model of software components for a computational grid is depicted in Figure 1.2.

Furthermore, to facilitate the collaboration of multiple organisations running diverse autonomous heterogeneous resources the following basic principles should be adhered to when designing middleware for computational grids. The middleware should:

- not interfere with the existing site administration or autonomy.
- not compromise existing security of users or remote sites.
- provide a reliable and fault tolerant infrastructure with no single point of failure.
- provide support for heterogeneous components.
- use standards, and existing technologies, and facilitate interoperability with legacy applications.

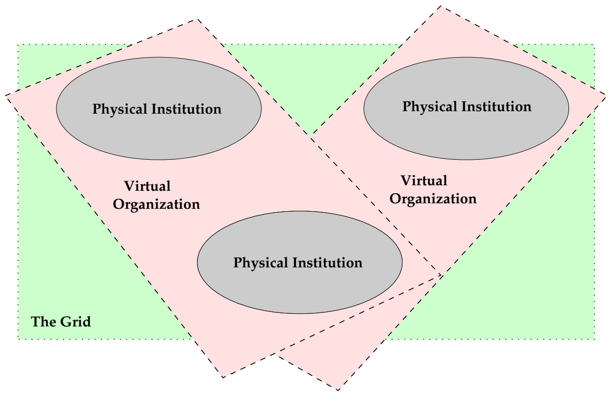


Figure 1.3: Virtual organisations allow for a logical partitioning of interrelated grid participants. Institutions and individuals can be part of several virtual organisations concurrently.

The primary benefit of computational grids is the enhanced possibility to share resources. However, alongside resource sharing comes a host of issues regarding fair sharing, security, user management and other implications. To embrace these concerns [3] defines the concept of virtual organisations (VO) as a set of individuals and/or institutions governed by the same sharing rules.

A VO is a number of *individuals* and/or *institutions* that have decided to share *resources* among each other. Thus, a VO represents a notion of trust between the involved parties. Furthermore, in the context of computational grids, the participants in the VO have to use the same set of *tools* to both use and share their resources in the VO. These tools are referred to as the *middleware* in the generic model of a computational grid as illustrated in Figure 1.2 on this page.

The VO concept is essential for computational grids as it addresses two fundamental issues, namely those of user management and means of managing resource sharing. It seems only fair to expect that any grid should support the VO concept and that any grid must provide the necessary infrastructure to allow for VO's.

1.3 Project Goals

There exists several grid toolkits for aiding the implementation of grid middleware and work has also been done to form standards for grid architectures. The Globus Alliance is developing a toolkit for building a grid, including frameworks for building information systems for registration of resources and user authentication, replica location systems for management of data, and direct interfaces to resources. The Globus Toolkit has recently been released in a fourth version, where the major focus is on standardising communication as web services [5].

Parts of the Globus Toolkit have been used as the basis for several grid solutions, both in national grids and international efforts. One of the larger currently operating grids is NorduGrid. Other large globus-based grids include the Large Hadron Collider Grid Computing Project [6] at CERN, the Grid Physics Network (GriPhyN) [7] and Teragrid [8] (which are both developed and run by American universities).

Referring to the generic model from Section 1.2 on page 5, the core grid middleware of NorduGrid is comprised of a *grid manager* and an *information system*, while the user level middleware consists of a *user interface*. The grid manager runs on every resource in NorduGrid, enabling the resource to receive jobs from users, submitted via the NorduGrid user interface. From a users point of view, the only interaction is with the NorduGrid user interface. From a grid point of view the user interface looks up available resources in the information system, using it as the "yellow pages" on the grid, in order to locate suitable and available resources. After finding, perhaps several, suitable and available resources, the user interface submits the job to the grid manager running at one of these resources. When a grid manager, at a resource, receives a job it is responsible for executing it. This generally includes downloading necessary input data, executing the job program itself, handling any failures that might occur. This is a simplified outline of the middleware of NorduGrid, the NorduGrid Advanced Resource Connector (ARC), as the middleware is called, is discussed in further detail in Chapter 2 on page 11.

At the time NorduGrid was conceived the primary objective for the developers was rapid deployment. The middleware for NorduGrid should be deployed as fast as possible with the least possible effort. Four years later the middleware, and NorduGrid in general, has proven to be a success. The grid has obtained critical mass in terms of day-to-day users, and furthermore it is part of a major European grid initiative for processing massive amounts of data from experiments in high energy physics. Therefore, the future of NorduGrid seems promising to some extent.

However, new grid middleware is being developed concurrently and the NorduGrid grid manager, currently version 0.4.5, is still in its infancy. As new use cases for computational grids emerge, so does the need for new features in the grid manager. Furthermore, the continuous use of NorduGrid has revealed the need to customise and adapt the grid manager to the environment in which it is used. As an example, recall that the grid is responsible for downloading input data for the jobs it receives. This task has surprisingly proven to be one of the most demanding for the computer running the grid manager. If NorduGrid is to maintain its position as a leader in the current field of middleware for computational grids, the grid manager needs to be designed to cope with future demands.

The purpose of this project is to design and implement a drop-in replacement *grid manager* for NorduGrid, i.e. a backwards compatible grid manager able to co-exist with the existing grid manager running at other grid sites. The replacement grid manager should be designed for change, embracing the inherent need to *customise*, *extend* and adapt the grid middleware in response to altered functionality and/or performance requirements. Naturally, the replacement grid manager should be designed to process jobs *efficiently*. The exact meaning of these requirements in this context is elaborated below:

- **Efficiency.** The grid manager should, whenever possible, employ the solution which best balances low latency and high throughput when processing jobs, e.g. avoid file-based communication and polling in general.
- **Extensibility.** Covers the ease with which it is possible to extend the functionality of the grid manager, e.g. implementing any of the advanced features mentioned below.
- **Customisability.** Covers the possibilities of tweaking existing functionality in the grid manager to better suit local demands or requirements. For instance changing the way jobs are monitored in LRMS, changing job states due to local policies or ideas.

- **Fault-Tolerance.** Covers the dependability of the grid manager. It is absolutely vital that the grid manager provides all reasonable measures to ensure that jobs and data are not lost, even in the case of system crashes.

Furthermore, the grid manager should facilitate the implementation of advanced features required by modern grid middleware. The following present the requirements of advanced features.

- **Advance reservations.** This feature allows for a user to reserve a resource for later use. For this feature to be implemented the grid manager must be able to provide operations allowing such reservations. [9] has already shown how to accomplish without using the grid manager but instead using a rather awkward extension to parts of the Globus Toolkit.
- **Separation of the input/output component.** Based on the experiences with the current grid manager implementation in NorduGrid, the design of the input/output component should allow for it to be separated from the core of the grid manager, that is the part that handles incoming job submissions and job management in general.
- **Live upgrades of the grid manager.** A truly customisable and extensible grid manager should provide mechanisms to allow a running grid manager to be completely reconfigured *on-the-fly*, i.e. no service failures are perceivable from the users point of view during the upgrade.
- **Distributing the grid manager for high availability.** Distributing the components of the grid manager onto separate systems could be useful to provide fail-over functionality and load balancing in the grid middleware.

The choice to improve the grid middleware specifically for NorduGrid has been influenced by our experiences with this middleware from an earlier project[10]. Furthermore, the fact that the NorduGrid middleware is actually running at Aalborg University allows for easy access to testing the developed software in the NorduGrid infrastructure.

1.4 Summary

The concept of computational grids have been established as a type of parallel and distributed system that enables the sharing of resources. Computational grids bears some resemblance with the well known power grids that our world depends on, but of course the analogy has some limitations, e.g. the fact that computational resources can not be preserved over time as electric current can.

A generic model for designing grid middleware, including generic requirements to such middleware, has been presented. The model defines four basic components of grid middleware, namely fabric, core grid middleware, user-level grid middleware and grid applications and portals.

To outline the goals of this project, several requirements to the design of a replacement grid manager for NorduGrid has been defined. The requirements fits into two different categories; general design requirements, and specific feature requirements. The general requirements improves the quality of the design, while the feature requirements ensures that the designed grid manager is able to support the features of tomorrows computational grid.

To learn from existing experiences with the design of a grid manager, a thorough examination of the current middleware in NorduGrid is necessary. Furthermore, such examination will outline how a new drop-in replacement grid manager can be designed to integrate seamlessly into the existing middleware, and also how such a grid manager should be designed to meet the requirements stated in the above.

The rest of the thesis is structured as follows. In Chapter 2 on this page the current middleware of NorduGrid is examined in detail. In Chapter 3 on page 27 the design of the architecture implementing a grid manager for NorduGrid is presented. In Chapter 4 on page 37 a prototype drop-in replacement for the current NorduGrid grid manager, based on the new architecture, is presented. In Chapter 5 on page 45 the extensibility of the new grid manager is established by illustrating the possibilities of adding new advanced features. Chapter 6 on page 59 contains an evaluation of the new architecture and prototype grid manager implementation. Finally Chapter 7 on page 65 presents a conclusion for the project.

2 NorduGrid Advanced Resource Connector

This chapter presents a thorough examination of the NorduGrid middleware and its constituents. The middleware is called the NorduGrid Advanced Resource Connector, or simply the NorduGrid ARC. Initially, the history, purpose and present state of NorduGrid is presented, followed by a description of the job concept. The architecture of the middleware is introduced by following the task flow of a job submission into the NorduGrid grid manager. After presenting the design, the problems identified by examining NorduGrid is discussed, and related to the design requirements set forth in Chapter 1 on page 3. The identified problems and their relation to these requirements motivates the decision to design an architecture on which to build the new grid manager.

2.1 History of the Project

The NorduGrid project started in May 2001 as a collaborative effort between research centres in Denmark, Norway, Sweden and Finland. The project was initially named the Nordic Testbed for Wide Area Computing and Data Handling [11]. Continuous development and challenges have matured the project beyond the state of a testbed and into a widely deployed production grid. At present the grid development, resources and usage are administered by the Nordic Data Grid Facility (NDGF), which is part of the North European Grid Consortium. The grid, however, is still referred to as NorduGrid.

The purpose of the NorduGrid project was to create a testbed for a Nordic grid infrastructure and eventually to develop a grid that meets the requirements for participating in the ATLAS experiment [11]. ATLAS, A Toroidal LHC ApparatuS, is a detector for conducting high-energy particle physics experiments that involve head-on collisions of protons with very high energy. The protons will be accelerated in the Large Hadron Collider, an underground accelerator ring 27 kilometres in circumference at the CERN Laboratory in Switzerland. The ATLAS experiments have around 1800 physicists participating from more than 150 universities in 34 countries; the actual experiments are planned to begin in 2007 [12].

The experiments are expected to generate 12-14 petabytes data per year that needs to be analysed. To prepare the participating entities for these massive amounts of data, the ATLAS Data Challenges have been posed. The first of these data challenges ran throughout 2002 and was completed in 2003. NorduGrid was the Scandinavian contribution to this challenge [13].

During the initial phase of designing the grid, the fundamental idea was that NorduGrid should be built on existing pieces of working grid middleware and the amount of software development within the project kept at a minimum. Once the basic functionality was in place the middleware was to be further extended, gradually turning the testbed into a production grid. And at the start of the NorduGrid project in May 2001 a general design philosophy was formulated as follows:

- Start with simple things that work and proceed from there
- Avoid architectural single points of failure
- Should be scalable
- Resource owners retain full control of their resources
- As few site requirements as possible
  - No dictation of cluster configuration or install method
  - No dependence on a particular operating system or version
- Reuse existing system installations as much as possible
- The NorduGrid middleware is only required on a front-end machine
- Computational resources are not required to be on a public network
- Clusters need not be dedicated to grid jobs

Existing toolkits were then examined to decide on which to base the NorduGrid middleware. The available possibilities were narrowed down to two alternatives; the Globus Toolkit and the software developed by the European DataGrid project (EDG). But the initial idea to use only existing middleware components, and not having to develop any software in the project soon proved impossible to realise. Since then, much software has been developed to take NorduGrid into its current state as a fully fledged computational grid. In the following we describe the NorduGrid software. [11]

2.2 The Job Concept

The purpose of a computational grid is to provide means for executing large scale jobs. A fundamental design decision is thus how a job should be represented in the grid. NorduGrid has decided to base their job representation on the Resource Specification Language (RSL) from the Globus Toolkit.

```
1   & (executable=hellogrid.sh)
2     (jobname=hellogrid)
3     (stdout=hello.out)
4     (stderr=hello.err)
5     (gmlog=gridlog)
6     (architecture=i686)
7     (cputime=10)
8     (memory=32)
9     (disk=1)
```

Figure 2.1: XRSL job description specifying a simple job.

2.2.1 Resource Specification Language

As the name implies, RSL is meant as a language for specifying resources over jobs. However, the actual use of RSL is to specify which resource is requested along with a description of the job to be executed. In this way any brokering facility employed by the grid is able to find relevant matches for an RSL specification. Hence, RSL is used to specify not only how a job is composed, but also which resources are needed to execute the job.

RSL corresponds to a dictionary of attribute-value pairs, only that the set of available attributes is given by the specification. To enable additional attributes, NorduGrid has developed an extended version; appropriately named eXtended RSL, or simply XRSL. In brief, most of the extensions provided by the extended version facilitates the use of clusters as the unit of computation in NorduGrid, in contrast to the single machine unit of computation targeted by the original RSL. XRSL also specifies two different versions of RSL, namely user-side RSL and gm-side RSL. Users only has acquaint themselves with user-side RSL. Gm-side RSL is used internally in the grid manager only. For a complete reference of XRSL the reader is deferred to [14].

Figure 2.1 shows an example of an XRSL job description, using only simple constructs from the language. The first few lines (1 to 5) specifies which file is to be executed, what name the job should have in queues etc, the standard output and error pipes, and finally specifies that all job-related messages from the grid manager should be stored in a file called gridlog. The last four lines (6 to 9) provides help to the grid middleware by specifying some simple job characteristics. In Figure 2.1 the job requires an i686 architecture, is expected to run for approximately 10 seconds, and requires only 32Mb of RAM and 1Mb of disk space.

2.2.2 Job Management

To manage jobs in NorduGrid, the grid manager employs a state transition system. This system includes 11 states for managed jobs, including 3 pending states, and a number of transitions to link these states. Figure 2.2 on the following page shows the states along with possible transitions. The pending states are included because the grid manager can be configured with limits on the number of jobs in certain states, namely: `PREPARING`, `SUBMITTING`, `FINISHING`. Thus, if a job is temporarily denied a transition to a state due to these limitations, e.g. `PREPARING`, it is placed in a pending state called `PENDING:PREPARING`. Below is a short description of each "strict" state, i.e. excluding the pending states, and how they are used in NorduGrid version 0.4.x. [15]

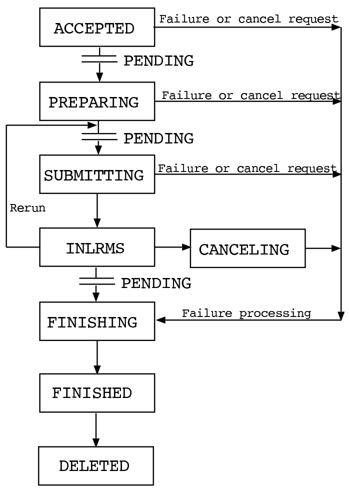


Figure 2.2: An overview of the job states in the current NorduGrid ARC grid manager and the possible transitions. It consists of 8 strict states plus 3 additional "pending" states. [15]

- **`ACCEPTED`**. In this state the job has been submitted to the grid manager and the job description is analysed. If the job description cannot be processed successfully the job will move to state `FINISHING`, otherwise to state `PREPARING`.
- **`PREPARING`** In this state the grid manager is downloading the input data specified in the job description. As data can also be uploaded from the submitting UI, the grid manager must also wait for these transfers to complete. In the case that the grid manager is unable to download any of the specified input files or an upload from UI takes too long time, the job will move to state `FINISHING`, otherwise to state `SUBMITTING`.
- **`SUBMITTING`** In this state the job is submitted to the LRMS. If the submission is successful the job moves to state `INLRMS`, otherwise it moves to state `FINISHING`.
- **`INLRMS`**. In this state the job is either in the queue of the LRMS or being executed at a number of nodes in the LRMS. The information system clarifies this by calling the state either `INLRMS:Q` or `INLRMS:R`. The grid manager waits for job execution to terminate, either successfully or failed - in both cases the job transitions to state `FINISHING`. If the grid manager receives a request to cancel the job, from e.g. a user, the job will transition to state `CANCELLING`.
- **`CANCELLING`** In this state the grid manager takes necessary actions to cancel the job in the LRMS. From here the job transitions to state `FINISHING`.
- **`FINISHING`** In this state the output data is being transported to the destinations specified in the job description, if the job terminated successfully. Thus the data is being uploaded to storage elements and possibly registered with replica catalogues. In the case of failed job execution no data are transfered. All files not specified as output are removed from the session directory of the job. From here the job transitions to state `FINISHED`.
- **`FINISHED`**. At this state the session directory is kept available for the user to download the output data. The job is in this state for typically 1 week before transitioning to the `DELETED` state.

- `DELETED`. At this state the job's session directory is deleted and only minimal information about the job kept at the grid manager.

The above states are relevant only for the current release of NorduGrid, namely version 0.4.x. According to the documentation already available on version 0.6 of the NorduGrid ARC, the grid manager will employ a richer state transition system to embrace the increasing number of supported LRMS flavours. Consequently, the `INLRMS` state will contain several *sub-states* besides the current `INLRMS:Q` and `INLRMS:R`. A new `UNKNOWN` state will also be added in version 0.6 to handle situations where the grid manager has become temporarily unavailable.

In version 0.6 of the NorduGrid ARC each state will furthermore have two different representations, namely an internal and an external. The internal representation will be used only in the grid manager when processing jobs, while the external representation will be used in the information system and the grid monitor. The reason for these different representations is to make the job concept more feasible to users, while enhancing the power of the job concept internally.

2.3 Architecture and Design

The NorduGrid tools are designed to handle every aspect of using, maintaining and administrating a grid. This includes job submission and management, user management, data management and monitoring. It should be noted that the supported computational resources in NorduGrid currently only include clusters running batch systems, e.g. PBS.

The three most important parts of the NorduGrid ARC architecture are:

- A client-server architecture for the resources, where each cluster runs a grid manager which is contacted directly by users of the grid that wish to submit jobs.
- A hierarchically distributed system, the information service, for indexing resources so that the users can discover, browse and use them.
- A client-server architecture for the data distribution in the form of file servers that are accessed with an extended version of FTP[16], GridFTP[17].

We first illustrate the architecture with an example of the task flow when submitting a job and afterwards describe the primary components of the design.

2.3.1 Job Submission Task Flow

A job submission scenario is depicted in Figure 2.3 where the numbers indicate the order of the tasks involved in submitting a job.

1. To enter NorduGrid, a cluster (or resource in general) must run three distinct services; a grid manager, an information service and a GridFTP server. These services are required to run on an Internet accessible front-end to the cluster. The information service collects local information and, as illustrated by the first arrow, registers itself with the index service to signal its participation as a resource in the grid.

