Henrik Thostrup Jensen
Jesper Ryge Leth

A Job Manager for the NorduGrid ARC

Dat6 Project
February 2004 - June 2004

To be evaluated June 28^{th} 2004

Department of Computer Science
Aalborg University
Fredrik Bajersvej 7E
DK–9220 Aalborg
DENMARK

Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE:

A Job Manager for the NorduGrid ARC

PROJECT PERIOD:

Dat6,
1st February 2004 -
14th June 2004

PROJECT GROUP:

B2-201/d603a

GROUP MEMBERS:

Henrik Thostrup Jensen
Jesper Ryge Leth

SUPERVISOR:

Josva Kleist

NUMBER OF COPIES: 6

REPORT PAGES: 102

APPENDIX PAGES: 18

TOTAL PAGES: 120

SYNOPSIS:

This report describes the development of a system capable of managing jobs in NorduGrid. The project is a continuation of a former project, in which a daemon capable of resubmitting jobs were developed.

The report starts by giving an introduction to grids, what they are, and how resource sharing is organized by creating virtual organizations. Hereafter a general grid model is presented, along with examples of three different grid architectures. The NorduGrid project is then described with regards to its political structure and architecture of the toolkit.

After the description of the NorduGrid project, our initial considerations for the project is described. The need for a job manager is analyzed, our design philosophy is presented and features of the Job Manager discussed.

Hereafter an overview of the Job Manager is presented, where after the small modules in the Job Manager are discussed. The next chapter describes how Job Management is done by separating bookkeeping of jobs and actions into different modules, making it possible to redefine the way jobs are dealt with. It is then described how to avoid the Job Manager becoming a single point of failure, by making other Job Managers acting as failover. Lastly a future work chapter presents further ideas.

Finally we conclude that the Job Manager delivers a framework for production systems while also providing applications with a simple interface to access the grid.

Project Summary

This report describes the development of system capable of managing jobs in computational grids running the NorduGrid Advanced Resource Connector (NorduGrid ARC) middleware.

Computational grids are a way of interconnecting high performance computational (HPC) resources, in order to be able to solve larger problems.The general idea is based on an analogy to the electric power grid, and that it should be possible to connect to the grid to use computational power. However grid technology is a rather new technology and there is a long way to go before it will mature it into such a transparent system.

Most modern grids reminds of the batch systems from the times of yore, where a job is submitted to a queue and executed, without interaction on the part of the user. Upon completion the user collects the output from the job. There are however systems that tries to implement a grid that are more similar to the concept of a supercomputer.

On of the most popular toolkits for constructing grids are the Globus toolkit and the NorduGrid ARC is build on top of Globus extending and replaces some of the Globus components. The main elements of NorduGrid ARC is the Information System, the Grid Manager and the User Interface. The Information System makes it possible to monitor the grid and query the resources for information about the resources. The Grid Manager runs on the frontend of the resources managing the interaction between the grid and the local resources and adheres to policies set locally. The User Interface is a command line interface for the standard Unix shell; it makes it possible to submit batch jobs to the resources connected to the grid and retrieve data.

When a job is submitted it is possible to check the status through a command line interface or through a web portal. However in its current form the user interface does not deliver any means of automatically responding to changes in the state of the grid or jobs without user intervention and neither does it provide an interface for applications to interact with. It is these problems this project addresses.

The decision to develop a Job Manager was based on experience from a set of experiments to evaluate if grids connected by the NorduGrid ARC was mature enough to be used for processing and analyzing data from the Large Hadron Collider, which is being build at the European research center for high energy physics (CERN). In this experiment a lot of time was spent monitoring jobs and resubmitting failed jobs manually.

The job manager developed in this project delivers a tool that monitors the grid and responds to changes in the jobs and resources. It is the continued development of a former project, also developed by us, where we made a daemon that could monitor and automatically resubmit jobs if the failed.

The goal of the project is to create a complete Job Manager, that automates tedious tasks, is extensible, and delivers a clean interface making it easier for application developers to take advantage of the underlying grid infrastructure. In addition the Job

Manager provides failover and supports caching and user defined handlers (plug-ins)

The Job manager is constructed as a layer between the application and the grid. It runs continuously, making it capable of monitoring jobs and reacting to changes in the grid. The Job Manager delivers the necessary functions for the application to use the grid through an XML-RPC protocol, which lays on top of OpenSSL. The standard NorduGrid credentials, i.e., X.509 proxy certificates, are for authentication.