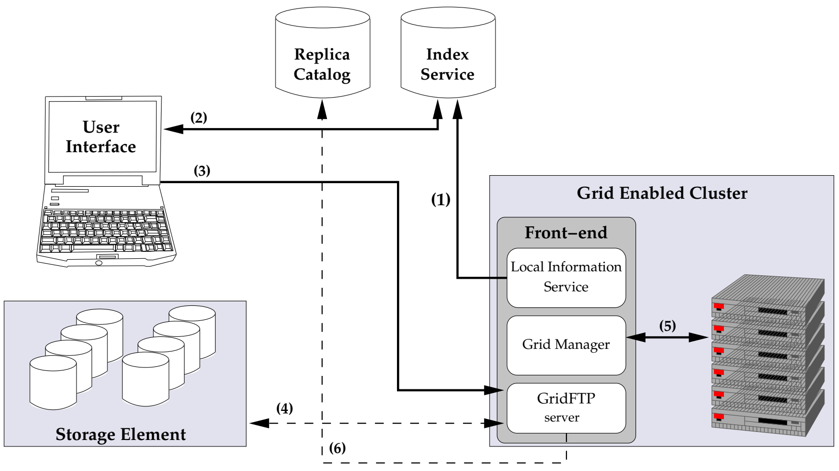


Figure 2.3: An overview of the NorduGrid ARC architecture. The bold lines indicate the flow of control and data in the operation of the grid. The dashed lines indicate optional data flow depending on job specific properties.

2. The user prepares a job description and submits it through the user interface. Before actually submitting the job, the user interface performs resource selection by matching the resource requirements specified in the job description with existing clusters in the grid. Any cluster that matches the requirements are eligible for processing the job and the user interface selects one based on scheduling policies.

3. Upon selecting the destination cluster where the job will run, the user interface uploads the job description to the selected cluster by means of the GridFTP server running at the front-end of that cluster. The processing and transferring of input data for a job will subsequently be referred to as staging input data.

4. The uploaded job description is detected and subsequently parsed by the grid manager running at the receiving cluster. The grid manager creates a session directory where all information and data pertaining to the job will eventually be stored. Depending on the job description the input data for a job is either fetched from a remote location, as illustrated by the fourth arrow, or uploaded along with the job description (step 3).

5. When all input data for a job is available in the session directory, the grid manager at the site submits the job to the local resource management system for execution. Currently NorduGrid supports a variety of such systems, e.g. OpenPBS, PBSPro, TORQUE, Condor and Sun N1 Grid Engine [18].

6. When a job has completed successfully the grid manager processes the output data according to the job description and transfers the result to a remote location, as illustrated by the sixth arrow, or simply leaves the output data in the session directory. The processing and transferring of output data will subsequently be referred to as staging output data. Finally, if requested by the user, the grid manager sends out a notification about the completion of the job, e.g. by email.

2.3.2 Grid Manager

The grid manager is the primary interface between the grid and the local resource. It runs on the front-end machine of every cluster in NorduGrid and handles incoming job submissions, including the pre- and post-processing of data related to the jobs.

The grid manager is implemented as a layer on top of the Globus Toolkit libraries and services. The grid manager provides the features missing from the Globus Toolkit for use in NorduGrid, such as integrated support for replica catalogues, sharing of cached files among multiple users, staging of input/output data, etc.

The grid manager software also contributes a job-plug-in to the GridFTP server in the Globus Toolkit. For each job the job-plug-in creates a session directory to hold all data pertaining to specific job, i.e. input files, output files and general job data such as exit status etc. Since the gathering of job and input data is performed by the cluster front-end in combination with a specific user interface, there is no single point that all jobs have to pass through in NorduGrid.

Users can either upload files directly to the grid manager or it can be instructed to download them on its own using a variety of protocols, such as HTTP(S), FTP, GridFTP, etc. When all input files are present in the session directory for a given job the grid manager creates an execution script that, besides executing the job, handles required configuration of environments for third party software and/or libraries. This script is submitted to the LRMS, e.g. OpenPBS, where it the job is queued and ultimately selected for execution at a subset of the nodes in the cluster.

After a job has finished executing, output files can be transferred to remote locations by specifying it in the job description, or alternatively, be left in the session directory for later retrieval by the user. The grid manager is also able to register files with replica catalogues should the job description request it.

2.3.3 Replica Catalogue

The replica catalogue in NorduGrid is used for registering and locating data sources. It is based on the replica catalogue provided by the Globus Toolkit with a few minor changes for enhanced functionality.

The changes primarily improves the ability to handle the staging of large amounts of input and output data for jobs and the adds the ability to perform authenticated communication based on the Globus Security Infrastructure mechanism. Objects in the replica catalogue are created and maintained by the grid managers running at each resource in NorduGrid, but can also be accessed by the user interface for resource selection.

2.3.4 Information System

The NorduGrid ARC implements a distributed information system which is created by extending the Monitoring and Discovery Services (MDS) provided by the Globus Toolkit. In NorduGrid there is an MDS-based service on each resource and each of these is responsible for collecting information on the resource on which it is running.

The MDS is a framework for creating grid information systems on top of the OpenLDAP software. An MDS-based information system consists of the following:

- An information model defined as an LDAP schema.
- Local information providers.
- Local databases.
- Soft-state registration mechanisms.
- Index services.

We describe each in turn in the following.

The **information model** employed by the original MDS is oriented toward single machines as the unit of computation and as such not well suited to describe the cluster-based approach of NorduGrid. The NorduGrid information model is a mirror of the architecture and hence describes its principal elements, i.e. clusters, users and jobs. The information about these elements is mapped onto an LDAP tree, creating a hierarchical structure where every user and every job has an entry. Replica managers and storage elements are also described in the information system although only in a simplistic manner.

Local information providers are small programs that generates LDAP entries as replies to search requests. The NorduGrid information model requires that NorduGrid specific information providers are present on the front-end of each cluster. The information providers interfaces with the local batch system and the grid manager to collect information about grid jobs, users and the queueing system of the grid. The collected information is used to populate the local databases.

The **local databases** are responsible for implementing the first-level caching of the LDAP entries generated by the information providers. Furthermore, they are responsible for providing the requested grid information used for replying to queries through the LDAP protocol. Globus includes an LDAP back-end called the Grid Resource Information Service (GRIS). NorduGrid uses this back-end as its local information database. The local databases in NorduGrid are configured to cache the output of the information providers for a limited period.

The local databases are registered as **soft-state** in the index services, which in turn can use soft-state registration in other higher-level index services. Soft-state dictates that resources must keep registering themselves periodically to avoid being purged from the information system.

The **index service** of NorduGrid is used to maintain dynamic lists of available resources. A record in the index service contains the LDAP contact URL for a soft-state registered resource. A user must then query the local databases at the resources for further information which reduces the overall load on the information system. The Globus developed back-end for the index service, the Grid Information Index Service (GIIS), uses a hierarchical topology for the indices, as illustrated in Figure 2.4, where local information providers register with higher level GIIS services which in turn register with the highest level GIIS services.

2.3.5 User Interface and Resource Selection

Users of NorduGrid interact with the grid through the user interface which has commands for submitting jobs, for querying the status of jobs and clusters and for killing jobs. Commands for managing input and output data on storage elements and replica catalogues are also included.

The user interface is also responsible for scheduling the job by choosing an appropriate cluster to run it on.

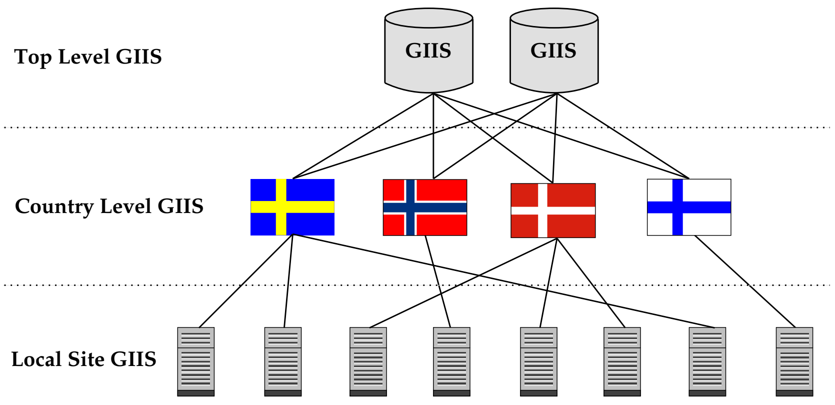


Figure 2.4: The hierarchical structure of the information system in NorduGrid. Higher level GIIS hosts should be replicated to avoid a single point of failure in the information system.

2.3.6 GridFTP and Storage Elements

GridFTP is the transfer protocol used for all data transfers in the NorduGrid ARC even for job submission. GridFTP is a modified FTP server provided by the Globus Toolkit as described in [17]. NorduGrid also provides a modified implementation which allows for access control based on user certificates.

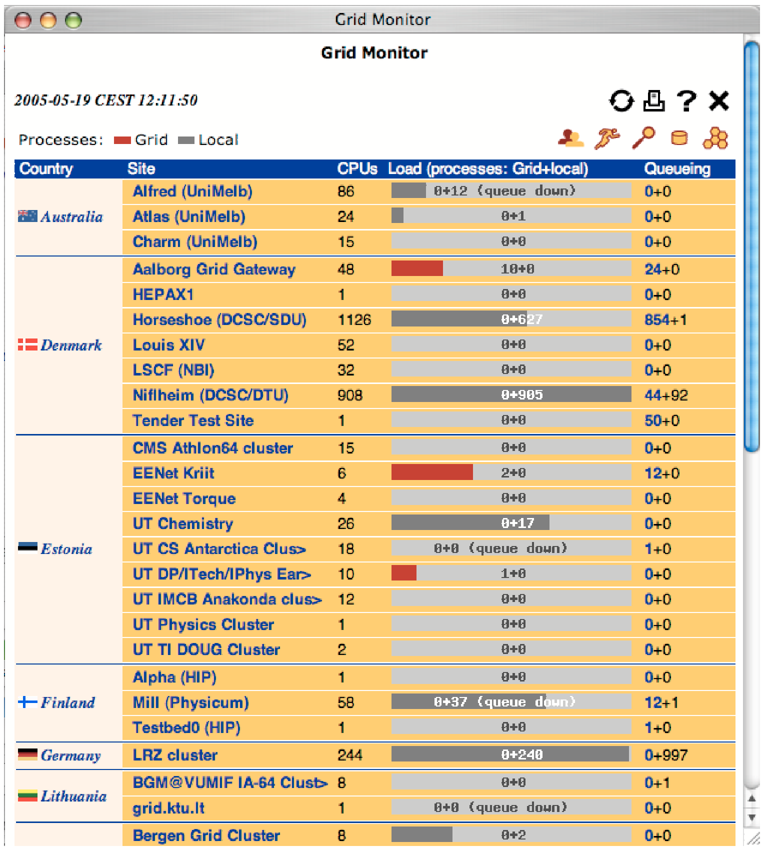
A storage element is a separate service responsible for storing data in the grid. In its current incarnation storage elements are mere GridFTP servers, but recently effort has been put into extending the capabilities of the storage element service. The NorduGrid Smart Storage Element (SSE) is supposed to be a replacement of the current storage element, will be based on standard protocols such as HTTPS, Globus GSI and SOAP, and will provide flexible access control, data integrity between resources and support for autonomous and reliable data replication [19]. A job description can specify that input data should be downloaded from a storage element.

2.3.7 Monitoring

NorduGrid provides a web-based monitoring tool for browsing the grid information system. It allows users of the grid to view all published information about the currently active resources.

The structure of the monitoring tool corresponds to the hierarchical structure of the information system itself. Hence, the initial screen of provides an overview of the entire grid, e.g. the number resources in terms of CPUs, the length of queues etc. The user can browse deeper into the hierarchy and at the lowest level inspect the queues, available software, hardware platform and so forth on a selected resource.

The web server that provides the grid monitoring tool runs on a machine independent from the grid itself. However, the web interface refreshes its information every 30 seconds by performing LDAP queries, which ultimately burdens the information system.



Grid Monitor

2005-05-19 CEST 12:11:50

Processes: 🟥 Grid ▨ Local

Country	Site	CPUs	Load (processes: Grid+local)	Queueing
Australia	Alfred (UniMelb)	86	0+12 (queue down)	0+0
	Atlas (UniMelb)	24	0+1	0+0
	Charm (UniMelb)	15	0+0	0+0
Denmark	Aalborg Grid Gateway	48	10+0	24+0
	HEPAX1	1	0+0	0+0
	Horseshoe (DCSC/SDU)	1126	0+627	854+1
	Louis XIV	52	0+0	0+0
	LSCF (NBI)	32	0+0	0+0
	Niflheim (DCSC/DTU)	908	0+905	44+92
	Tender Test Site	1	0+0	50+0
Estonia	CMS Athlon64 cluster	15	0+0	0+0
	EENet Kriit	6	2+0	12+0
	EENet Torque	4	0+0	0+0
	UT Chemistry	26	0+17	0+0
	UT CS Antarctica Clus>	18	0+0 (queue down)	1+0
	UT DP/ITech/IPhys Ear>	10	1+0	0+0
	UT IMCB Anakonda clus>	12	0+0	0+0
	UT Physics Cluster	1	0+0	0+0
	UT TI DOUG Cluster	2	0+0	0+0
Finland	Alpha (HIP)	1	0+0	0+0
	Mill (Physicum)	58	0+37 (queue down)	12+1
	Testbed0 (HIP)	1	0+0	1+0
Germany	LRZ cluster	244	0+240	0+997
Lithuania	BGM@VUMIF IA-64 Clust>	8	0+0	0+1
	grid.ktu.lt	1	0+0 (queue down)	0+0
	Bergen Grid Cluster	8	0+2	0+0

Figure 2.5: The initial window of the NorduGrid monitoring web site.

2.3.8 Security

The NorduGrid ARC uses the Grid Security Infrastructure (GSI) [20], available through the Globus Toolkit, as its security framework. GSI employs asymmetric cryptography as the basis for its functionality, and provides:

- Secure communication between elements in NorduGrid.
- Security mechanisms across organisational and institutional boundaries to ensure a consistent security scheme in NorduGrid as a whole.
- Single sign-on for users of NorduGrid via certificate proxies, allowing computations involving multiple resources and/or sites.

A central concept in GSI authentication is a certificate. Every entity in NorduGrid, e.g. a user, resource or a service, is identified via a certificate containing sufficient information to identify and authenticate the owner. GSI certificates are encoded in X.509 format [21], established by the Internet Engineering Task Force (IETF). The standardised format allows for the certificates to be used with other software employing asymmetric cryptography, such as web browsers, email clients, etc. A GSI certificate includes four primary pieces of information used for authentication purposes:

- The name of the *subject* for this certificate. Identifies the name of the user or object that this certificate represents.
- The *public key* belonging to the subject.
- The identity of the *Certificate Authority* (CA) that has signed the certificate. The CA is trusted and certifies that the public key does belong to the given public key.
- The *digital signature* of the named CA.

For further information about asymmetric cryptography, certificates, digital signatures and authentication the reader is deferred to [22].

GSI provides a *delegation* capability: an extension of the standard SSL protocol which reduces the number of times a user is required to enter his pass phrase. This is especially useful in the context of a computational grid as jobs can require multiple resources at possibly multiple sites. The delegation capability is provided via the use of proxies.

Proxies in GSI are certificates signed by a user, or by another proxy, to avoid providing a password in order to submit a job. They are primarily intended for short-term use, e.g. when the user is submitting many jobs and cannot be troubled to repeat his password for every job. In NorduGrid a user is required to create a proxy in order to submit jobs.

A proxy consists of a new certificate with a new public and private key. The new certificate is signed by the user that created it instead of the CA, which establishes a chain of trust from the CA to the proxy through the owner, as illustrated in Figure 2.6 on the next page. The subject of a proxy certificate is the same as the subject of the certificate that signed it, with `'proxy'` added to the name. The grid manager will accept any job submitted by an authorised user, as well as any proxies he has created.

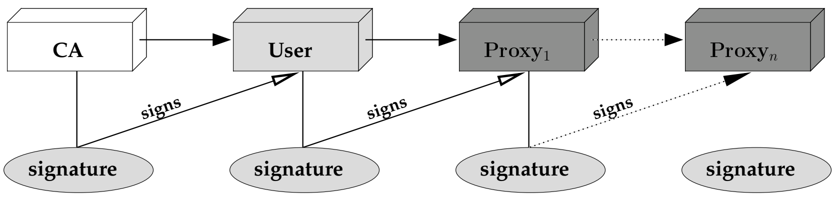


Figure 2.6: Delegation of trust through proxies.

Proxies provide the single sign-on feature, a convenient alternative to constantly entering passwords, but they are also less secure. Therefore, they are kept as user-readable only, and deleted after they are no longer needed (or after they expire). To create a proxy in NorduGrid a user simply runs the command `'grid-proxy-init'`. The expiration of the proxy can be regulated by specifying for how many hours the proxy should be valid.

2.4 Identified Problems

In the following, we summarise the identified problems in the current implementation of the NorduGrid ARC grid manager. To simplify discussion the problems have been divided into four categories, namely core, data moving, LRMS and GridFTP.

2.4.1 Core Specific Problems

The core component of the grid manager in NorduGrid is the primary driver for most grid services running at participating sites. As explained in Section 2.3 on page 15, the grid manager initiates the staging of input data upon detecting an incoming job description. When the job and its required input data are in place at the site the grid manager is responsible for executing the job, by submitting it to an LRMS. Finally, when a job completes, the grid manager is responsible for staging the output data appropriately.

In the process of job-handling the grid manager interfaces with many components of the NorduGrid architecture. The means to communicate with these components are fixed to either file based communication, e.g. checking for the existence of a certain file, parsing of log-files to extract information or simply regular method invocation. The current grid manager provides no means to tailor the means of communication between components to site specific needs, e.g. to allow for the grid manager to run distributed on multiple hosts by using some sort of RPC for communicating between components. Not essentially a problem, the lack of customisation inhibits the use of the grid manager unnecessarily.

With the current grid manager it is not possible to customise job phases/states, staging of job related data or customise the handling of jobs in other regards. Hence it is not possible to tailor the grid manager to site specific needs, e.g. to allow for specialised logging of jobs, to adapt the grid manager to local resources, bandwidth etc.

In general, the core component of the grid manager, in its current implementation, does not allow for customisation and extending the grid manager with extra features is not at all straightforward.

2.4.2 Data Moving Specific Problems

The data moving component is responsible for downloading and uploading data when requested by the grid manager. The data moving component supports several protocols: GridFTP, FTP, HTTP(S), HTTPg[1].

The data moving component employs a caching scheme to minimise the number of necessary downloads. Hence, if a request for a cached file is received, the cached copy will be used, instead of downloading the data again. If the request was for a writable file the cached file is copied to the requested destination, otherwise the data moving component creates a symbolic link to the file. [15, 23]

The data movement component is controlled by the grid manager. The grid manager can be configured to only allow for a maximum number of active data movement processes [24]. A data movement process uses several threads to perform and monitor the transfer. The number of threads can degrade performance, presumably due to excessive context switching between the data transfer threads and the other services provided by the front-end machine. An alternative to the current GridFTP solution exists for the data moving component, namely an Apache http-server with a Curl module. According to [25], this approach performs significantly better by sustaining both more stable and generally higher throughput.

The caching scheme employed by the data moving component only honours completed downloads. This means that if a data moving process is actively downloading a file, a, and is not yet completed, and another data moving process is about to download the same file a, then the new process will start from scratch to download the file.

2.4.3 LRMS Specific Problems

When the grid manager has prepared the input data for a job successfully it is ready to start the job. The working nodes in NorduGrid are running as part of a cluster and to manage jobs for that cluster an LRMS is used, e.g. PBS. For the grid manager to run a job it needs to know how to submit a job with the LRMS, e.g. know the appropriate commands for submitting a job.

The LRMS component provides an interface to the various LRMS flavours supported by NorduGrid, e.g PBS and Condor. The component provides a set of scripts for interfacing with the relevant LRMS. The following commands are provided for PBS, Condor and the simple *fork* which simply forks the job as a new process (i.e. the asterisk can be replaced with either `PBS`, `Condor` or `fork`):

- **`submit-*-job.sh`**
  Used to submit a job to the LRMS.
- **`cancel-*-job.sh`**
  Used to cancel a job that has already been submitted to the LRMS.
- **`scan-*-job.sh`**
  For gathering information about jobs, e.g. queue status, exit status, etc.

[1]HTTP over Globus GSI

The current implementation of the LRMS scripts is specialised toward various LRMS types. No code reuse or modularisation has been employed in developing the LRMS component. Hence, if a new LRMS is to be supported, new scripts for that LRMS have to be written completely from scratch.

As an example, consider the existing `scan-*-job.sh` script. This script parses log-files from the LRMS system to determine the exit status of a job. Consequently, the coupling between NorduGrid and the supported LRMS becomes so tight that a new version of e.g PBS, with different log-file formatting, will render the LRMS scripts to PBS unusable. Decoupling the grid manager from the various LRMS flavours does not seem possible at present, but a more robust solution is needed.

2.4.4 Job Submission Specific Problems

The GridFTP component handles incoming job submissions. The GridFTP server running at all NorduGrid front-ends has a job-plug-in installed to detect incoming jobs.

The job-plugin of the GridFTP server does not communicate directly with the grid manager, instead *file communication* is used, i.e when the GridFTP detects that an uploaded file is a job description it creates a session directory, and puts a status file containing the text "`ACCEPTED`". The grid manager continually polls the control directory to discover submitted jobs, i.e. a newly created status file containing "`ACCEPTED`". This polling is ineffective and introduces a significant delay from the job has been uploaded, until the grid manager discovers the newly uploaded job.

2.5 Summary

We have presented the current middleware of NorduGrid, comprising three major components; the grid manager, the user interface and the information system. This has provided insight into how these components interact and how a drop-in replacement of the grid manager can be realised. Furthermore, the identified problems in the present middleware have been presented and discussed. This section summarises these problems and relates them to the design requirements from Chapter 1 on page 3. Some of the problems directly violates the requirements, while others merely indicates dubious design decisions.

The existing implementation of the grid manager in NorduGrid is indeed usable, but many issues remain that needs to be addressed to attract more resources, developers and users to NorduGrid. For instance, several components in the current implementation of the grid manager employ a high latency communication scheme, typically file based communication, log parsing and polling. This directly violates the design requirement of *efficiency*, which dictates that another communication scheme should be employed. The overall performance characteristics of the grid manager, and the grid as a whole, can be improved by solving this.

The present version of the Globus Toolkit used in the NorduGrid ARC is a customised NorduGrid version, namely 2.4.3-16ng, a ~ 150 MB sized binary distribution. The Globus Toolkit aims to be a complete foundation for grid middleware, which unfortunately has resulted in a heavy-weighted distribution which is unsuitable for customisation [26]. This is especially unfortunate when the Globus dependency propagates to client utilities, i.e. the tools for submitting and querying the grid, as it is often very cumbersome to install in most client environments. This could be severely alleviated by moving from a GridFTP based submission interface.

Currently, the only way to submit jobs to the NorduGrid ARC is through the GridFTP server. Clearly FTP is designed for transferring files and therefore does not pose a very suitable interface for job submissions. No form of negotiation is currently possible between the resource broker, in the user interface, and the grid manager. Substituting GridFTP as job submission interface with some form of RPC would facilitate advanced possibilities for the grid while removing the Globus dependency from the client side. The use of GridFTP for job submission furthermore hinders implementation of the *advance reservation* feature requirement as it requires non-standard GridFTP commands to be supported.

The major problem with the current grid manager software is easily discovered when attempting to introduce new features or customising the current functionality. The present level of extensibility in the NorduGrid ARC is negligible. To extend or improve the grid manager involves re-writing a huge amount of code, with complicated interdependencies with the Globus Toolkit or other NorduGrid ARC code. Clearly, the NorduGrid ARC has been designed to be a final solution and is inherently unsuited for modification or plugging in extensions. This violates the design requirements of *extensibility* and *customisation*.

To facilitate an extensible NorduGrid ARC, and thus grid manager, it is necessary to enforce clearly defined interfaces for all interaction between components. In adhering to clearly defined interfaces components are allowed to be loosely coupled and concerns can be separated to responsible components only.

As a whole, the current implementation of the grid manager is inherently difficult to extend or modify, and the prospect of reconstructing the implementation seems to be a daunting task. Instead, we have chosen an alternative solution, namely that of designing an architecture from scratch that is capable of hosting the implementation of a new grid manager. This might appear as an even greater task than simply reconstructing the existing implementation, but the inherent violations of the design requirements in the current grid manager makes this stand out as the most viable solution. Another point, from the classic computer science, made by Parnas in [27] is that:

> *It is essential to recognize the identification of usable subsets as part of the preliminaries to software design. Flexibility cannot be an afterthought.*

In the following chapter we present an architecture designed to facilitate a grid manager implementation that satisfies the design requirements from Chapter 1 on page 3.

3
The Design of a New Architecture

This chapter documents the design of an architecture to enable a grid manager implementation that complies with the design requirements set forth in Section 1.3 on page 7. Initially we refresh the general pattern for the tasks that a grid manager must be able to perform. The general patterns motivate the fundamental concepts employed by the architecture.

After presenting the overall design of the architecture, the implementation is documented. The design elaborates on the components of the architecture and their interrelations, e.g. how they communicate etc. The section on the implementation of the architecture clarifies the subtleties of the architecture along with relevant decisions not strictly dictated by the architecture, e.g. the choice of programming language. Finally, issues of the architecture regarding trust are briefly outlined.

3.1 Modus Operandi

The general pattern for a grid manager is to have jobs submitted, facilitate the execution of the job and ultimately make the results of this execution available for the user that submitted the job.

The central concept is surely jobs. The information pertaining to the job must be encapsulated nicely while ensuring the persistence of the job, e.g. across a grid manager crash, power outage or similar. This also calls for a method to determine the current state of the job, to correctly recover and track its progress through the system. The processing of jobs, from one job state to another, should be customisable and each step in the process should be contained in separate module, preferably a plug-in. It should be possible to introduce custom steps into the processing, that perform site specific tasks.

The architecture should not dictate the method of job submission, neither protocol nor job description format. This calls for a way to customise the method whereby the grid manager receives external stimuli, or simply the way that external components communicate with the grid manager.

A grid should also provide information about the job processing, e.g. job state and queue position. This is typically stored in an information system. A grid manager should provide a way for this information system to interface with grid manager, in order to provide users with the needed job information.

3.2 The New Architecture

The architecture is a flexible and customisable event-driven framework based on plug-ins to perform the processing of jobs and monitoring of events. The architecture is centred around these concepts: *events, monitors*, *jobs, informers, handlers*, and *driver*.

- An *event* is used to contain a single occurrence in the system. Each event has a type and a data container to describe the event. An example would be a job submission event, its type is clearly job submission and the data associated with it could be a textual job description.
- *Monitors* create events in the system. They act as adaptors translating external occurrences into events. An example could be a job submission monitor that acts as the interface for job submission and introduces "*job submission*" events.
- *Jobs* is the central concept of the system, encapsulating a job description and current state of the job. Each job has a well defined state at any time and job state changes result in events.
- *Informers* provide the interface between the information system in the job information stored in the grid manager.
- A *handler* performs a well defined processing operation on a job, e.g. staging in the necessary data for a job. The entire job processing of a grid manager will consist of several handlers, each performing a portion of the job processing. Handlers can change the state of jobs as the processing of a job progresses. A handler will register to be launched on specific events. For instance a handler performing the initial processing of a job will register to be launched whenever a "*job submission*" event occurs.
- The *driver* is the "driving force" of the system. It receives the events from monitors and state changes and launches the appropriate handlers as a response.

In the following an example of a job submission using the architecture is presented. See Figure 3.1 on this page for an overview of the steps involved. The example shows an job submission in a simplified grid manager using the new architecture.

1. The external event of a job submission by a user, is detected by the submission monitor. The monitor creates a job submission event based on the job submitted by the user.
2. The driver is notified of the job submission event by the submission monitor. The driver has knowledge of the various handlers registered in the system and finds two handlers registered for the job submission event, namely "*Parse JD*" that processes the job description and "*Authorise User*" that decides if the user is authorised to run the particular job at this grid site.

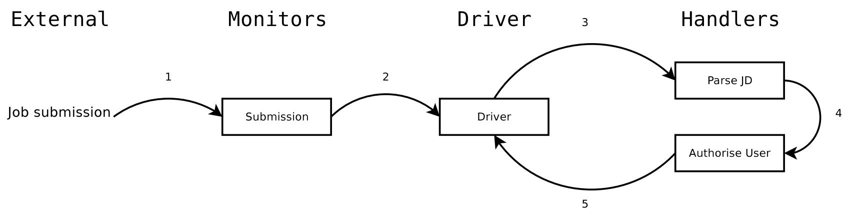


Figure 3.1: Example of a job submission in the architecture. Monitors, driver and handlers are known from the architecture. External represents components external to the system, that produce external events.

3. The driver executes the first handler. This handler reads the job description from the event.

4. The first handler terminates successfully and the execution of the second handler is begun. This handler verifies that the submitting user has the appropriate rights to execute the given job at the grid site.

5. The second handler terminates and the driver proceeds to handle the next unhandled event.

The architecture is tied together by a central component, the driver. The driver initiates the system, loading the monitors and handlers as plug-ins. When loading handlers the driver gains knowledge of which events that trigger the respective handlers. When the entire system is initialised the driver will get events regarding jobs from monitors and state changes, and trigger the appropriate handlers.

Given that multiple handlers can register for the same event, a system wide ranking of handlers for each event must be employed. The ranking dictates the ordering in which handlers are launched. Handlers with low ranks are run first and handlers with equal ranks for an event are disallowed.

Each event has a numeric priority that is used by the driver to handle multiple concurrent events on the same job. The driver receives an event e_1 that concerns job j, the priority of this event is p_1. Handlers h_1 and h_2 are registered to respond to event e_1. The driver responds by running the handler with the lowest ranking, in this case h_1. The driver now receives a new event e_2 with priority p_2 that also concerns job j. However, the first handler h_1 is still running and h_2 has not even begun yet. This is a case of simultaneous events for the same job, and the driver resolves this by comparing the priorities of the events.

- $p_1 \leq p_2$: The priority of the second event is greater than the priority of the first. In this case handler h_1 is allowed to complete, however h_2 is not run. Instead the driver stops further actions in response to event e_1 and begins responding to e_2.

- $p_1 > p_2$: The second event has a lower priority than the first event. Here the driver completes responding to event e_1, running both handlers h_1 and h_2 , before responding to event e_2.

3.3 Implementation

The following sections will document the implementation of the different components in the architecture.

3.3.1 Choice of Programming Language

Any implementation is of course done in a programming language, choosing the proper programming language is a crucial decision in a software development project. This project clearly calls for a high-level programming language, where performance has a lower priority than readability of source code. The reasoning behind this is that readability of both plug-in and architecture code has a high priority in aiding the extensibility and degree of customisation. Whereas the tasks performed by the grid manager is not expected to be a bottleneck in the processing jobs, rather the choice of mechanism for interaction with external components.

The considered languages are based on the implementors previous experiences, and constitutes the following: C++, Ruby, Python. The readability of C++ code is very varying and depending on the programmers understanding and use of the powerful template feature and the Standard Template Library. However using Globus from C++ is easy as the software is primarily written in C/C++. Ruby and Python are considered for their high readability, and one could easily argue that they can be viewed as competitors in the category "*best object-oriented interpreted scripting language*", if there ever were such a category. Which of the two languages comes out as the winner is clearly subjective and depends on the task at hand. However the availability of the pyGlobus[1] project with no known equivalent for Ruby, clearly favours Python when developing for NorduGrid that is heavily based on Globus software.

Python is the language chosen. Ruby lacks good Globus bindings and C++ can easily become too complex and has inferior readability when compared to the other alternatives.

3.3.2 Structure

A UML diagram of the implementation is shown in Figure 3.2 on this page. The driver is the central object that instantiates objects of the other classes, and thus initiates the system. All configuration of the architecture is stored in XML format, thus writing utilities that visualise, generate and manipulate configuration is simplified significantly.

3.3.3 Driver and Workers

The driver is central object that initiates the system. It starts the system by loading states, handlers, monitors and jobs from disk. Which handlers and monitors to load, along with the valid states for the system, is specified in the configuration file of the driver. The jobs loaded from disk are jobs already under the control of the grid manager, i.e. jobs already submitted and persistent from a previous invocation of the system.

After startup, the responsibility of the driver is to respond to incoming events by running the appropriate handlers. To accomplish this the driver has a pool of workers available. Workers

[1]Bindings to Globus API from Python

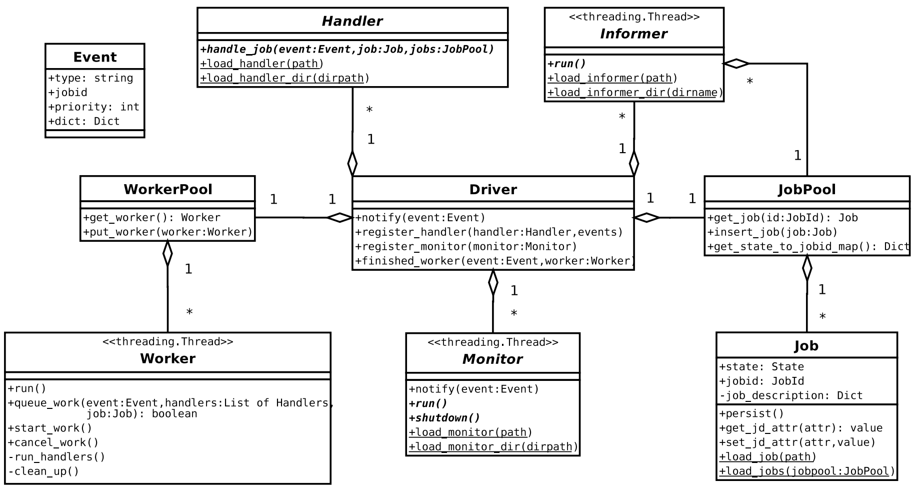


Figure 3.2: The class diagram for the architecture implementation.

are continuous threads that perform the actual running of handlers. The driver gets an idle worker from the worker pool, to which it assigns a workload. A workload consists of an event, an ordered list of handlers and a job object on which to perform the handlers. The driver can cancel the workload currently assigned to a worker, according to the priority rules, or append additional workloads to be performed.

The configuration file of the driver specifies the number of workers in the job pool, and thus the maximal number of concurrently running handlers in the system. An example of a driver configuration file can be seen in Figure 3.3 on the following page.

3.3.4 Jobs

Jobs are encapsulated in the `Job` class. The responsibility of the class is to represent the information given in the job description, record the current state of the job and ensure persistence of the job onto disk. Ensuring persistence of the job is crucial to the grid manager, as the grid manager must be able to handle failures. The job description is stored in a dictionary, mapping attributes to values, where values can be a single value, a list of values or even a dictionary. This structure can easily match the structure of e.g. an xRSL document as used in NorduGrid, see Section 2.2 on page 12.

The state of a job is stored in the property variable of the same name. Getting and setting is automatically guarded by mutexes, and setting invokes the `persist()` method. The `persist()` method provides the mechanism for ensuring persistence of the job by storing a string representation of the job object to file. The representation is created by the pickle and unpickle functionality of python, that serialises and deserialises objects into string representations. When saving a new job representation to disk, it is first stored in a temporary file. When the writing of the temporary file is complete, the temporary file is renamed to the filename of the now outdated job file. This ensures that in the event of failure, a valid job description always exists as the renaming of the temporary file is considered an atomic operation on modern file system.

```
<driver workers="5">
  <handlers>
    <dir>standard/</dir>
    <file>specific/custom.xml</file>
  </handlers>

  <monitors>
    <dir>monitors/</dir>
    <file>special/stuffy.py</file>
  </monitors>

  <informers>
    <file>snow/informer.py</py>
  </informers>

  <states>
    <state>
      COMING
    </state>
    <state>
      GOING
    </state>
  </states>

</driver>
```

Figure 3.3: An example of a driver configuration file. This file specifies that the worker pool consists of 5 workers, and that the driver should load all handlers in the directory `standard/` and a single handler from `specific/custom.xml`. Similarly it loads monitors from the directory `monitors/`, the file `special/stuffy.py` and an informer from the file `snow/informer.py`. Finally, it specifies the valid state names `COMING` and `GOING`.

3.3.5 Job Pool and Informers

The job pool is a simple container that keeps references to all known jobs in the grid manager. Handlers have a reference to the job pool to allow handlers to submit new jobs into the system by creating a job object and insert it into the job pool. It is also possible to retrieve jobs from the job pool by job id.

Informers provide a way for the information system to interface with the grid manager to obtain information about jobs. An informer is a plug-in loaded by the driver at startup, it is started as a new thread with a reference to the job pool. An implementation should override the `run()` method, where it can setup communication with the information system and respond to queries by using the job pool.

The `get_state_to_jobid_map()` method supplies the informers with job information of all known jobs, the method is available on the job pool. The result is a mapping from state names to job ids, that informers can use to get a list of job ids of jobs in the various states. To get specific information of a job, the informers can call the `get_job(jobid)` method which returns the job object with the given job id.

3.3.6 States

To provide an environment where local grid managers is allowed to have custom job states while still retaining compatibility with the grid they are part of, it is necessary to introduce three types of job state in the system. Namely: external states, internal states and handler states. The external states are states for jobs in the entire grid context, e.g. valid NorduGrid job states. Internal states are job state names that are local to the specific grid manager. Handler states are symbolic names for states used by each handler and may differ from handler to handler, e.g. `SUCCESS` and `FAILURE`.

Handler states decouple handlers from the internal states employed by the grid manager. This makes it possible to introduce new internal states for jobs or to perform an entire renaming of the internal state names, while remaining compatible with the other participants in the grid and reusing handlers with little effort.

There exists a mapping between the internal states of the grid manager to the external states of the grid, to decouple the two state types. Similarly each handler has a mapping from its own symbolic handler states, to the internal states of the grid manager. For a visualisation of these mappings see Figure 3.4 on the following page.

As an example of how the mapping can be used, consider a mapping from the handler state `SUCCESS` to the internal state `STAGED_IN`. The `STAGED_IN` state is mapped to the external state `READY_TO_EXECUTE`. Hence, a handler that has successfully staged the input data for a job sets the state of the job to be `SUCCESS`. The grid manager then picks up the event that a job has changed state, and perceives the job to be in the `STAGED_IN` state. If the grid manager is asked by an external entity, e.g. the information system, to report the state of the job it will reply with the external state of the job, namely `READY_TO_EXECUTE`.

The external states should be agreed upon by all grid managers in the grid, however the internal states as well as the handler states can be modified and adapted to suit local preferences and requirements.

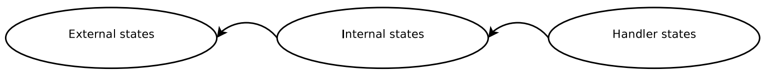


Figure 3.4: The three types of states in the system, an arrow indicates a mapping from origin to the head of the arrow.

```
<handler>

  <events>
    <event rank="3">STATE_LRMS</event>
    <event rank="4">EXT_LRMS_FAIL</event>
  </events>

  <statemap>
    <state internal="IN_LRMS">SUCCESS</state>
    <state internal="CANCEL">FAILURE</state>
  </statemap>

</handler>
```

Figure 3.5: An example handler manifest file. It specifies two events to which the handler should respond, namely `STATE_LRMS` and `EXT_LRMS_FAIL` with system-wide ranks 3 and 4, respectively. It also specifies a state mapping from handler states `SUCCESS` and `FAILURE` to internal states `IN_LRMS` and `CANCEL`, respectively.