To use the Job Manager, the application submits a job, in the form of an xRSL description, over XML-RPC. The Job Manager performs the necessary steps to submit the job to a cluster fitting the description.

The NorduGrid ARC is single execution oriented, and each time a job is submitted it is assigned a unique jobid, as identifier. This means that there are no way of tracking a job through several executions. For the Job Manager to work, this had to be changed and the concept of a job tag was introduced, in addition to the existing job id. The job tag does not change and is used by the Job Manager as a consistent way to keep track of the job. Furthermore a lot of meta data is collected about a job, e.g., resubmission attempts, failed executions, and successful executions.

In addition to the basic features of the Job Manager, it has support for handlers, which are plug-ins that extend or change the way the manager handles jobs. An example of this is that it is possible to change the default scheduler, with a scheduler written by the user.

Since grids are highly distributed and dynamic systems and it is not desirable to have single points of failure. In order to prevent the Job Manager from becoming such, it is possible to start several managers to act as failover managers taking over the management of the users jobs, if one should fail.

The Job Manager was developed in the programming language Python for two purposes; to make it platform independent and to speed up the development process. Most of the NorduGrid ARC is developed in either C or C++, breaking with this tradition meant that time was spent writing wrappers and generating bindings to the existing NorduGrid ARC code base.

The project has focused on the development of the Job Manager, but there are some changes to the NorduGrid ARC that are necessary for the manager to function properly in a production grid. Among these are; the Information System should be changed to integrate the Job Manager like other resources in the grid making it possible to identify and locate Job Manager running. The description of jobs in the Information System should likewise be extended to support job tags. It would also be preferable to have the Job Manager support multiple users, but due to Globus dependencies, this would require a massive reworking to make possible.

The project has showed that it is possible to provide more advanced jobs control for the NorduGrid ARC without many changes. Should the Job Manager be used in a production system, some of the changes should be considered. Furthermore the Job Manager makes it simpler to develop applications for the grid, due to the choice of a protocol that is supported on a multitude of platforms and languages. The possibility of using handlers with the Job Manager also makes it an excellent platform for performing experiments with grid middleware, e.g., the performance of different schedulers.

Contents

Preface

This report is a master thesis written at The Department of Computer Science at Aalborg University. The main topic of study is distributed systems and the project is concerned about the development of an automatic job management tool for the NorduGrid Advanced Resource Connector (ARC).

The project is a continuation of work initiated in a previous project, and builds upon experience gained from this. Even though this project is a continuation, knowledge of the previous project is not a prerequisite, as the relevant concepts and ideas will be introduced and explained in this report as well. Readers who have already read the first report can skip Chapter 2 and Chapter 3.

The reader is presumed to have knowledge of distributed systems and computer science in general. Knowledge of computational grids or NorduGrid ARC is not necessary, as the concepts are explained in the report as they appear.

The project is written under the supervision of Associate Professor Josva Kleist, and we would like to thank him for input and constructive criticism as well as making it possible to meet the developers of the NorduGrid ARC. In addition we would like to thank Niels Elgaard Larsen, Jakob Langaard Nielsen and Anders Wäänänen for being of valuable assistance and providing us with input, as well, as the people on the nordugrid-dicuss mailing list for assisting us and answering questions regarding the middleware.

Henrik Thostrup Jensen

Jesper Ryge Leth

Chapter 1

Introduction

This project is in some respect a continuation of a project *Automatic Job Resubmission in the NorduGrid Middleware* [40], which was completed in January 2004. That project serves as the preliminary work to this project. In the previous project a client which handled resubmission in the *NorduGrid Advanced Resource Connector* (NorduGrid ARC), was developed and tested. This was mainly done as a proof-of-concept, setting the stage for the development of a more advanced job management tool. This project describes the design and development of a full featured tool capable of managing jobs running on a grid, providing extended functionality and job control to the user.

The NorduGrid ARC is a middleware used for connecting high performance computing resources to form a grid. By connecting resources into a grid, the entire range of resources resembles a giant batch system. This makes it possible to take advantage of all the resources by enabling the user to submit jobs, to the resources connected to the grid, through a uniform interface. Computational grids are a rather new technology being used in the roughly same areas as traditional super computing, but other areas are starting to appear. A more in depth discussion of these aspects as well as grid technologies and the NorduGrid ARC can be found in Chapter 2 and Chapter 3.