3.3.7 Handlers

A handler consists of two things, a class inheriting from the abstract `Handler` class and a manifest file. The class implements the desired behaviour of the handler by overriding the `handle_job()` method. The arguments accepted by this method are the event that triggered the handler, the corresponding job object and finally a reference to the job pool. Most handlers will be handling jobs already in the system, this is the reasoning behind passing the job object along to the `handle_job()` method. However some handler must perform the actual submission of the job into the system, this handler will be given an empty reference to a job and must instead create a job object, from a job description, to be inserted into the job pool.

The manifest describes the characteristics of the handler, e.g. events that will activate the handler, the ranking of these events, and a mapping of handler states to internal states. This separation of behaviour and specification makes it easy to customise the system, e.g. reusing standard handlers in a system with custom states. An example manifest file can be seen in Figure 3.5.

3.3.8 Monitors

A monitor consist of a class inheriting from the abstract `Monitor` class, which inherits from `threading.Thread`. The implementing monitor has to override the `run()` method, in which it can setup the monitoring of events. To notify the system of an event, the monitor calls the `notify()` method, implemented in the `Monitor` class, with an event object.

Monitors are free to define the mechanism whereby they perceive the external events they wish to notify the system of. This could for instance be binding to network sockets and receiving notification of external events through some network protocol, e.g. XML-RPC, SOAP or simply pure HTTP(S). Another monitor could be watching the contents of a file or directory in a file-system notifying the system of changes. However, to avoid performance problems with such a monitor event based IO should be employed above polling whenever possible. Most operating systems provide mechanisms for event based IO, e.g. Linux provides `select` which listens for IO events in the kernel.

3.4 Trust, Plug-ins and Configuration

Given that the architecture is heavily extensible and customisable it is very possible to construct plug-ins that exhibit unwanted behaviour, either deliberately or unintentional. The power of monitors and handlers is so great that the plug-ins have to be trusted before being put to use at a grid site. It is the responsibility of the site administrator to gain this trust, e.g. by code review or similar. Unwanted behaviour can also be experienced with unfortunate configuration of the grid manager. It is also the responsibility of the site administrator to verify that the configuration, i.e. combination of handlers, events and monitors, performs as intended. This can be aided by using tools that visualise the event and job flow in the system, generated from the manifest and configuration files. Development of such tools would be an important step toward a successful deployment of the architecture.

3.5 Summary

This chapter has presented a software architecture for implementing a grid manager. Furthermore, a Python implementation of the architecture has been presented that further elaborates on the design. The implemented architecture embraces the need to design an efficient, extensible and customisable grid manager. The flexibility has been achieved by employing simple, yet powerful, decoupled components in the form of monitors, handlers and informers. The flow of the architecture is driven by events. Finally aspects regarding the trust and configuration dilemmas involved with the using architecture have been discussed.

The following chapter documents the design of an actual grid manager based on this new architecture. The grid manager is designed as a drop-in replacement for the current grid manager software of NorduGrid. This new grid manager demonstrates the power of the architecture.

4
A New Grid Manager

This chapter presents a prototype for a drop-in replacement grid manager for NorduGrid, implemented in the architecture presented in Chapter 3 on page 27. The grid manager is designed from a specification of a state-transition system defining the task flow for jobs. The core functionality of the new grid manager is contained in multiple handlers and two monitors.

Certain parts of the NorduGrid ARC software have been reused. This includes the user interface, the GridFTP server and job submission interface, and finally the local information providers, documented in Section 2.3.4 on page 17. The reuse of the information providers enables the new grid manager to be plugged into NorduGrid.

Initially we present the state-transition system to describe the job states in the new grid manager. The state-transition system is followed by a documentation of the design and implementation the handlers and monitors. In summary, we conclude the design of the new grid manager by reviewing its the functionality from a users the point of view and its ability to integrate with the current NorduGrid ARC.

4.1 Jobs

The job concept in terms of states and transitions currently employed by the NorduGrid grid manager needs a slight adaptation to suit the new architecture. However, experienced NorduGrid users should still be able to recognise the state of their jobs, even though it is being handled by the new grid manager. Hence, we propose new states and transitions for jobs to suit the architecture. The current states and corresponding transitions in NorduGrid are described in Section 2.2 on page 12.

The semantics of a state is to be modified from the existing states found in NorduGrid. In the current incarnation of the grid manager, job states are used to describe processes above actual states. As an example of the current state names of the NorduGrid grid manager, consider the state `PREPARING` which describes that input data are currently being staged for the job. In the new grid manager states are used in a more conventional manner, i.e. a state describes a stable situation. As an example the state `StagedIn` describes that all input data has been staged for the job. The transition system illustrating the states and possible transitions of the emulated NorduGrid grid manager is illustrated in Figure 4.1.

For the purpose of providing information to users it is still useful to describe the active processing of a job. Hence, the name of each job state can be postfixed with the name of the currently active handler. As an example, `StagedIn:LRMSSubmitHandler` would describe that all input data has been staged for the job, and that the job is currently being processed by a handler called `LRMSSubmit`.

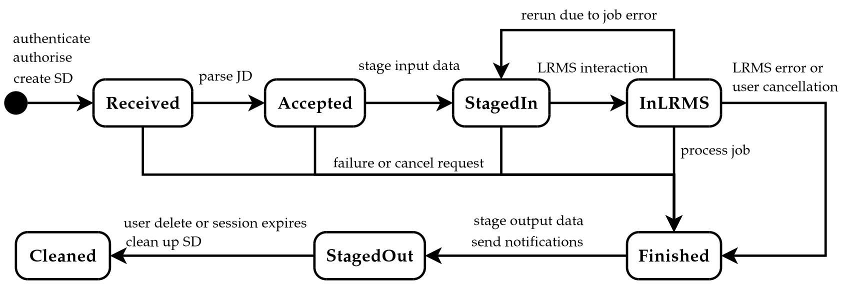


Figure 4.1: The job states of the new grid manager for NorduGrid. SD is an abbreviation of session directory, and JD is an abbreviation of job description.

- **`Received`** At this state the user submitting the job has been authenticated and authorised to allocate resources at the corresponding site. The session directory has been created and the job description uploaded.
- **`Accepted`** This state describes that the job description, located in the session directory, has been parsed successfully.
- **`StagedIn`** A job reaches this state when all its input data has been downloaded and processed.
- **`InLRMS`**. The job has now been submitted to the LRMS running at the site.
- **`Finished`** This state describes that the job has now finished executing inside the LRMS, either because it exited successfully or some error occurred. Note that this state acts also as a failure handler, i.e. if some failure occurs in any legal transition between states `Accepted`, `Prepared`, `StagedIn`, `InLRMS`, the job will transition to this state.
- **`StagedOut`** A job reaches this state when all output data has been staged out.
- **`Cleaned`** This state describes that the session directory pertaining to this job has been deleted.

4.2 Reused Components

In implementing the new grid manager both GridFTP and the implemented tools for the MDS-based information system from the current NorduGrid grid manager have been reused. It is possible to re-implement these components within the architecture presented in Chapter 3 on page 27, but in order to maintain backward compatibility with the NorduGrid user interface and information system these components have been reused.

Reusing GridFTP and especially the NorduGrid job-plug-in for GridFTP dictates the data and directory structure of the grid manager, as well as the file based approach. This essentially means that the emulating grid manager will be based on job descriptions in the XRSL language, employ a session directory and a control directory for storing job related data.

- **The session directory.** Contains all executable programs, input and output files related to a job. The executables and the input files are transfered either from the submitting host, via the user interface, or from a storage element somewhere in the grid. The output files are typically generated during execution of the job.
- **The control directory.** Contains all information about a job relevant to the grid manager only. In practice this involves four files. The control directory must be writable for the user who starts the driver.
  - **Description.** Contains the XRSL job description.
  - **Local.** Contains local batch system specific information for this job.
  - **Proxy.** Contains the proxy certificate used for submitting the job.
  - **Status.** Contains only the name of the current state of the job.

Reusing GridFTP furthermore has the benefit that authentication and authorisation is handled in this plug-in through the NorduGrid CA certificates and the grid-map file, which are used to authenticate users connecting to the GridFTP server. Hence, if the user submitting a job is not authorised to allocate resources at the site at which the the grid manager is running, he is never allowed to upload a job description.

The GridFTP job-plug-in developed as part of the NorduGrid ARC has been slightly modified to make the job submission process more effective. Specifically, the job-plug-in has been modified to send an event to the grid manager via XML-RPC in order to signal that a job submission has taken place. This is in contrast to the actual NorduGrid grid manager which scans the control directory for the existence of new status files to detect new job submissions. This modification is motivated by the requirement to avoid polling in the new grid manager.

Reusing the tools for registering the grid manager, and the associated LRMS, in the NorduGrid information system alleviates the need for developing a new LDAP client to handle this task. The tools consists of a set of Perl scripts extracting cluster info, queue info and job info by using the available commands in the LRMS or by scanning log files. The tools have been reused without modifications. The new grid manager appears in the NorduGrid web monitor interface along with all other sites running the original grid manager software and is eligible for job submissions from authorised users.

4.3 Handlers

In the following, the design of the handlers providing the core functionality in the new grid manager is documented. Each handler corresponds to a transition in Figure 4.1 on this page, or a part of a transition, and is described via the following template:

- **Events:** A description of which event(s) will trigger the activation of this handler.
- **Preconditions:** The conditions that must be satisfied for this handler to function properly.
- **Effect:** A description of the outcome of this handler, success or failure, along with potential side-effects.
- **Rank:** Describes the rank of this handler. Recall from Section 3.2 on page 29, that rank determines the order in which handlers registered for the same event are run.

Parse Job Description

This handler manages the submitted job description and extracts all necessary information in order to process the job, e.g. setup relevant environments, stage input and output data, etc.

- **Events:** A job made a transition to the `Received` state.
- **Preconditions:** A valid job object with a job id and the xRSL job description must be contained in the event.
- **Effect:** Success leads to the job transitioning to the `Accepted` state. Failure leads to a transition to the `Finished` state for this job.
- **Rank.** This handler should be the first one executed when a job enters the `Received` state in order to annotate the job with correct(attribute, value) pairs. Hence it is assigned rank 1.

To extract the job description this handler uses pyRSL, a yet to be released RSL parser, based on SPARK[28], written by Henrik Thostrup Jensen. The RSL parser has been slightly modified to support the extended attributes of both client-side and gm-side XRSL as specified in [14].

Stage Input Data

This handler provides functionality for downloading executables and input data for a job in the new grid manager by using the `ngcopy` tool from NorduGrid. In a complete implementation of the grid manager this handler should merely dispatch transfer requests to a separate IO subsystem. An example of a framework for such a subsystem will be presented in Section 5.2 on page 47.

- **Events:** A job made a transition to the `Accepted` state.
- **Preconditions:** A valid job object with a job id and a map of XRSL (attribute, value) pairs must be contained in the event.
- **Effect:** Success leads to the job transitioning to the `StagedIn` state. Upon failure this job transitions to the `Finished` state.
- **Rank.** Handlers performing pre-processing of the staged input data would wish to be run after this handler. Hence, this handler is assigned a rank 10, allowing other handlers to have both higher and lower rank.

Initially this handler ensures that all files specified as executables in the job description are in the session directory, and are indeed executable. In order to download the input files, the existing NorduGrid tool ngcopy is used. For each file to download the `ngcopy` tool is spawned as a subprocess of the handler, with source file and destination file as arguments. Hence, this handler supports the same protocols as `ngcopy`.

The `ngcopy` tool is part of the NorduGrid user interface and does not support specifying a proxy certificate to be used by GSI for the transfer. Instead, the environment variable `X509_USER_CERT` is used for every transfer. Fortunately, the proxy certificate of the user who submitted the job is located in the control directory, and thus handler simply points `X509_USER_CERT` to this proxy certificate.

Currently, this handler does not employ any of the advanced schemes required by a full fledged stagein handler. Thus no caching, scheduling, performance measuring etc. is implemented, as well as no limits on the number of concurrent downloads are enforced.

Generate Job Wrapper

This handler generates a wrapper based on the job description. The generated wrapper initiates all necessary means of communicating from the computing node to any monitors running at the site. Furthermore, the job description can request special environments and/or libraries to be available during run-time, the generated wrapper should see to fulfil such requests also.

- **Events:** A job made a transition to the `StagedIn` state.
- **Preconditions:** A valid job object with a job id and a map of XRSL (attribute, value) pairs must be contained in the event.
- **Effect:** Success in generating the job wrapper does nothing, but allows subsequent handlers to be activated. Upon failure this job makes a transition into the `Finished` state.
- **Rank.** This handler should be the first one executed for a job in the `StagedIn` state, hence it is assigned rank 1.

The wrapper is created as a Python program, that sets up necessary runtime environments, and spawns the job executable with appropriate arguments in a subprocess. The wrapper measures the running time and exit code of the executable. When the executable has completed, the wrapper sends an event containing the running time, exit code and the id of the completed job, using XML-RPC to the primary monitor. Thus, the handler requires that the host running the monitor is reachable from the computing nodes.

In the case that there is no connection from the computing nodes to the primary monitor, the communication can be based on file communication. The wrapper script should place the job execution information in the file. A monitor could poll the session directories of running jobs to discover that the job has terminated.

Submit to LRMS

This handler manages submission of the wrapper to the LRMS running at the grid site. NorduGrid currently supports several such LRMS flavours, however for the prototype of the new grid manager, supporting only PBS variants will suffice.

- **Events:** A job made a transition to the `StagedIn` state.
- **Preconditions:** An executable python program named `wrapper.py` must be present in the session directory.
- **Effect:** Success leads to the job transitioning to the `InLRMS` state. Failure leads this job to the `Finished` state.
- **Rank.** This handler should be the last to be executed for a job in the `StagedIn` state. Hence it us assigned a rather high rank, 100.

This handler assumes that the wrapper program is in session directory of the job. The wrapper program is submitted by spawning the PBS submission command, `qsub`, as a subprocess.

Once submitted to PBS, this handler does nothing to monitor the job. In a full fledged implementation a monitor should be developed to support this task. The PBS command `qstat` is typically used to show the status of jobs in queue and would be useful for such purpose.

Stage Output Data

This handler provides functionality for uploading the output data from a job to storage elements by using the `ngcopy` tool from NorduGrid. In a complete implementation of the grid manager this handler should merely dispatch transfer requests to a separate IO subsystem. An example of a framework for such a subsystem will be presented in Section 5.2 on page 47.

- **Events:** A job made a transition to the `Finished` state.
- **Preconditions:** A valid job with a job id and a map of XRSL (attribute, value) pairs must be contained in the event.
- **Effect:** Both success and failure leads to the job transitioning to the `StagedOut` state.
- **Rank.** The rank of this handler should allow for other handlers post-processing the output data before stageout. Hence this handler is assigned rank 10.

Initially this handler ensures that all files specified as output files in the job description are present in the session directory, and that the protocols and URL's are indeed valid. In order to upload the output files, the existing NorduGrid tool `ngcopy` is used. For each file to upload, the `ngcopy` tool is spawned as a subprocess of the handler, with source file and destination file as arguments. Hence, this handler supports the same protocols as `ngcopy`.

The handler employs the same method of changing proxy certificate for transfers as the stagein handler described in the above.

Currently, this handler does not employ any of the advanced schemes required by a full fledged stageout handler. Thus no caching, scheduling, performance measuring etc. is implemented, as well as no limits on the number of concurrent uploads are enforced.

Job Expiry

This handler is responsible for registering the session directory to be deleted after expiry of the job. The lifetime of a job is configurable, but is one week by default.

- **Events:** A job made a transition to the `StagedOut` state or the `Tick` event is received from the timer monitor.
- **Preconditions:** A valid job object with a job id must be contained in the event.
- **Effect:** Success, nor failure, has any immediate effect on the job. However, failure results in an error message to the site administrator, signalling that the expiry of the job could not be registered.
- **Rank.** This handler should be the last one to be executed for a job in the `StagedOut` state, hence it is assigned rank 100.

This handler register jobs that reach the `StagedOut` state for deletion, and receive `Tick` events from the timer monitor with a certain interval, typically 24 hours. A `Tick` event prompts the handler to check for expired jobs in its registry and deletes the session directory of these jobs. The registry of jobs in `StagedOut` is persisted onto disk, so that registrations remain in the case of crashes.

Notify User

This handler manages user notification. As currently practised in NorduGrid, users can request to be notified by email when their job reaches certain states. This is done by setting the attribute `notify` to an email address along with flags to indicate the states to notify in, in the job description.

- **Events:** A job made a transition to any state.
- **Preconditions:** A valid job object with a job id and a map of XRSL (attribute, value) pairs must be contained in the event.
- **Effect:** Both success and failure leads to the job transitioning to the `StagedOut` state.
- **Rank.** This handler should be one of the first to be executed when a job reaches a new state, hence it is assigned rank 2.

This handler connects to an SMTP server and sends an email notifying the user that a job with the appropriate job id has made a transition to its present state if it is one of those requested to be notified on by the user, e.g. `Finished.` Valid states to request notifications on are all those described in Figure 4.1 on page 38.

4.4 The Monitors

Primary Monitor

The primary monitor in the simplistic grid manager constitutes the external interface of the grid manager. It is implemented as an XML-RPC server acting on the following two types of incoming messages.

- **`job_submit_from_GridFTP`.**This message is sent from the modified GridFTP job-plug-in when a job has been submitted. Upon receiving this message the monitor fires an event to the driver signalling that a job has entered the `Received` state. The event contains the appropriate job id, the job description and path to the session and control directory respectively.
- **`wrapper_finished`.** This message is sent by the wrapper program from a computing node in the LRMS. Upon receiving this message the monitor fires an event to the driver signalling that a job has entered the `Finished` state. The event contains the appropriate job id.

These two types of messages all rely on an XML-RPC server being available, and thus they are all implemented in a single monitor. However, the architecture does allow for several monitors to be registered concurrently in the driver, thus enabling complex monitors to be developed as separate modules.

Timer Monitor

This is extremely simple monitor that simply fires a `Tick` event with a regular interval, in the standard configuration every 24 hours. The event is used to trigger job expiry handler to check for expired jobs, and thus the precision is in no way critical to the operation of the system.

4.5 Summary

This chapter has documented the design of a drop-in replacement grid manager for NorduGrid. The new grid manager is based on the architecture presented in Chapter 3 on page 27. The grid manager is only a prototype of a production grid manager, with the most basic functionality implemented. The grid manager is able to receive jobs submitted with the `ngsub` command from the NorduGrid user interface, and eventually submit them to the LRMS. Furthermore, stagein and stageout of job related data is supported via the `ngcopy` command. Finally, the user interface command `ngclean` is supported to enable the cleaning of a job upon a users request. Jobs controlled by the new grid manager can be monitored via the web-interface to the NorduGrid information system.

Section 1.3 on page 7 specifies that the new grid manager should facilitate the implementation of a set of advanced features. These features are all difficult to implement in the current NorduGrid ARC grid manager. To demonstrate the power of the architecture the following chapter presents how these features can be realised in the new grid manager.

5 Advanced Use Cases

5.1 Advance Reservations

This section will outline an approach of implementing advance reservations in the new grid manager. Advance reservations makes it possible for users to reserve resources in advance in the grid environment. This opens up for more advanced use cases for the grid, e.g. interactive use of resources, but can also provide a more predictable grid, where resource brokers can provide guarantees of job execution times. Co-allocation, i.e. reservation of several separated grid resources at the same point in time, is also made possible.

This approach is based on the work of [9], that shows how to implement advance reservations in the current NorduGrid ARC.

5.1.1 Previous Work

In [9] performing an advance reservation and submitting a job using the reservation is two separate steps. Reservations are performed using a reservation protocol implemented at the GridFTP server and client performing the reservation. Two commands “get reservation” and “release reservation”, `GERE` and `RERE` are the FTP commands respectively, are available:

- The first command, “get reservation” or `GERE`, takes the arguments: *nodes*, *duration* and *start time*, with an optional argument *project*. *Nodes* is the number of CPUs requested, *duration* the length of the reservation, *start time* is the start time of the reservation and *project* is an account to charge for the reservation.
  The result of the command has three possible cases:
  - `res_id` *start time*: the reservation request was successful, returns the reservation id identifying the reservation at the LRMS.

  - `failed` *next start*: the reservation failed, but *next start* is the next possible time at which the LRMS can hold the reservation.
  - `failed` *0*: the reservation failed and is never possible.

- The second command, "release reservation" or `RERE`, will release or cancel a reservation already reserved at the LRMS. The command takes a single argument *res_id*, which the identifier of the reservation to be released. The result of the command is one of two possible: "Reservation released" or "No such reservation", either the reservation was released or the identifier was unknown to the system.

A reservation is performed by querying the LRMS of whether it is able to guarantee the reservation. Thus advance reservations at a grid site require the use of a LRMS that supports this feature, e.g. OpenPBS with MAUI.

After the reservation is performed the user submits a job via GridFTP with an attribute in the job description specifying the reservation identifier from the "get reservation" command. The grid manager has two places where it needs to treat jobs with advance reservations specially, namely when the job changes state to `SUBMITTING` and `FINISHING`.

- When changing to `SUBMITTING` the grid manager checks that the reservation identifier is indeed valid and that the reservation has been performed by the user submitting the job, and while submitting to LRMS associate the job with the reservation identifier of the LRMS.
- When changing jobs to state `FINISHING` and submitted with a reservation, the reservation is cleaned at the grid manager and released at the LRMS. This is to guarantee that that a user can only use a reservation for one job.

5.1.2 Using Remote Procedure Calls

Extending the procedure of performing advance reservation from [9] to the new architecture, opens up new possibilities of interfacing when performing reservations. Generally the use of GridFTP for job submission and indeed performing advance reservations can be considered an awkward interface. Instead the use of remote procedure calls (RPC) is a much better suited interface. This is easily accomplished in the new architecture, where creating a monitor that acts as an RPC interface for the grid manager is trivial. Choosing a specific RPC solution is not a priority, as the solution should just support procedure calling and returning of results while performing marshaling, which can be considered the very essence of RPC.

There are at least three alternatives for creating the reservation at the LRMS:

1. Let the reservation be handled by a process external to the grid manager. This process would handle the negotiation from client to LRMS via RPC. When a reservation is successfully performed the process will notify the grid manager of this, associating a grid identity to the reservation ticket of the LRMS.
2. Let the reservation be handled by a monitor in the grid manager. No well defined communication path exists for letting handlers respond to monitors, and a reservation request

needs a response. Thus it is not optimal letting the monitor notify events for these methods and having the requests handled by a handler. The monitor must then perform or cancel the reservation with the LRMS, and respond the result as the returned value of the RPC.

3. Let the reservation be on par with a regular job in the grid manager, i.e. a job always leads to a reservation at the LRMS. Thus the job submitter will always have the start time of submitted jobs returned when submitting. This solution will probably have handlers performing the reservation and communicating with the submission monitor to provide the result to the user. This solution requires a major restructuring of the grid manager and is possibly the best solution.

For this use case it is chosen to go with the second alternative, it shows how to integrate advance reservations with the architecture in a proof-of-concept fashion. The approach to adding advance reservation to the grid manager thus involves adding reservation methods to the RPC monitor, assuming that such a monitor exists. To create a mapping to the approach from [9] two methods are added, namely `get_reservation()` and `release_reservation()`.

It is now possible performing and cancelling reservations with the LRMS at the grid site, however the grid manager does not know how to handle jobs with reservations. Three changes are needed to achieve similar functionality as [9]. Namely a job with a reservation needs to have the reservation verified (1), the submission to LRMS needs to know of the reservation (2) and finally the cleanup after the job has run needs to clear the reservation (3). The following will present a solution to achieving this in the grid manager presented in Chapter 4 on page 37:

1. Verifying the reservation can be accomplished by creating a new handler, to be launched after the “parse job description” handler. The “verify reservation” handler will check the job for a reservation reference, if one is found it must be verified. In the case that the reservation can be verified, or one was not present in the job description, the job is transitioned to the next state. If the reservation cannot be verified, the job is cancelled, i.e. transitioned to the Finishing state.

2. Making LRMS aware of the reservation when submitting. This can be accomplished by modifying the “submit to LRMS” handler to add the necessary information when submitting.

3. Cleaning up after the reservation. This can be accomplished by adding a new handler to the “Finished” state. The new handler must check the job for a reservation reference, if one is present it will release the reservation at the LRMS and clean up additional information recording the reservation at the grid manager.

This concludes the outline of a method for adding advance reservations to the grid manager presented in Chapter 4 on page 37. Functionality similar to [9] is achieved while extending this by supplying a RPC interface as an alternative to the awkward GridFTP reservation protocol.

5.2 An Advanced IO Subsystem

This section outlines the design of an architecture for an input/output subsystem to be used with a grid manager based on the architecture designed in Chapter 3 on page 27. The design of an IO architecture does not immediately qualify as an advanced use case of the grid manager architecture, however, it serves as an illustration of how such a system can be realised to support the architecture. Input/Output will henceforth be abbreviated IO.

The utilities currently available in NorduGrid for transferring input and output data consists of the DataMove API and the `ngcopy` command. The NorduGrid grid manager does not employ a separate service to manage how jobs stage their input and output data as they are embedded within the grid manager. The current grid manager of NorduGrid provides caching, replica management and control of the maximum number of allowed downloads.

5.2.1 Specification and Requirements

A versatile and efficient data management system for a grid manager should be able to cope with different needs that a grid enabled site might have. The designed IO architecture should be able to support generic data management needs and furthermore facilitate the implementation of an IO subsystem accommodating the following requirements.

- **Fault tolerance.** In a grid environment, many types of failure is to be expected, e.g. failed network connections and crashing clients, servers and/or storage systems. It cannot be expected of grid applications to handle all of these failure gracefully, instead care must be taken to hide these failures from the applications. An example could be to employ a kill-and-restart mechanism to avoid halted data transfers to hang forever, blocking other applications from using the IO subsystem.
- **Overload prevention.** A useful IO subsystem should allow for administrators to control the number of concurrent transfers from/to any storage system. Furthermore, space allocations and deallocations could be employed to ensure that necessary storage space is available before transfers are initiated.
- **Runtime adaptation.** Given the unforeseeable nature of failures in the grid, an IO architecture should provide means for the IO subsystem to adapt to failure of storage servers and performance variations during runtime. This can be achieved by allowing the subsystem to reconsider the choice of protocol and/or server, if alternatives are available, under conditions where data transfers fails repeatedly or performs poorly.
- **Support for heterogeneous resources.** It can not be assumed the all storage systems in a grid support the same set of protocols and communication mechanisms. The mechanism that applications use to access the IO subsystem should be independent of the type of both data source and destination, e.g. whether it is an FTP server, a file system or perhaps even a database.
- **Extensibility.** The IO architecture should facilitate the implementation of an IO subsystem that is easily extended to support new protocols and storage systems.

The above categories of data management needs as well as the features required by an IO subsystem constitutes the requirements to the designed IO architecture.

It is assumed that the IO architecture should provide a framework for an IO subsystem capable of transferring files between three different types of spaces, namely `gmspace`, `extspace` and `uspace`. Where:

- `gmspace.` Covers data locations that are available internal to the grid manager only, such as a cache.
- `extspace`. Covers data locations external to the grid manager, such as remote storage servers.
- `uspace`. Covers data locations local to the user, such as the session directory, or a workstation local to the user.

In a future IO architecture, possible extensions to this model could be considered, e.g. a DSM based IO architecture. But at the time of writing this sort of IO architecture still seems years away from appearing.

5.2.2 The Globus XIO Framework

Since version 3.2, the extensible Input Output framework (XIO) has been part of the Globus Toolkit. XIO is a simple and intuitive API for IO implementations, with integrated support for pluggable protocols, modularised file processing, integrated error handling and many more of the features that an IO architecture should ultimately support. The XIO framework has recently been used to extend Globus with support for multicast transmissions by Jaecle et. al. [29]. In the following we present the fundamental design of the XIO framework from [30].

The Globus XIO framework manages IO operation requests that an application makes via the user API. The framework does no work to deliver the data in an IO operation nor does it manipulate the data. All of that work is done by the drivers. The job of the framework is to manage requests and map them to the drivers interface.

Drivers

A driver is the component of Globus XIO that is responsible for manipulating and transporting the users data. There are two types of drivers, *transform* and *transport*. Transform drivers manipulate the data buffers passed to it via the user API and the XIO framework. Transport drivers are capable of sending/receiving data over a wire.

Drivers are grouped into stacks, i.e. one driver on top of another. When an IO operation is requested, the Globus XIO framework passes the operation request to every driver in the order which they are stacked. When the bottom level driver (the transport driver) finishes transferring the data, it passes the data request back to the XIO framework. Globus XIO will then deliver the request back up the stack in this manner until it reaches the top, at which point the application will be notified that the request is completed. The cooperation between the framework and the drivers is illustrated by the arrows in Figure 5.1 on the next page.

The transport driver is the one responsible for sending or receiving the data. Once this type of operation is performed it makes no sense to pass the request down the stack, as the data has just been transfered. Hence, there can be only one transport driver in a stack. When the transport driver has been run, and the data transferred, the operation is transferred back up the stack.

A transform driver is one that can manipulate the requested operations as they pass. In contrast with the single allowed transport driver, there can be many transform drivers in a driver stack. Some good examples of transform drivers are security wrappers and compression operations.

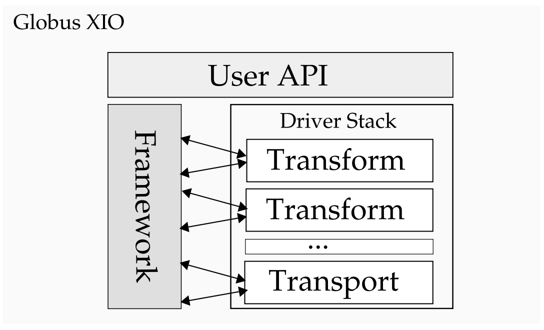


Figure 5.1: Overview of the XIO framework.

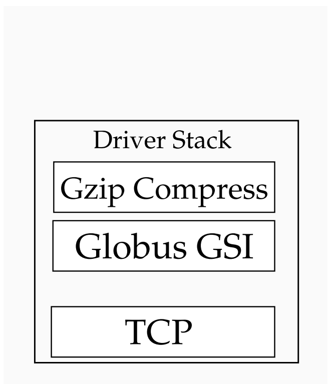


Figure 5.2: A real driver stack.

However, a transform driver can also be one that adds additional protocol. For example a stack could consist of a TCP transport driver and an HTTP transform driver. The HTTP driver would be responsible for marshaling the HTTP protocol and the TCP driver would be responsible for shipping that protocol over the wire.

Example

Imagine that a user has built a stack consisting of two transform drivers and, of course, a single transport driver. The transport driver is a TCP stack, and as required for all transport modules it is at the bottom of the stack. Above the TCP driver is the GSI transform driver which performs necessary messaging to authenticate the user and verify integrity of the data transmitted data. At the top of the stack there is a gzip transform driver, which compresses the file to be transfered. Figure 5.2 illustrates how the driver stack is constructed.

The first thing the user application will do after building the stack is call the XIO user API function globus_xio_open(). The Globus XIO framework will create internal data structures for tracking the operation and then pass the operation request to the driver at the top of the stack, namely the driver for gzip compression. The gzip driver needs a handle to a GSI connection to function, and therefore forwards the request to the next driver in the stack. The GSI driver has nothing to do either before the underlying stack has opened a handle so it simply passes the request down the stack. The request is thereby passed to the TCP driver. The TCP driver will then execute the socket level transport code contained within it to establish a connection to the given contact string.

Once the TCP connection has been established the TCP driver will notify the XIO framework that it has completed its request and thereby the GSI driver will be notified that the open operation it had previously passed down the stack has now been honoured. At this point the GSI driver will start the authentication processes (note that at this point the user does not yet have an open handle). The GSI driver now has an open handle to the TCP connection and performs several sends and receives to authenticate the connection. If the GSI driver is not satisfied with the authentication process it closes the handle it has to the stack below it and tells the XIO framework that it has completed the open request with an error. If it is satisfied it simply tells the XIO framework that it has completed the open operation. The gzip driver is now given a handle to the open GSI connection, if the GSI driver did not fail, and is able to initialise a gzip stream through that connection. Finally, the user is notified that the open operation completed, and if it was successful the user now have an open handle.

Other operations work in much the same way. When a user posts a write request, it is first delivered to the gzip driver, the gzip driver will wrap the buffer and pass it down the stack. The GSI driver will once again wrap the buffer and pass it down the stack. The framework will then deliver the read request with the newly doubly modified buffer to the TCP driver. The TCP driver will write the data across the socket mapped to this handle. When it finishes it notifies the framework, which notifies the GSI driver and subsequently the gzip driver. Neither driver has anything more to do so they notify the framework that the operation is complete and the framework then notifies the user.

Evaluation

Globus XIO is an extensible framework facilitating the implementation of new protocols in a pluggable fashion. However, it falls short of devising any means to achieve overload prevention and runtime adaptation. Furthermore, the approach of implementing low level transfer mechanisms, separated from higher level protocols, seems to be superfluous in a grid setting where a TCP connection is expected to be the superior choice in most cases.

5.2.3 Stork

In the following we briefly review the design of Stork, a scheduler for data placement activities in grids. For a complete reference on Stork the reader is deferred to [31, 32]. The declared goal of Stork is to make data placement a first class citizen in the grid, meaning that in Stork, data placement tasks are treated as if they were actual grid jobs. This does not imply that Stork does not distinguish regular jobs from data jobs, however, the latter is queued, scheduled, monitored and managed in a fashion similar to that of regular jobs.

As with regular jobs in a computational grid, Stork jobs are specified by a job description. Stork job descriptions are written in the ClassAd Language[33], and encapsulates more than just moving data between two locations. A Stork job is a combination of the following kinds of tasks.

- **Reserve.** The job description of a data placement job allows a user to specify how much storage space is needed for input data to a computational job. If the job description contains such a specification the reserve task is run preliminary to the staging of input data. This guarantees that enough space is available at the site of execution before any downloading is performed. Only some storage systems support this mechanism.
- **Release.** This task is executed when reserved storage space is no longer needed.
- **Transfer.** Any data placement job in Stork requires the transfer of data from one location to another. This task is used to perform data transfers between a remote site and a local site.
- **Locate.** If supported by the storage system, this task can be used to locate a specific file.
- **Stage.** This task is similar to the transfer task, except that it is used to move data between two remote sites.

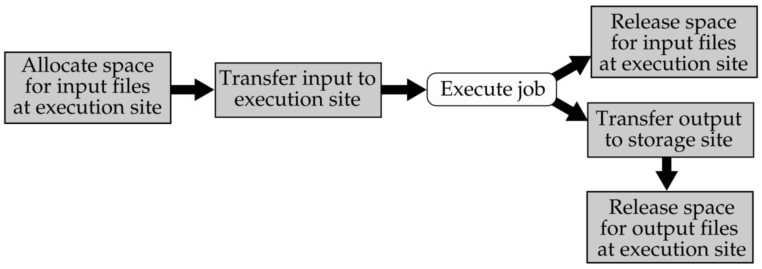

Allocate space for input files at execution site
Transfer input to execution site
Execute job
Release space for input files at execution site
Transfer output to storage site
Release space for output files at execution site

Figure 5.3: An example of a DAG based on a typical grid job. The grey boxes represents data related tasks while the white box represents job related tasks.

Listing 5.1: A Stork task specified in the ClassAd language.

```
[
  dap_type = "transfer";
  src_url = "file://$HOME/data/foo.dat";
  dest_url = "ftp://tender.grid.aau.dk/tmp/bar.dat";
]
```

Stork is not a self contained IO framework. To be fully functional it depends on several components from the Condor project, namely the Directed Acyclic Graph Manager (DAGMan), MatchMaker and the ClassAd language. The Condor project is an effort to develop, implement, deploy and evaluate mechanisms and policies for High Throughput Computing (HTC)[34].

The DAGMan is used by Stork to represent a job as a directed acyclic task graph. An example of a Stork job, represented as a DAG, is shown in Figure 5.3. The MatchMaker component is used by Stork to decide which transfers to schedule, and especially between which sites to transfer data – if multiple possibilities exist. Storage servers and Stork jobs are both specified using the ClassAd language. The MatchMaker knows the storage servers in the grid and it receives job descriptions from Stork, and is therefore able to *match* servers with appropriate data placement jobs.

Listing 5.1 shows an example of a Stork task specified in the ClassAd language. Performing the task transfers the local file `foo.dat` to `tender.grid.aau.dk` where it is stored as `foo.dat`.

Evaluation

The Stork scheduler is an interesting framework for an IO subsystem. The design decision of considering data transfers as regular jobs is compelling in a grid context because a multitude of mechanisms for queueing, scheduling and executing such jobs already exists. Stork embrace the fundamental characteristics of data transfers, and illustrates how most of the requirements to the IO architecture can be taken into consideration. However, as a negative point, the Stork framework has been designed as part of the Condor project and thus depends on several components from Condor.

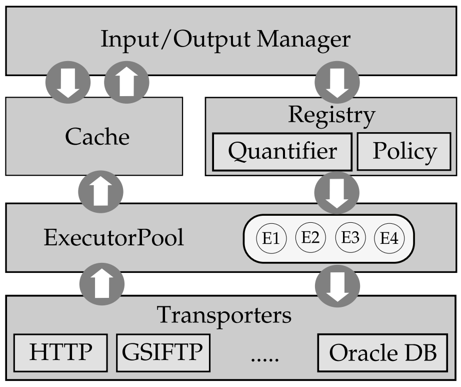


Figure 5.4: An overview of the IO architecture. The arrows indicate the communication interfaces between components.

5.2.4 The New Architecture

In Subsection 5.2.1 on page 48 several requirements for the IO architecture was set forth. Reviewing two existing technologies, the Globus XIO library and the Stork IO scheduler, has clarified how some of these requirements can be implemented.

In designing an IO architecture the choice is now whether to reuse the design of the reviewed frameworks or to design a new IO architecture using them merely as inspiration. Reusing the available frameworks is appealing as it minimises the effort necessary to design and implement the new architecture. However, designing a new IO architecture, using the available technologies merely as inspiration, allows for a completely customised IO architecture that integrates seamlessly with the grid manager architecture proposed in Chapter 3 on page 27.

Reusing Globus XIO is ruled out as the framework it does not provide enough functionality to warrant making the IO architecture Globus dependent. It does however illustrate the use of pluggable protocol handlers which would enhance the extensibility of the IO architecture. Reusing the Stork seems compelling. However, the framework depends on Condor components and dictates the use of the ClassAd language to describe data transfers, reducing the ability to support heterogeneous resources.

Based on these considerations we choose to design a new IO architecture based on the specification of requirements in Subsection 5.2.1 on page 48, using the Globus XIO library and the Stork framework as inspiration.