The NorduGrid ARC is primarily being developed to be used in the area of high energy physics and NorduGrid is one of several projects that tries to meet the challenge of creating an infrastructure, to analyze data from the Large Hadron Collider being build at CERN[1]. In order to test the different middlewares being developed, a series of test-scenarios, known as The Atlas Data Challenge, have been devised [24]. Two of these, Atlas Data Challenge 0 and Atlas Data Challenge 1, have been completed. It was the experience from this experiment that demonstrated the need for an automatic production management system [49].

The problems found by the users of NorduGrid ARC was, that it was the responsibility of the user to monitor the jobs running on the grid. In case of failure or other abnormal circumstances, it is the users responsibility to take the appropriate action [49]. During NorduGrids participation in the Atlas Data Challenge 1, a lot of time was spent manually keeping track of jobs, especially which jobs that had failed, and resubmitting them. This is a task that should not require human intervention and should be automated.

Using this problem as a basis for our work in the previous project, we developed a daemon, NG Proxy[2], that was able to monitor jobs and automatically resubmit them in

[1] The European research center for high energy physics

[2] Later renamed to NG Job Manager, due the general confusion about the word proxy in grid terminology.

case of failure. The project and the development process is described in [40].

1.1 Experiences From NG Proxy

The development of NG Proxy, provided us with experience regarding the software, and the usage of grid systems, especially the NorduGrid ARC. These lessons can help us and since we do not have to learn the basics of grid and the NorduGrid ARC; we can use the time to focus on creating a more usable solution.

If grid computing is to have success it needs users, after all without users, their is no interest in the grid. therefore no grid. Easy access to the grid is necessary for wide adoption of the technology by the end users. The problem is that users does not appear simply because the technology is there, even though they may benefit from it. They need some incentive or easy access in order for them to try it out and determine if it is useful to them.

The introduction of a automatic production system on which the applications and users can rely for easy access to the grid, would bring the users a step closer to the grid. The need for this type of production system is also to prove useful in many other application areas than high energy physics. Also it allows application developers to "gridify" their applications faster, as a clean interface (simple protocol and API) to the grid is provided, delivering a higher abstraction layer. The general idea of such a production management system is shown on Figure 1.1. The new system should be a

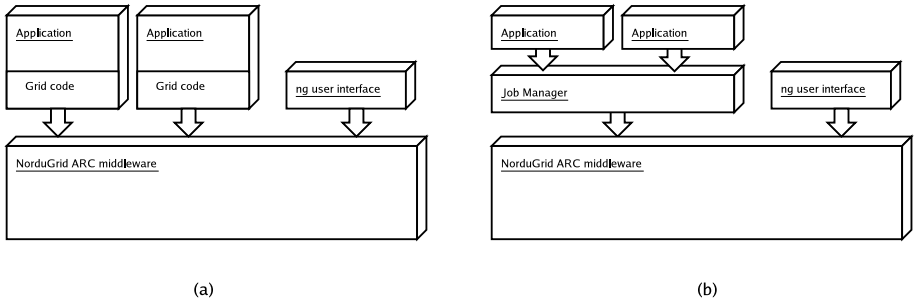


Figure 1.1: The Job Manager introduces an abstraction layer between the grid and the application, providing a clean interface to the application programmer and ease the development of grid applications by providing easy access to common functionality.

supplement to the existing user interface and not force a more complex system upon uses who have no need for it.

This approach of providing a tool working as a tier between the application an the grid software have been proposed and used with varying features and degrees of success by other grid projects such as PROGRESS [9], Nimrod/G [1], and the EDG Resource broker [21].

As previously discussed, NG Proxy was only developed as a proof-of-concept and thus there are some concerns regarding the further development on it, since it is not geared for this. Therefore an implementation should start from scratch. The need to start from scratch is also made apparent by the fact that during the development of NG Proxy, the list of wanted features grew; from conversations with people on the NorduGrid discussion mailing list. The list of features grew from a simple question of resubmission, to more intricate aspects of job control. The list also included features

such as controlling NG Proxy remotely and handling different kinds of failure. However, NG Proxy was created primarily as a proof-of-concept and thus it is not designed to handle the implementation of the features on the list.

1.2 Scope of the project

The project was started with a meeting with, the Danish part of the NorduGrid ARC developers. The ideas and thoughts from this meeting was written as a proposal for a Job Manager for the NorduGrid ARC, see Appendix A, and posted to the NorduGrid discussion mailing list for further ideas and comments. The feedback from this posting made it clear, that there was different expectations and use patterns for an automatic production system, but it helped determining the most important features and requirement imposed on such a system.