An overview of the designed IO architecture can be seen in Figure 5.4. The architecture consists of multiple components, facilitating the required functionality, which will be elaborated in the following. At the top of the architecture resides the *IO manager*, which is the internal interface of the architecture. The external interface is configurable, like the grid manager architecture was, through the use of monitors. Recall that monitors are part of the grid manager architecture presented in Chapter 3 on page 27.

Upon receiving a request to download a file, the IO manager looks up the file by interfacing with the *cache* component. If the file is found in the cache, the file is copied to its destination immediately. The benefit of this is that requests pertaining to cached files are honoured immediately. It is expected that the majority of download requests require only readable permissions to the file in question and therefore merely linking to the cached file will suffice.

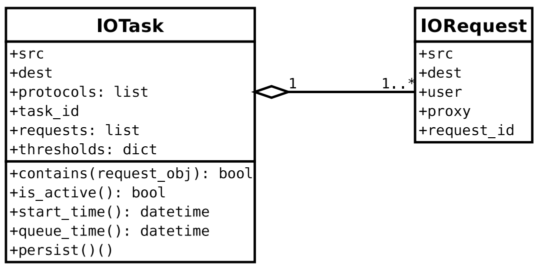


Figure 5.5: A rudimentary UML design for IOTask and IORequest.

If the file is not cached, the download request is added to the *registry*. This introduces a new concept into the IO architecture, namely an *IO task*. The purpose of an IO task is primarily to encapsulate transfer requests, and of course other information useful to the registry. The registry contains only IO tasks. Upon receiving a request the registry tries to find an IO task encapsulating a similar request. If such a request is found in an IO task the new request is added to that same IO task. A rudimentary UML diagram, illustrating the design of an IO task and an IO request, can be seen in Figure 5.5.

The mapping between an IO task and multiple transfer requests allows for the IO architecture to avoid multiple concurrent downloads of the same file. When an IO task has completed, the IO manager is notified via an event and consequently marks all corresponding requests as completed. When a request is completed some sort of notification must be sent to the entity from which the request emanated in the first place, signalling the completion of the request.

To perform the actual IO operations, and download the requested file, the architecture employs a set of *executors* collected in an *executor pool*. Whenever an executor is available, the IO manager requests an IO task from the registry by interfacing with the *policy*. The primary responsibility of the policy is to choose an IO task from the registry. The policy has a list of currently executing IO tasks, a list of pending IO tasks and a *quantifier* available to assist it in making qualified decisions about which IO task to start.

The quantifier is a component assigning a numerical value to a URL. The value represents a fitness function for the URL. The numerical value can be based on several measures regarding the URL, or several URL's, of an IO request. Such measures could be based on e.g. the distributed coordinate system Vivaldi [35], or more simple measures such as the registered bandwidth, round-trip time (RTT) and so forth.

Finally, when an executor is available to the IO manager, along with an IO task, as selected by the policy, the executor performs the data transfer by using a *transporter*.

Transporters can be considered a higher level of abstraction of protocols. They utilise one or several protocols to send/receive data between two uniform resource locators, possibly manipulating the data. The protocols supported by a transporter are specified in its corresponding manifest. When an executor is about to perform a transfer it selects an appropriate transporter by comparing these protocols with the ones specified in the IO task at hand. Transporters furthermore contain a set of *inspector capabilities*. Recall that an IO task contains a dictionary of thresholds, which specifies that e.g. the speed of a transfer should not drop below 50 kilobytes pr. second. The inspector capabilities of a transporter indicates exactly which type of thresholds it is able to inspect. Most transporters will be able to support inspecting the speed of a transfer, while other thresholds might be limited to certain specialised transporters, e.g. the number of records downloaded pr. second will probably be limited to transporters working on databases. The inspector capabilities of a transporter is specified in its manifest.

5.2.5 Summary

This section has documented the design of an IO architecture to be used with a grid manager implemented in the architecture proposed in Chapter 3 on page 27. Initially, a specification of an IO architecture was presented along with requirements to such an architecture. To draw on experiences from similar projects the design of the Globus XIO framework and the Stork data placement framework was reviewed. Finally, the resulting architecture was presented.

We conclude the design by evaluating how the architecture accommodates the initial requirements.

- **Fault tolerance.** This requirement is accommodated by the architecture by allowing the registry to store both IO tasks and IO requests onto disk using the `persist()` method on the IO task. Hiding failures from applications is the responsibility of the transporters. The inspector capabilities of these can be exploited to support automatic restart of failed or halted transfers.

- **Overload prevention.** The policy enables an IO subsystem based on the designed architecture to control the terms and conditions under which data transfers are started, stopped or restarted/resumed. Furthermore, the number of executors in the executor pool provides a means to control the number of active data transfers.

- **Runtime adaptation.** The inspector capabilities of the transporters enables an IO subsystem based on the designed architecture to adapt during runtime by monitoring active transfers and respond accordingly. Furthermore, if multiple protocols or storage resources are specified for an IO task, failure using one of the protocols or storage resources can be responded to by changing to another protocol or storage resource. The quantifier, assigning numeric values to URL's, furthermore allows the policy to adapt to current performance conditions.

- **Extensibility.** The extensibility of the design is illustrated by the transporters. The transporters are implemented as plug-ins to the architecture.

- **Support for heterogeneous resources.** The architecture does not directly employ any means to support heterogeneous resources, but neither dictates anything that diminishes this support.

The above evaluation of the designed IO architecture is based purely on the *design* of the architecture. An actual implementation of an IO subsystem, based on this architecture, will most certainly result in a design that does not accommodate the initial requirements as well as the good intentions behind the architecture.

5.3 Live Upgrades of the Grid Manager

This section outlines a method for upgrading a running grid manager *on-the-fly* with no downtime experienced from users, known as live upgrading. Live upgrading is useful for increasing the availability of a grid site. The alternative of shutting down the grid site when making changes to the grid manager software, leads to an unreliable grid service.

Live upgrading is divided into two cases that are handled differently, namely minor and major changes in the grid manager. Minor changes are:

- **Reloading a plug-in**. E.g. handlers and monitors. This could be desirable in the case where a new version of a plug-in is available with similar functionality, fixing errors or performance problems.
- **Loading a new plug-in.** For instance introducing a new monitor for handling job submission.
- **Disabling a running monitor.** This could be desirable if the grid site is performing a scheduled shutdown and thus wants to disable the submission of new jobs for a period of time.
- **Changing certain parameters of the system**. E.g. changing the number of workers or ranking of handlers for certain events.

All the listed minor changes are possible to achieve while the grid manager continues to process events and jobs. However a mechanism for achieving this in the current implementation of the new grid manager is not yet available.

Examples of major changes are:

- **Changing to a new version the grid manager software.** This could be a new version that introduces bug-fixes or new functionality to the grid manager.
- **Changes to the fundamental structure of the grid manager.** E.g. a migration to a incompatible setup of states, handlers and monitors.

Major changes cannot be achieved in the same way as minor changes, the grid manager process must be stopped at one point while another is started, to ensure consistent handling of jobs. The following will present an approach for reloading the grid manager software at a grid site without users experiencing the transition, e.g. no downtime and no change in the job processing from users perspective. The scenario is the following, a running grid manager gm_1 is about to be replaced with a new gm_2.

- The job submission monitor of gm_1 is brought down while bringing up gm_2 which will handle all new job submissions to the grid site. Similarly the informers of gm_1 are shutdown and replaced by the informers of gm_2.
- The jobs already being processed by gm_1 are allowed to be processed to a final state within the setup of gm_1, to ensure that they are handled consistently.

However care must be taken with shared resources between the two grid managers. Shared resources include the LRMS, previous job information but also unsharable resources on the computer system running the grid managers, e.g. network ports etc. Thus monitors binding to network ports have to be coordinated between the two running grid managers, which can prove to be quite the conundrum. The sharing of previous job information, i.e. persisted jobs and informers feeding the information system needs to be handled by the grid manager.

The informers of gm_2 will replace the informers of gm_1, and the system will proceed with two job pools for the duration that gm_1 continues to execute. The job pool of gm_1 will continue to "own" the jobs currently and previously processed at gm_1, however this does not prevent the job pool and informers of gm_2 to provide information to the information system. The job pool instances simply brand each persisted job file with the process ID of the grid manager and a unique version number. The version number is used to identify the format of the persisted job data, in the case that changes to the format are made. New versions of the software need to provide functionality to read the formats of previous versions. When the job pool of gm_2 loads persisted jobs from files, it notes the ones not "owned" by gm_2. These jobs not "owned" by gm_2 are treated differently, as they are always loaded from disk when informers needs to supply the information system, possibly employing some caching scheme to enhance performance. Recall that any changes to the information pertaining to a job are immediately stored onto disk in a manner that is considered atomic. Thus the information displayed by informers at gm_2 about jobs currently processed at gm_1 will be up to date. The jobs "owned" by gm_1 will only be modified by handlers at gm_1.

This concludes the outline of an approach to achieving live upgrades to the new grid manager. The problem is transformed into the task of coordinating the execution of two concurrent grid managers on a single system.

5.4 Distributing the Grid Manager

This section will present the reasoning behind distributing the grid manager and a possible solution for achieving this with the new architecture.

At a grid site with a huge body of computing nodes, it can be very expensive if the computing nodes are not utilised maximally corresponding to the extent of available user jobs. This could easily become the case if the grid manager fails, either due to software or hardware failure, which results in no grid jobs being run at the cluster or resource. Thus it is a desirable feature to distribute the grid manager onto several independent computer systems, which would provide a grid manager service that is resistant to failures at some of the systems. The motivation for this distribution of the grid manager functionality is to increase the availability of the grid manager service.

An example of the distributed architecture is shown in Figure 5.6 on the following page. It is based on the idea of distributing jobs to semi-independent grid manager instances running at different computer systems. The grid managers share some resources, namely the IO subsystem, LRMS and a file system. In order for the IO subsystem to perform successful scheduling of the network resources available at the system, it should be kept as a separate service shared by the grid managers.

The load balancing is performed on a network level, i.e. load balancing functionality of a switch or a round-robin DNS approach. This means that clients are oblivious to the fact that job submissions or information system requests are distributed to the available grid managers.

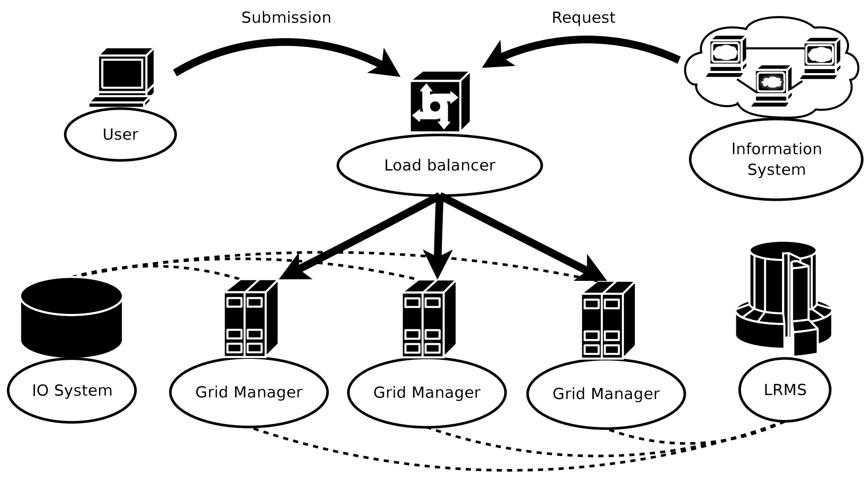


Figure 5.6: The distributed grid manager, consisting of three systems running the grid manager. IO subsystem and LRMS are still centralised components. User requests are load balanced by a specialised component, e.g. round-robin DNS or similar.

The grid managers operate on a shared file system for storing persisted job data. This makes it possible for informers running at any of the grid managers to reply to information system queries, by making the job pool query persistent jobs in the file system. This is similar to the approach for handling multiple grid managers when performing live upgrades in Section 5.3 on page 56.

6
Evaluation

This chapter will present an evaluation of the new grid manager for the NorduGrid ARC and the architecture it is based upon. Following the evaluation is a discussion of the future outlook for NorduGrid with the addition of the new grid manager.

Recall the requirements for the new grid manager listed in Section 1.3 on page 7, namely that the grid manager should be *efficient*, *extensible*, *customisable* and *fault-tolerant*. These four terms will be used to evaluate the new grid manager and architecture.

From this point on, the current NorduGrid grid manager is abbreviated CGM and the new grid manager NGM.

6.1 Efficiency

The efficiency requirement states that the grid manager should whenever reasonably employ the best performing approaches when processing jobs. To evaluate the efficiency of the grid manager, a performance test of the current NorduGrid ARC grid manager and the new grid manager is performed.

The performance test is designed to measure the minimal overhead added by the two different grid managers. No files are to be transfered during stagein or stageout, i.e. the test shows the minimal processing time for the most simple job in both grid managers respectively. The setup used for the tests is the following:

- The job is the execution of `/bin/date`, i.e. the standard UNIX command for displaying the current time of a system. The output acts as a witness to the execution time of the command.

	CGM	NGM
$S \rightarrow R$	38.4 s	1.0 s
$S \rightarrow F$	130.4 s	1.0 s

Table 6.1: Results for 1 submitted job. $S \rightarrow R$ is the time from job submission until the job is run by LRMS. $S \rightarrow F$ is the time from job submission until the job reaches the `Finished` state in the grid manager. The granularity of the test was 1 second and all tests were repeated 5 times, the results are the average of the 5 test runs.

- The jobs are submitted to a system running GridFTP from NorduGrid ARC version 0.4.5 and the CGM used is also from this distribution. The LRMS system used is TORQUE version 1.2.0p3, the only node in the "cluster" is the system itself. The operating system used is Debian Linux 3.1 running in a Xen[36] domain with 640 MB memory. The system sports an Intel Pentium 4 CPU running at 2.80GHz.
- The job is submitted from the system running the grid manager. This simplifies the measurement of time, compared to submitting from a different system, and also reduces the overhead of network communication for GridFTP.
- The system is otherwise idle, no other jobs are resident in the grid manager or the LRMS.
- The test is performed for a single job and for a simultaneous submission of 25 identical jobs.

The results for a single job submission are shown in Table 6.1. In all test-runs the elapsed time from submission to execution, and from execution until the job reaches the `Finished` state, were measured to be 1 second for NGM. However, the granularity of the test was also 1 second, thus to achieve a more precise measurement of NGM a finer granularity is needed. The results of the tests are nevertheless extremely clear, the average time from submission to execution of the job is 38 times greater for the CGM and 130 times greater from submission to the `Finished` state. This is undoubtedly due to the use of polling in CGM, this conclusion is further supported by the variance of the results of CGM. The elapsed time from submission to execution was in the interval $[2, 85]$ seconds, and from submission to the `Finished` state in the interval $[55, 205]$ seconds.

The elapsed time for the simultaneous submission of 25 identical jobs was 59 and 535 seconds for NGM and CGM respectively, measured from submission of the first job until the last job reached the `Finished` state. The elapsed time is not, as one might expect, simply 25 times that of a single job submission, in fact it seems that CGM is scaling better than previously, as it is only ~ 9 times slower on this test. This could be explained by the polling employed by CGM which, under pressure, is able to pick up multiple jobs for each poll, as opposed to the test involving only a single job submission. The results are influenced by the 25 concurrent GridFTP connections which induce load onto the system and the performance of the LRMS also influence the test results. However, these influences affect the tests of both grid managers.

A more exhaustive performance test would be interesting to conduct, however this has not been possible within the time-frame of this project. The conducted tests, however simple, clearly indicates that NGM is performing many times $(9 - 130)$ faster than CGM when comparing the minimal overhead. This shows that NGM has eliminated some of the performance nuisances found in CGM, and furthermore that it accommodates the efficiency requirement.

6.2 Extensibility

The extensibility of the system is measured by the ability to add new features and functionality to the grid manager. This is motivated by the rigidness of CGM that does not provide any apparent ways of achieving many of the extended features mentioned in Section 1.3 on page 7. In Chapter 5 on page 45, feasible approaches to achieving these features in NGM were presented. Thus NGM is considered extensible.

6.3 Customisability

A customisable grid manager is one that provides the power of easily tweaking the behaviour of the grid manager, for instance for a grid site administrator to modify the way jobs are monitored in LRMS or the way files are made available to the computing nodes. The meaning of tweaking is thus making small adjustments to the functionality of the grid manager.

The power to customise the NGM stems from two aspects, namely the chosen architecture and the chosen programming language. The architecture provides many ways of customising the behaviour of the grid manager. Changing much of the behaviour of the grid manager is possible without actually touching any code, by simply modifying the configuration and manifest files of the appropriate handler. This provides the power to change state names and modify the ordering of handlers. However, the power is further increased by the simplicity of modifying and creating new handlers. This is provided by the Python programming language, and the handler and monitor concepts of the architecture. For the source code of the handlers and monitors in the new grid manager please refer to Appendix A.

It should be clear that tweaking the behaviour of the grid manager can be quite simple, due to the loosely coupled components, the readability and possibilities added by the Python programming language and extensive library.

6.4 Fault-tolerance

The fault-tolerance of the new grid manager relates to the way the system handles crashes. Crashes can occur at any point in time and the system must be able to recover from such a crash to a correct meaningful state, without losing job information, in order to exhibit fault-tolerant behaviour.

Informal tests have been conducted to test the fault-tolerance of the new grid manager. They show that once a job object has been successfully created in the system it is mirrored onto a string representation persistent on disk. The job pool logic is able to reload all persistent jobs into the system. However, if the system crashes while performing the initial writing of data to the job file, the job will be lost. This can be remedied by also persisting the received and unhandled events, as they are lost when the system crashes in the current implementation.

This also concerns the monitors of the architecture. Two cases exist for monitors to guarantee that external events are not lost, monitors that are able to acknowledge the reception of an external event and those that do not have this possibility.

- If monitors have the possibility of acknowledging received events, this should be done but not until the event has been persisted. In case a client to the monitor does not receive this acknowledgement, a protocol must be established between monitor and client in order to guarantee the reception of the external event.
- Monitors without this possibility must be able to guarantee that they can rediscover unpersisted events, if the system is to guarantee that external events are not lost.

Similarly, a guarantee of fault-tolerance also depends on the handlers being idempotent or *rerunnable*, i.e. the result of one execution of the handler on a given job should be similar to any number of executions. This property is necessary in the case where the system crashes while executing a particular handler, the operation of the handler is not completed and the event is unhandled. Thus, when the system is restarted the driver will relaunch the handlers registered for the unhandled event, and the handlers must be able to handle this gracefully.

This shows that fault-tolerance is not guaranteed in the current architecture implementation, and furthermore that a guarantee also depends on *contracts* between the architecture and the implemented monitors and handlers. This is clearly a subject of further work to achieve a mature and fault-tolerant grid manager and architecture.

6.5 The Outlook for NorduGrid

In this section the future outlook for NorduGrid with the addition of the new grid manager is discussed, under the assumption that the advanced use cases listed in Chapter 5 on page 45 are implemented successfully. The basic principles of a generic grid middleware design from Section 1.2 on page 5, will provide the basis for a comparison between the current NorduGrid middleware and the possible future middleware.

The grid middleware should:

- *not interfere with the existing site administration or autonomy:*
  This relates to the flexibility to adapt the grid manager for special environments and setups, where it was not originally designed to operate. The current grid manager is rather inflexible and requires a rather wholesome array of software requirements that can be difficult for site administrators to accept. Among these are:
  - The ~ 150MB sized binary installation can prove a huge mouthful for site administrators, e.g. reviewing the untrusted code seems an impossible task. The bulk installation of the Globus toolkit is necessary to use the CGM. Wishes to replace the Globus toolkit with other, standard and proved technologies have been ushered. The new grid manager architecture provides a possible transition path toward fulfilling this request by the extensibility and flexibility inherent in the architecture. And thus cutting down on the size of the binary footprint.
  - The current NorduGrid requires several communication channels to function correctly, among these are the MDS system and GridFTP server. Certain site policies may prohibit the opening of a firewall to allow for these channels, effectively restraining these sites from participating in NorduGrid. It would be desirable to develop alternative protocols to allow for the participation of these sites. The new grid manager makes this possible by allowing a transition from the current protocols, toward new alternatives.