The overall idea of the design and implementation presented in this report is to provide a better interface to the NorduGrid Toolkit for applications to use and providing extended functionality for them. Currently applications must use a suite of simple command line tools, described in Appendix B, to interact with the NorduGrid ARC. This approach have several advantages due to its simplicity, but it also imposes significant limitations on the task, a user without any programming skills is able to perform. It is in light of this discussion, that we determine the scope and focus of this project:

> *"To design and implement a general way of managing jobs and automatizing control over them and provide a clean and simple interface for applications to use for interaction with the NorduGrid ARC. Additionally the Job Manager should provide extended functionality along with automation of trivial tasks, and provide assisting functionality to applications (clients) wherever possible."*

Having determining the scope of the project, a introduction to the concept of computational grids, and a tour of the NorduGrid ARC is given in the next chapters. Following this, the design and implementation of the Job Manager is explained and discussed.

Chapter 2

Computational Grids

This report is focused on the concept of computational grids, and in this chapter we will clarify and explain the main concepts and ideas, one may find during an exploration of the jungle of computational grids and industry buzzwords.

At the moment it is quite possible that the word grid means something different to everyone. Over the last couple of years it has been hyped, and everybody seems to have their own idea or understanding of what grids are. There does seem to be some consensus, though, that is has something to do with technology that utilizes distributed and/or parallel computing. For instance, when Sun Microsystems talks about grid, they primarily mean clustering and load balancing [46, 47]. Likewise Oracle mainly mean distributed databases [15, 48]. Because of this discrepancy it will be explained what we mean when talking about grids.

The term grid was coined in 1998, in the book "The Grid: Blueprint for a New Computing Infrastructure" by Ian Foster and Carl Kesselman, and it was used to describe a new computing infrastructure [31]. At the time the Grid was defined as:

> *"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high end computational capabilities.*" [29]

This was the beginning of the idea of "The Grid" a supercomputer made up of every computer on the Internet. An idea which still lurks in the head of a some people.

The name grid came from an analogy to the electric power grid, and the idea was to consider computational power as a resource, that could be traded in the same manner as electric power. On a historical note we observe that it was not the discovery of electricity, but the invention of the power grid that launched the electrical revolution and made electrical devices for everyone to own. The power grid enabled everybody to get as much power as they needed without having to pay large sums of money for their own generator.

A power grid is a dynamic heterogeneous infrastructure which consists of a network with a lot of different producers and consumers connected. The producers are power plants which vary with respect to price and contribution – from a nuclear power plant to a farmer with a single windmill in his back yard. The producers and consumers can appear and disappear from the grid without notice. When a producer disappears others take over, without disruption in the service from the consumers point of view. The power grid serves all types of consumers, from private households to million dollar corporations. All the consumers are connected to the same power grid, and uses as

much of power they need, and only pay for the amount being used. This is one of the key points to the success of the power grid; even though a power plant is very expensive and a large investment, the power is still relatively cheap to the consumer [30].

The original vision was to view super computers (computational cycles) and other expensive equipment such as analytical and special equipment as resources which everyone could have access to, by subscribing to the grid. This grid would consist of every computer connected to the Internet, essentially acting as one supercomputer which everyone could connect to, in order to use or contribute resources. By running software on the grid, CPU cycles and other resources can be traded just like power, and consumers are being billed for what they use, and producers are being paid for what they contribute.

But – analogies can be a dangerous thing, and computational power is not electricity, as they differ in several respects [30]. This is important to note this for a number of reasons. First computational power is a highly volatile resource and it cannot be stored for future use as electricity. This is partly true for some types of electricity as well, e.g., wind mills which produce power when the wind is blowing, but often the production of electricity can be reduced by delaying conversion of energy (e.g., coal and oil), until needed. This cannot be done for CPU cycles that are lost if they are not used immediately. This serves as a large incitement for developing ways to trade idle computational power, but also speaks against the analogy to electricity. The analogy to the grid is also not accurate in other aspects as there can be a number of different resources connected to the grid, not just computational power, but at wide range of different equipment.

Despite the difference, this was the vision of the people behind the idea of computational grids. Grid technology today is "a work in progress" and there are issues regarding distribution, security, scheduling, scalability, and almost every other problem imaginable when dealing with distributed systems on a large scale that has to be solved to implement the vision of "The Grid". Because of the the focus have shifted today, from trying to create one large grid toward constructing smaller and more specialized grids. An example of this is sharing of resources among smaller groups collaborating, but spread geographically, to try to solve a subset of the problems and make a grid that works, even though it is on a smaller scale and with a specific purpose in mind. Some predict that we will have a working grid, with these issues solved, in a matter of a few years, while others are not so optimistic.