- *not compromise existing security of users or remote sites:*
  The current NorduGrid security system does adhere to this principle.

- *provide a reliable and fault tolerant infrastructure with no single point of failure:*
  The current grid middleware employs services that can be considered single points of failure. Among these are the index servers of the information service, where the crash of a single machine could mean that an entire nation disappears from the grid. This does clearly not adhere to this principle. However this does not seem to be an urgent problem as the information service does allow for the use of fail-over servers. This is however not currently done.
  The reliability of the grid service is greatly increased by the introduction of live upgrades and the ability to distribute the NGM compared to the functionality of CGM.

- *provide support for heterogeneous components:*
  Currently the supported resource type in NorduGrid are computing nodes in clusters. For this to change, and allow for more specialised resources, e.g. telescopes and super computers, will require a restructuring of the information system and job description language. Again, a gradual transition path is provided by the new grid manager, by not dictating neither the type of information system or job description language. The NGM can ultimately lead to the support for truly heterogeneous resources.

- *use standards, and existing technologies, and facilitate interoperability with legacy applications:*
  The NGM employs the use of XML throughout the configuration and XML-RPC is the current RPC mechanism of choice. Both are good examples of standard technologies. The use of GridFTP as a job submission interface is clearly a poor choice, and work should be done to replace this with a more natural job submission interface, e.g. some form of RPC. The use of GridFTP for data transmission protocol can furthermore be criticised. [25] shows that and HTTP(S) protocol served by Apache2, with a curl based client, performs more steadily compared to a GridFTP. Replacing the data transportation protocol can be accomplished as a gradual transition in NGM.
  Replacing the current job description language of NorduGrid, XRSL, with an XML based language, e.g. JSDL, would further aid in adhering to this principle. The ability of NGM to provide a transitional path can facilitate the adaptation of a new job description language.

6.6 Summary

The evaluation concludes that the new grid manager does live up to three of the four listed demands. However the handling of fault-tolerance does not guarantee that jobs or events are not lost in the case of a system failure. The proposed solution provides the basis of future work to provide this guarantee.

The outlook for NorduGrid shows many possible improvements to NorduGrid by way of the new grid manager and architecture. The ability of functioning as a transitional path, supporting both old and new technologies is a major force of the new grid manager, that can greatly aid the replacement of tired technologies in the current middleware.

7 Conclusion

This chapter concludes the thesis. First a project summary is presented, followed by suggestions for future work relating to the new grid manager.

7.1 Project Summary

This thesis documents the efforts of designing, implementing and evaluating a new grid manager for the NorduGrid ARC.

In Chapter 1 on page 3 the concept of computational grids is introduced along with a generic model for grid middleware. The grid middleware of NorduGrid is briefly outlined and the general requirements for a new grid manager are specified.

This leads to a more thorough examination of the NorduGrid ARC middleware in Chapter 2 on page 11. The history, purpose and current state of NorduGrid is presented. The job concept is introduced followed by the task flow involved with a job submission to the NorduGrid grid manager. Following this, several identified problems with the current NorduGrid middleware are presented and compared to the requirements from Chapter 1 on page 3.

This motivates the design of a new architecture for grid managers, which is presented in Chapter 3 on page 27. The chapter initially presents the general modus operandi of grid managers, which leads to the introduction of the central concepts of the new grid manager architecture. The architecture is a plug-in-based event-driven architecture, which aims to provide an extensible and customisable foundation for creating grid managers. The design and implementation of the architecture is presented to conclude the chapter.

Chapter 4 on page 37 presents the design of a prototype drop-in replacement for the current NorduGrid ARC grid manager. The new grid manager is based on the architecture previously introduced. First the chosen job state transition system is presented, followed by a description of the components reused from the current NorduGrid middleware. The designed handlers and monitors are then presented, before finally providing an evaluation of the designed grid manager.

Obtaining an easily extensible grid manager is one of the primary reasons for developing the new grid manager. Chapter 5 on page 45 presents several advanced use cases for the NorduGrid middleware along with ways to implement them in the new grid manager.

An evaluation of the new grid manager is presented in Chapter 6 on page 59. The grid manager and architecture are related to the demands set forth in Chapter 1 on page 3, namely: efficiency, extensibility, customisability and fault-tolerance. While evaluating the efficiency of the grid manager it is compared to the current NorduGrid grid manager. The new grid manager proves to be between 9 and 130 times faster, when measuring the minimum overhead from job submission to termination of processing in the grid manager. It is found to be both extensible and customisable. However, evaluating the fault-tolerance of the new grid manager does reveal some unresolved issues regarding this requirement. The new grid manager does not guarantee the persistence of events and jobs in some cases of system failure.

To conclude this thesis we point to issues that requires future work if the new architecture and grid manager is to be successful. The subjects for future are based on the current functionality of the implementation.

7.2 Future Work

This section suggests subjects for future work based on the current state of the implementation of the new grid manager. The implementation is currently what can be considered a functioning prototype for a drop-in replacement for the current NorduGrid grid manager. The prototype furthermore constitutes a proof-of-concept of the architecture.

The current implementation is able to:

- Submit jobs via the NorduGrid client utility `ngsub`. Ideally over time this should be transitioned to another protocol than GridFTP. Ideal alternatives would be some form of RPC, for instance XML-RPC over HTTPS, which is directly supported by the monitor concept. This would remove the Globus dependency from the client utilities and provide a much more natural interface for negotiating job submissions.
- Have the appropriate data downloaded and uploaded via GridFTP, HTTP(S) and FTP. Currently there is no support for replica catalogues and caching of files. However a non-prototype implementation should ideally employ an implementation of the IO subsystem described in Section 5.2 on page 47.
- Have the job submitted to LRMS. Currently only PBS and variants are supported, i.e. those providing a `qsub` command with similar functionality of PBS. This should be extended to contain at least the LRMS flavours currently supported by NorduGrid and support monitoring of jobs queued at the LRMS.
- Be notified of job termination at LRMS. The implementation assumes a direct network connection from the system running the grid manager to the computing nodes of the cluster, in order for the wrapper script to perform the notification of job completion. Alternatives to this approach should be provided to support sites where this is not possible.

Also a method for discovering job termination at the computing nodes when the grid manager is not running, e.g. if it has crashed, is needed.

- Send notification mails when jobs reach certain states. Notification could possibly be extended to other protocols than email, for instance instant messaging, e.g. the Jabber protocol.
- Clean session directory after a specific amount of time.

The implementation also lacks some features:

- Work is needed to ensure that the grid manager is indeed fault-tolerant, as mentioned in Section 6.4 on page 61.
- Security in the grid manager is an almost untouched area. Providing ways to change the user id of submitted jobs, i.e. adhering to the gridmap file of NorduGrid is not supported. A sane security advice is to let the grid manager run with the least amount of privileges needed, and escalating the privileges temporarily when needed. Sand-boxing of handlers, i.e. letting handlers run with limited privileges, should also be examined in this context.
- A well-defined means of communication between a handler and the monitor that created the event currently processed by the handler. This could be useful in cases where the monitor needs to reply to a client, but delegates the work of creating such a reply to a handler. The functionality could be implemented by creating a protected communication object that is placed on the event when it is notified to the driver. This object would be shared by the monitor and the handler processing the event, and could thus be used as a means of communication.
- Work should be done to implement some of the advanced features discussed in Chapter 5 on page 45. Particularly the IO subsystem and the possibility of changing and upgrading a live grid manager are essential features to provide a highly available and reliable grid service.
  Providing support for advance reservations will almost certainly result in more specialised use-cases for resource sharing on the grid. It enables support for co-allocation, i.e. allocating multiple resources at different sites simultaneously. Interactive use of resources is also made possible, something that is very bothersome without advance reservations.
- Some way of configuring the handlers. Most handlers have some parameters that can be modified, for instance the email address where notification emails should originate from and which SMTP server should be used to send them. There needs to be a way to specify these parameters in the manifest file of a handler.

Working with the grid manager and architecture has also given inspiration to more exotic suggestions for improvements:

- Providing ways of interactively managing and monitoring the grid manager through a user interface, i.e. not simply using log files and signals. Such an interface could allow for the administrator to enable and disable handlers or monitors at runtime by the click of a mouse. This could possibly be achieved through a specialised monitor that would interface with the driver and naturally a program acting as user interface.
- Providing better support for testing new functionality of handlers and monitors, i.e. ways to monitor and debug plug-ins at runtime.

A

Source Code For Grid Manager Implementation

A.1 Job Submission Script

Listing A.1: Source code for the job submit script called from the GridFTP job plugin.

```
#!/usr/bin/env python

import sys, os
from xmlrpclib import ServerProxy
import socket

import logging
logger = logging.getLogger('job_plugin_submit')
hdlr = logging.FileHandler('job_plugin_submit.log') # FIXME: from config
formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
hdlr.setFormatter(formatter)
logger.addHandler(hdlr)
logger.setLevel(logging.WARNING)

if len(sys.argv) != 4:
  print '''usage:
    %s controldir sessionroot jobid

    where:
      controldir is the path to controldir of job
      sessionroot is the path to the root of the session dirs
      jobid is the id of the job to be submitted''' % sys.argv[0]
  sys.exit(os.EX_USAGE)
```

```
controldir = sys.argv[1]
sessionroot = sys.argv[2]
try:
  jobid = str(int(sys.argv[3]))
  # looks stupid, but nordugrid jobids are longer ints than supported by xmlrpc
except ValueError:
  logger.error('Integer conversion of jobid failed, aborting')
  sys.exit(os.EX_DATAERR)

server = ServerProxy("http://localhost:9000") # local server
logger.info('submitting controldir: %s sessionroot: %s jobid: %s'
    % (controldir, sessionroot, jobid))

try:
  res = server.submit_job_from_gridftp(controldir, sessionroot, jobid)
except socket.error, errval:
  errno, errstr = errval
  logger.error('Error code %d: %s ... aborting' % (errno, errstr))
  sys.exit(os.EX_IOERR)

if not res:
  sys.exit(os.EX_IOERR)
```

A.2 Monitor

Listing A.2: Source code for the job submission and job termination monitor.

```
from monitor import Monitor
from event import Event

import SimpleXMLRPCServer, SocketServer, threading

class ThreadingXMLRPCServer(SocketServer.ThreadingMixIn,
    SimpleXMLRPCServer.SimpleXMLRPCServer):
  pass

class ng_monitor(Monitor):
  def run(self):
    server = ThreadingXMLRPCServer(("localhost", 9000))
    self.server = server
    # register functions
    server.register_function(self.submit_job_from_gridftp)
    server.register_function(self.wrapper_finished)
    self.logger.info("ng_monitor is a threading xmlrpc server, serving forever ↩
        ...")
    server.serve_forever()

  def shutdown(self):
    self.server.socket.close()

  def submit_job_from_gridftp(self, controldir, sessionroot, jobid):
    jd_path = controldir+'/job.'+jobid+'.description'
    self.logger.info("ng_monitor opening jd: %s" % jd_path)
    self.logger.info("ng_monitor sessionroot: %s" % sessionroot)
    description = open(jd_path, 'r')
    rsl = ''
    for line in description.readlines():
      rsl = rsl + line

    event = Event('RECEIVED', 5, jobid)
    event.dict['rsl'] = rsl
    event.dict['controldir'] = controldir
    event.dict['sessionroot'] = sessionroot
    event.dict['sessiondir'] = sessionroot+'/'+jobid

    self.notify(event)

    return True

  def wrapper_finished(self, jobid):
    self.logger.info("got wrapper finished for jobid: %s" % jobid)
    event = Event('FINISHED', 5, jobid)
    self.notify(event)
    return True
```

A.3 Job Description Parser

Listing A.3: Manifest for the job description parsing handler.

```
<handler>
  <events>
    <event rank="1">RECEIVED</event>
  </events>

  <statemap>
    <state internal="ACCEPTED">SUCCESS</state>
    <state internal="FINISHED">FAILURE</state>
  </statemap>
</handler>
```

Listing A.4: Source code for the job description parsing handler.

```
from handler import Handler
from job import Job
from ext.pyrsl import rsl

class parse_jd(Handler):
  """ responsible for parsing job description in event and attaching it to
      the appropriate job.
  """
  def handle_job(self, event, job, jobpool):
    # event: create the new job
    assert job == None
    job = Job(event.jobid, 'INITIAL')

    rsl_jd = event.dict['rsl']
    ##print 'ORIGINAL:',rsl_jd
    spec = rsl.build_spec(rsl_jd, debug = False)
    rsl_map = self.spec_to_map(spec)

    # for each key, add the key->value pair to JD of the job
    for key in rsl_map.keys():
      if rsl_map.has_key(key) and (rsl_map[key] != ""):
        job.set_jd_attr(key, rsl_map[key])

    # fix the clientxrsl if it is there -- FIXME: the parser can't do this yet
    if rsl_map.has_key('clientxrsl'):
      clientxrsl = ""
      for i in rsl_map['clientxrsl']:
        clientxrsl += i

      #remove leading and trailing " marks
      clientxrsl = clientxrsl[1:-1]

      # create a map of clientxrsl and replace the broke entry in job.JD
      self.logger.info('Fixing clientxrsl: %s' % clientxrsl)
      spec = rsl.build_spec(clientxrsl, debug = False)
      clientmap = self.spec_to_map(spec)
      job.set_jd_attr('clientxrsl', clientmap)

    # set remaining needed attributes in job
    event_to_jd = ['controldir', 'sessionroot', 'sessiondir']
    for ev in event_to_jd:
      job.set_jd_attr(ev, event.dict[ev])

    self.logger.info('parsed job description for job %s' % job.jobid)

    # the job has now been created, add it to the pool of jobs
    jobpool.insert_job(job)
    job.state = self.statemap['SUCCESS']

  def spec_to_map(self, spec):
    """ Converts a pyrsl specification to a pythonised version in a map. """
    result = {}
    rsl_tuples = rsl.pythonize(spec)

    # pr is a tuple of (key, relation) e.g. (jobname, (jobname=/bin/ls))
    # where relation.name = jobname AND relation.value_sequence = /bin/ls
    for tuple in rsl_tuples:
      for key, relation in tuple.items():
        assert key == relation.name
        result[relation.name] = relation.value_sequence
        # considering the parser being alpha, we should perhaps handle errors?

    return result
  def map_print(self, msg, map):
    """ Prints a map. Useful for debugging, and not much else. """
    print msg
    for key in map.keys():
      print '\t',key, '--->\t', map[key]
```

A.4 Notify User

Listing A.5: Manifest for the notify user handler.

```
<handler>
  <events>
    <event rank="1">ACCEPTED</event>
    <event rank="1">QUEUED</event>
    <event rank="1">FINISHED</event>
    <event rank="1">STAGEDOUT</event>
    <event rank="1">CLEANED</event>
  </events>
</handler>
```

Listing A.6: Source code for the notify user handler.

```
from handler import Handler
from job import Job

import smtplib
from email.MIMEText import MIMEText
from datetime import datetime
import time

SMTPSERVER = 'smtp.cs.aau.dk' # FIXME: retrieve from configuration
SENDER = 'grid@tender.grid.aau.dk' # FIXME: retrieve from configuration

class notify_user(Handler):
  """ Notify users of events on job by sending emails
  """
  def handle_job(self, event, job, jobpool):

    notify = job.get_jd_attr('notify')
    if notify == None:
      # No notify found in job description
      return

    # decide which flag this event represents
    # b begin (PREPARING)
    # q queued (INLRMS)
    # f finalizing (FINISHING)
    # e end (FINISHED)
    # c cancellation (CANCELLED)
    # d deleted (DELETED)
    eventtype_to_flag = {
      'ACCEPTED' : 'b',
      'QUEUED' : 'q',
      'FINISHED' : 'f',
      'STAGEDOUT': 'e',
      'CLEANED' : 'd' }

    if not eventtype_to_flag.has_key(event.type):
      # not an event we notify for
      return

    sent = False

    for notify_string in notify:
      notify_split = notify_string.split(' ')

      if len(notify_split) != 2:
    # does not match the format, go to next
    continue

      flag = notify_split[0]
      recipient = notify_split[1]

      current_event_flag = eventtype_to_flag[event.type]

      # assume first element is notification flags
      if not current_event_flag in flag:
    # user did not specify notification for this event
    continue

      # construct message and create connection to smtp
      msg = MIMEText('job %s has now received event %s.' % (job.jobid, event. ↩
          type))
      msg['From'] = SENDER
      msg['Subject'] = '[GRID] Job %s event recieved' % job.jobid
      msg['Date'] = datetime.now().ctime()

      session = smtplib.SMTP(SMTPSERVER, 25)

      # assume remaining elements are email addresses to receive notification
      msg['To'] = recipient
      smtpresult = session.sendmail(SENDER, recipient, msg.as_string())
      sent = True

      session.quit()

    if sent: self.logger.info('%s has completed. Email supposedly sent.' % job. ↩
        jobid)
```

A.5 Data Staging

Listing A.7: Manifest for the data staging handler.

```
<handler>
  <events>
    <event rank="100">ACCEPTED</event>
    <event rank="100">FINISHED</event>
  </events>

  <statemap>
    <state internal="STAGEDIN">STAGEDIN</state>
    <state internal="STAGEDOUT">STAGEDOUT</state>
  </statemap>
</handler>
```

Listing A.8: Source code for the data staging handler.

```
from handler import Handler
from job import Job
from urlparse import urlparse

FILECHECK_INTERVAL = 1 # seconds to sleep before re-checking for stagein files
TEMP_EXTENSION = '.tmp'

class dummy_stager(Handler):

  def handle_job(self, event, job, jobpool):

    state = job.state

    # some useful variables
    sessiondir = job.get_jd_attr('sessiondir')
    controldir = job.get_jd_attr('controldir')
    sessionurl = 'file://' + sessiondir
    userproxy = controldir + '/job.' + str(job.jobid) + '.proxy'

    if event.type == 'ACCEPTED':
      # just check that all executables are in place (slow slow gridftpd)
      if job.get_jd_attr('executables') != None:
        executables = []

        for filename in job.get_jd_attr('executables'):
          if not filename[0] == '/':
            executables.append(sessiondir + '/' + filename)

        self.wait_for_files(executables)

        # make sure the executables are executable
        for filename in job.get_jd_attr('executables'):
          if not filename[0] == '/':
            self.logger.info('changing permissions for %s' % filename)
            import os
            os.chmod(sessiondir + '/' + filename, 0500)

      #### DOWNLOAD ####
    ## now the executables are in place, let’s download inputfiles
    if job.get_jd_attr(’inputfiles’) != None:

      self.logger.info(’Found some inputfiles to get, please wait a bit.’)

      ## files we need to be in place before we change state
      watchlist = []

      for urls in job.get_jd_attr(’inputfiles’):
    # download files sequentially
        # remove any unnecessary " marks from the URL’s
        src_url = urls[1].strip(’"’)
        dest_url = sessionurl + ’/’ + urls[0].strip(’"’)

        # use ngcopy
        self.ngcopy(src_url, dest_url, userproxy)

        # FIXME: should we only add this if protocol supported?
        watchlist.append(sessiondir + ’/’ + urls[0].strip(’"’))

      # block until all files are in place
      self.wait_for_files(watchlist)
      self.logger.info(’Done downloading, lets move on to STAGEDIN’)

    new_state = self.statemap[’STAGEDIN’]
    self.update_status_file(controldir, job.jobid, new_state)
    job.state = new_state
    return

  #### UPLOAD ####
  if event.type == ’FINISHED’:

    ## the job has finished, now we should upload the outputfiles
    if job.get_jd_attr(’outputfiles’) != None:

      self.logger.info(’Found some outputfiles to upload, please wait a bit.’)

      # upload each file sequentially
      for urls in job.get_jd_attr(’outputfiles’):

        # remove any unnecessary " marks from the URL’s
        src_url = sessionurl + ’/’ + urls[0].strip(’"’)
        dest_url = urls[1].strip(’"’)

        # use ngcopy
        self.ngcopy(src_url, dest_url, userproxy)

      # FIXME: we should wait here until all uploads have completed
      self.logger.info(’Started all uploaders, lets move on to STAGEDOUT’)

    new_state = self.statemap[’STAGEDOUT’]
    self.update_status_file(controldir, job.jobid, new_state)
    job.state = new_state
```

```
      return

  def ngcopy(self, src_url, dest_url, userproxy):
    """
    Executes:
     sh -c "export X509_USER_CERT=#userproxy#; ngcopy #src_url# #dest_url#"

     - src_url and dest_url must be fully qualified URLs
     - userproxy must be a complete path to a proxy certificate file

     Examples of tested fully qualified URLs:
      - gsiftp://tender.grid.aau.dk:2811/storage/stdout.txt
      - file:///home/grid/data/samples.data

     If both source and dest ends with '/' ngcopy copies the source folder
     recursivly.

     See man ngcopy for more info
    """
    # check if we support the specified protocols
    supported_protocols = ['gsiftp','file']
    src_protocol = urlparse(src_url)[0]
    dest_protocol = urlparse(dest_url)[0]

    if src_url == "": src_url = 'EMPTY'
    if dest_url == "": dest_url = 'EMPTY'
    if src_protocol == "": src_protocol = 'EMPTY'
    if dest_protocol == "": dest_protocol = 'EMPTY'

    if (not self.supported(src_protocol) or not self.supported(dest_protocol)):
      return

    # use ngcopy to transfer file
    args = { "proxy":userproxy, "src":src_url, "dest":dest_url }
    cmd = ('sh -c "export X509_USER_CERT=%(proxy)s; ngcopy -d1 %(src)s %(dest)s &>ngcopy.log"' % (args))

    import os
    os.popen2(cmd)

    self.logger.info("Started transferring %s to %s" % (src_url, dest_url))

  def supported(self, test_prot):
    """ returns True if protocol is currently supported, False otherwise. """

    # return False if no protocol is specified
    if test_prot == "":
      return False

    # return True if we find a match
    supported_protocols = ['gsiftp','file']
    for supported in supported_protocols:
      if test_prot == supported: return True
    # no match --> return false
    return False

  def update_status_file(self, controldir, jobid, new_state):
    """ write the status of the job to the statusfile """

    status_fn = controldir + '/' + 'job.' + jobid + '.status'

    file = open(status_fn + TEMP_EXTENSION, "w")
    file.write(new_state + '\n')

    import os
    os.rename(status_fn + TEMP_EXTENSION, status_fn)

  def wait_for_files(self, filelist):
    """
    blocks until all files in filelist exists.
     - filelist must be either xrsl.inputfiles or xrsl.outputfiles
    """
    import os.path

    if len(filelist) < 1: return

    while True:

      missing = False

      for filename in filelist:
        if not os.path.exists(filename) and not os.path.isdir(filename):
          missing = True
          self.logger.info('%s is missing, sleeping and rechecking later' % ↩
              filename)

      if not missing:
        self.logger.info('All files are in place, stop blocking')
        return
      else:
        # missing files, sleep and check again
        from time import sleep
        sleep(FILECHECK_INTERVAL)
```

A.6 Wraper Creation

Listing A.9: Manifest for the handler that produces the wrapper script to be submitted to LRMS.

```
<handler>
 <events>
  <event rank="1">STAGEDIN</event>
 </events>
```

```
  <statemap>
    <state internal="FINISHED">FAILURE</state>
  </statemap>
</handler>
```

Listing A.10: Source code for the handler that produces the wrapper script to be submitted to LRMS.

```
from handler import Handler
from job import Job

class create_wrapper(Handler):
  """ Responsible for generating a job wrapper. The wrapper is what will
  actually be running inside the LRMS. Hence, the wrapper should setup some
  environment, and basically just start the job.

   NOTE: ldap backends could use this for getting job info
   NOTE: we can send event from the wrapper
  """
  def handle_job(self, event, job, jobpool):
    ## event: job transitioned to StagedIn

    w = Wrapper(job)

    fn = job.get_jd_attr('sessiondir')+'/wrapper.py'
    wf = file(fn, 'w')
    wf.write(w.script)
    wf.close()

    from os import chmod
    chmod(fn, 0700)

    self.logger.info('wrote wrapper to fn')

class Wrapper:
  """
  Just some stupid wrapper. Currently, only cmd and args is configurable.
  Needs to be tweaked if it is supposed to be able to run on Condor, PBS,
  fork, Torque, and other LRMS types.
  """

  def __init__(self, job):
    self.job = job
    self.script = ""
    self.script += self.python_env()
    self.script += self.setup()
    self.script += self.pbs_config()
    self.script += self.jobinfo()
    self.script += self.setupjob()
    self.script += self.running()
    self.script += self.epilogue()

  def python_env(self):
    return '#!/usr/bin/env python'
  def setup(self):
    return '''
import socket, datetime, os, logging

jobid = "'''+self.job.jobid+'''"
os.chdir("'''+self.job.get_jd_attr('sessiondir')+'''")

logger = logging.getLogger('wrapper_'+jobid)
hdlr = logging.FileHandler('wrapper.log')
formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
hdlr.setFormatter(formatter)
logger.addHandler(hdlr)
logger.setLevel(logging.INFO)

'''

  def pbs_config(self):
    return '''
### Job name
#PBS -N wrapper

### Declare job non-rerunable
#PBS -r n

### Output files
#PBS -e wrapper.err
#PBS -o wrapper.out

### Queue name (default, batch, small, medium, long, verylong)
#PBS -q default

### Number of nodes
#PBS -l nodes=1
'''

  def jobinfo(self):
    return '''
# print simple info
logger.info("Running on host %s" % socket.gethostname())
'''

  def setupjob(self):
    return '''
# Setup job command and arguments
path = "%s"
cmd = "%s"
args = %s
''' % (self.job.get_jd_attr('sessiondir'),
    self.job.get_jd_attr('arguments')[0],
    self.job.get_jd_attr('arguments')[1:])

  def running(self):
    res = '''
```

```
# Run the executable as:
#  sh -c 'cmd arg1 ... argN > stdout 2> stderr < stdin'
sharg = ''
for e in [cmd]+args:
  sharg = sharg + e + ' '

'''
    stdin = self.job.get_jd_attr('stdin')
    stdout = self.job.get_jd_attr('stdout')
    stderr = self.job.get_jd_attr('stderr')

    if stdin != None:
      res = res +'''
sharg = sharg+'< %s '
''' % stdin[0]
    if stdout != None:
      res = res+'''
sharg = sharg+'> %s '
''' % stdout[0]
    if stderr != None:
      res = res+'''
sharg = sharg+'2> %s '
''' % stderr[0]

    res = res+'''
from time import time
start_time = time()
result = os.spawnvp(os.P_WAIT, 'sh', ['sh','-c', sharg])
end_time = time()
'''
    return res

  def epilogue(self):
    return '''
logger.info('Runtime: %f' % (end_time - start_time))
logger.info('Return code: %d' % result)

from xmlrpclib import ServerProxy
import socket

server = ServerProxy("http://localhost:9000")
try:
  res = server.wrapper_finished(jobid)
except socket.error, errval:
  errno, errstr = errval
  logger.error('Could not signal wrapper finished. Error %d: %s' % (errno, ↩
      errstr))

'''
```

A.7 LRMS Submission

Listing A.11: Manifest for the handler that submits jobs to LRMS.

```xml
<handler>
  <events>
    <event rank="10">STAGEDIN</event>
  </events>

  <statemap>
    <state internal="QUEUED">SUCCESS</state>
    <state internal="FINISHED">FAILURE</state>
  </statemap>
</handler>
```

Listing A.12: Source code for the handler that submits jobs to LRMS.

```python
from handler import Handler
from job import Job

LRMS_TYPE = 'pbs'
TEMP_EXTENSION = '.tmp'

class lrms_submit(Handler):
  """ Responsible for generating a job wrapper. The wrapper is what will
  actually be running inside the LRMS. Hence, the wrapper should setup some
  environment, and basically just start the job.
  """
  def handle_job(self, event, job, jobpool):
    # event: wrapper ready for job

    import os
    sessiondir = job.get_jd_attr('sessiondir')
    controldir = job.get_jd_attr('controldir')
    wrapper_fn = sessiondir + '/wrapper.py'

    if LRMS_TYPE == 'fork':
      os.spawnvp(os.P_NOWAIT, wrapper_fn, ['wrapper.py'])
    elif LRMS_TYPE == 'pbs':
      sh_cmd = '(cd %s; qsub %s)' % (sessiondir, wrapper_fn)
      os.spawnvp(os.P_WAIT, 'sh', ['sh', '-c', sh_cmd] )

    # now the job has been submitted to lrms
    self.logger.info('submitted job %s to lrms (type %s)' % (job.jobid, ↩
        LRMS_TYPE))

    # FIXME: what if lrms or fork submit fails?

    new_state = self.statemap['SUCCESS']
    self.update_status_file(controldir, job.jobid, new_state)
    job.state = new_state

  def update_status_file(self, controldir, jobid, new_state):
```

```
  """ write the status of the job to the statusfile """

  status_fn = controldir + '/' + 'job.' + jobid + '.status'

  file = open(status_fn + TEMP_EXTENSION, "w")
  file.write(new_state + '\n')

  import os
  os.rename(status_fn + TEMP_EXTENSION, status_fn)
```

A.8 Job Expiry

Listing A.13: Source code for the monitor that sends tick events, used for cleaning jobs at regular intervals.

```python
from monitor import Monitor
from event import Event

EVENT_DELTA = 24 * 60 * 60 # seconds between each TICK

class timer(Monitor):
 def run(self):
  from time import sleep

  self.logger.info('sending ticks each %d seconds' % EVENT_DELTA)

  while True:
   event = Event('TICK', 5, 0)
   self.notify(event)

   sleep(EVENT_DELTA)

 def shutdown(self):
  pass
```

Listing A.14: Manifest for for the handler that handles job cleaning.

```xml
<handler>
 <events>
  <event rank="100">STAGEDOUT</event>
  <event rank="1">TICK</event>
 </events>
 <statemap>
  <state internal="CLEANED">SUCCESS</state>
  <state internal="TICK">TIME_EVENT</state>
  <state internal="STAGEDOUT">STAGEDOUT_EVENT</state>
 </statemap>
</handler>
```

Listing A.15: Source code for the handler that handles job cleaning.

```python
from handler import Handler
from job import Job

REGISTRY_FILENAME = 'job_expiry_registry'
DEFAULT_LIFETIME = 7 * 24 * 60 * 60 # default lifetime in seconds
MAX_LIFETIME = 2 * DEFAULT_LIFETIME

class job_expiry(Handler):
  """ Registers jobs for deletion.
  Wakes up with TICK events and checks if any limits have exceeded, meaning a
  job has to be deleted.
  """

  def handle_job(self, event, job, jobpool):

    if event.type == self.statemap['TIME_EVENT']:
      self.tick(jobpool)
      return

    if event.type == self.statemap['STAGEDOUT_EVENT']:
      self.register_job(job)
      return

  def tick(self, jobpool):
    """ handle the tick event """
    from time import time

    current_time = time()

    registry = self.get_registry()
    for entry in registry:
      (expiry_time, jobid) = entry
      if expiry_time <= current_time:
    registry.remove(entry)

    job = jobpool.get_job(jobid)
    if job == None:
      self.logger.info('attempted to delete an unknown job %s, ignoring' % jobid ↩
          )
      continue

    self.clean_job(job)

    self.logger.info('cleaned jobid: %s' % jobid)

    self.put_registry(registry)

  def register_job(self, job):
    """ register the job for deletion """
    # lifetime is given in seconds on the gmside xrsl
    lifetime = job.get_jd_attr('lifeTime')

    if lifetime == None:
      lifetime = DEFAULT_LIFETIME
    else:
      lifetime = int(lifetime)

    if lifetime > MAX_LIFETIME:
      lifetime = MAX_LIFETIME

    import time
    current_time = time.time()

    expiry_time = current_time + lifetime

    registry = self.get_registry()
    registry.append((expiry_time, job.jobid))
    self.put_registry(registry)

    self.logger.info('registered job %s for deletion not before %s' % (job.jobid ↩
        , time.asctime(time.localtime(expiry_time))))

  def get_registry(self):
    """ return the registry from file """

    # provide locking, assuring only one handler working with registry
    from threading import Lock
    if not self.__dict__.has_key('reg_lock'):
      print "no lock"
      self.reg_lock = Lock()

    self.reg_lock.acquire()
    import pickle

    # load the registry from pickled state
    register = None
    try:
      register = pickle.load(file(REGISTRY_FILENAME))
    except IOError:
      self.logger.info("No job expiry registry found in %s, creating a fresh" % ↩
          REGISTRY_FILENAME)
      register = []

    return register

  def put_registry(self, registry):
    import pickle
    assert registry != None # attempt to put None registry

    # write file to temporary
    TEMP_EXTENSION = '.tmp'
    f = file(REGISTRY_FILENAME+TEMP_EXTENSION, 'w')
    pickle.dump(registry, f)
    f.close()

    import os
    os.rename(REGISTRY_FILENAME+TEMP_EXTENSION, REGISTRY_FILENAME)

    # release lock
    from threading import Lock
    self.reg_lock.release()

  def clean_job(self, job):
    """ cleans a job, currently removes the session directory of a job """
    sessiondir = job.get_jd_attr('sessiondir')

    import os
    from os.path import join
    for root, dirs, files in os.walk(sessiondir, topdown=False):
      for name in files:
    os.remove(join(root, name))
      for name in dirs:
    os.rmdir(join(root, name))
    # sessiondir should be ready for rmdir'ing
    os.rmdir(sessiondir)
```

Bibliography

[1] R. Buyya, "Grid computing info centre FAQ."
`http://www.gridcomputing.com/gridfaq.html` .

[2] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, ch. 2. Morgan-Kaufmann, 1998.

[3] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Lecture Notes in Computer Science*, vol. 2150, 2001.

[4] M. Baker, R. Buyya, and D. Laforenza, "Grids and grid technologies for wide-area distributed computing," *International Journal of Software: Practice and Experience (SPE)*, vol. 32, pp. 1437–1466, 2002.

[5] "The Globus Alliance."
`http://www.globus.org/` .

[6] "The LHC Grid Computing Project."
`http://lcg.web.cern.ch/LCG/` .

[7] P. Avery and I. Foster, "GriPhyN Annual Report for 2003–2004," August 2004.

[8] C. Catlett, "The TeraGrid: A Primer," September 2002.

[9] J. Tordsson, "Resource Brokering for Grid Environments," Master's thesis, Umeå University, June 2004.
`http://www.nordugrid.org/documents/tordsson-thesis.pdf` .

[10] T. Christensen and R. A. K. et. al., "Nubis: A decentralised, flexible, fault-tolerant and scalable foundation for computational grids," tech. rep., Aalborg University, Dec. 2004.

[11] P. Eerola, B. Konya, and O. S. et. al., "The NorduGrid Architecture and Tools," in *Computing in High Energy and Nuclear Physics*, 2003.
`http://www.nordugrid.org/documents/MOAT003.pdf` .

[12] "The ATLAS Experiment."
`http://atlasexperiment.org/` .

[13] P. Eerola, B. Konya, and O. S. et. al., "ATLAS Data-Challenge 1 on NorduGrid," in *Computing in High Energy and Nuclear Physics*, 2003.
`http://www.nordugrid.org/documents/MOCT011.pdf` .

[14] "XRSL (Extended Resource Specification Language)."
`http://www.nordugrid.org/documents/xrsl.pdf` .

[15] A. Konstantinov, “The NorduGrid Grid Manager And GridFTP Server: Description And Administrators Manual.”
`http://www.nordugrid.org/documents/GM.pdf` .

[16] “RFC 959: File Transfer Protocol.”
`http://www.w3.org/Protocols/rfc959/` .

[17] The University of Chicago and The University of Southern Califonia, “GridFTP: Universal Data Transfer for the Grid,” September 2000.

[18] B. Kónya, “NorduGrid server installation instructions.”
`http://www.nordugrid.org/documents/ng-server-install.html` .

[19] “NorduGrid middleware, the Advanced Resource Connector.”
`http://www.nordugrid.org/middleware/` .

[20] “GSI: Key Concepts.”
`http://www-unix.globus.org/toolkit/docs/3.2/gsi/key/index.html` .

[21] “Internet X.509 Public Key Infrastructure Certificate and CRL Profile.”
`http://www.ietf.org/rfc/rfc2459.txt` .

[22] “Handbook of Applied Cryptography.”
`http://www.cacr.math.uwaterloo.ca/hac/` .

[23] A. Konstantinov, “The HTTP(s,g) And SOAP Framework.”
`http://www.nordugrid.org/documents/HTTP_SOAP.pdf` .

[24] A. Konstantinov, “ARC::DataMove Reference Manual.”
`http://www.nordugrid.org/documents/datamove.pdf` .

[25] A. McNab, “HTTP as a data protocol and HTTP-Downgrade.”
`http://www.gridlock.org.uk/20040726113603.html` .

[26] J. Chin and P. V. Coveney, “Towards tractable toolkits for the grid: a plea for lightweight, usable middleware,” tech. rep., University of London, Feb. 2004.

[27] D. L. Parnas, “Designing Software for Ease of Extension and Contraction,” in *Proceedings of the International Conference on Software Engineering*, pp. 264–277, May 1978.

[28] J. Aycock, “Compiling Little Languages in Python,” in *7th International Python Conference*, 1998.
`http://pages.cpsc.ucalgary.ca/ aycock/spark/paper.pdf` .

[29] K. Jeacle and J. Crowcroft, “Extending Globus to support Multicast Transmissions,” in *Proceedings of the UK e-Science All Hands Meeting*, (Nottingham, UK), September 2004.

[30] W. Allcock, J. Bresnahan, R. Kettimuthu, and J. Link, “The Globus eXtensible Input/Output System (XIO): A Protocol Independent IO System for the Grid,” in *19th IEEE International Parallel and Distributed Processing Symposium*, (Denver, Colorado), March 2005.

[31] T. Kosar and M. Livny, “Stork: Making Data Placement a First Class Citizen in the Grid,” in *In Proceedings of the 24th IEEE Int. Conference on Distributed Computing Systems (ICDCS2004)*, (Tokyo, Japan), March 2004.

[32] T. Kosar, "Stork: Making Data Placement a First Class Citizen in the Grid," 2004. `http://www.cs.wisc.edu/condor/stork/talks/talk_cern_may04.ppt` .

[33] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, (Chicago, IL), July 1998.

[34] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality* (F. Berman, G. Fox, and T. Hey, eds.), John Wiley & Sons Inc., December 2002.

[35] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: a decentralized network coordinate system," in *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 15–26, ACM Press, 2004.

[36] "The Xen virtual machine monitor," 2005. `http://xen.sf.net` .