2.1 Why The World Need Computational Grids

Under this rather pretentious heading, we will explain why we think that grids serves a purpose and are not just "hot air". We will do this by departing from the idea of a global supercomputer, and look at computational grids from a more pragmatic point of view.

The ordinary home user does not need a lot of computational power very often, and they have no real need for high performance computing facilities most of the time. In fact 70% of the time, a typical workstation in a corporate environment are idle [30]. However, in some situations even the ordinary user may need access to a lot of computing power, e.g., when checking his or her stock portfolio, or a similar demanding task, a lot of CPU time is needed to make more precise predictions of the rate of change. Here a subscription to a grid would come in handy by supplying on-demand computational power. Looking beyond the ordinary home user, the traditional users of high performance computing (HPC), would benefit from the added computational power of

a grid. This benefit comes from the grid enabling the sharing of expensive equipment between groups and organizations, for instance scientists located around the world to ease collaboration and lower costs. Below we have summarized what seems to be the main reasons for developing computational grids.

- **We have bigger problems** – The computational problems that needs to be solved have become bigger, or the idea of solving problems computationally has become more wide spread leading to problems which needs more resources to be solved. Even though we have bigger problems, our own resources are not always fully used, but costs almost the same amount of money to operate even when idle.
- **We need to save money** – HPC resources are expensive, and the acquisition and operation costs can be brought down by sharing these resources in a grid. The use of grid technology also enables users to solve problems faster than before, due to extended use of resources. If institutions with similar problems is given the ability to pool their resources they can solve a given problem faster.
- **We do not always have computational power at hand** – PDAs and other embedded devices have become more widespread during the past years, and this trend seems to continue. However PDAs still has limited computing power, and having an infrastructure that enables access to HPC resources from a PDA would be beneficial and enable pervasive use of high performance computing.

One of the requirements for these areas are the sharing of resources and we will look further into the concept of virtual organizations. A way to enable corporation and sharing of resources.

2.2 Resource Sharing and Virtual Organizations

The vision of creating an infrastructure that would bind together every computer on the Internet into a single giant supercomputer is ambitious. From an administrative point it raises a number of issues regarding security and sharing of resources. Not only do we need to share our resources, but we also needs to allow foreign code to run on our machines without a chance to review it. Not everyone would grant everyone access to their computational resources, and controlling access in a single giant grid can become troublesome, due to disagreement of who will get access and to what. This concern was addressed by Ian Foster and Steven Tuecke [31] and they modified the grid definition to take political and social issues into account.

> *"The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which the sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a* virtual organization." [29]

This definition introduces the concept of a virtual organization, which has become a fundamental concept of resource sharing in modern grids.

A virtual organization is a group of institutions and users, who have decided to share resources with each other. To do so they form a virtual organization, thereby establishing a relation of trust between the users and organizations, forming a grid. The users in the organization is able to use resources shared in this virtual organization. An institution, user or resource is not necessarily tied to a single virtual organization, since an institution may choose be a part of several virtual organizations, sharing some resources to one, some to another, and some resources too more than one virtual organization [31]. This is illustrated on Figure 2.1.

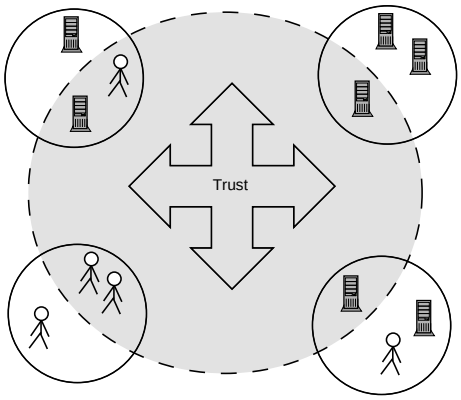


Figure 2.1: A virtual organization consists of users an resources from several organizations, that may not have anything in common on the organizational level, apart from belonging to the same virtual organization.

A running grid will have resources disappearing (e.g., hardware failure or resource withdrawal), and appearing. The situation is the same for users, since users can get access to a grid, and have it revoked as well. This means that grids will have users and resources appearing and disappearing, yielding a highly dynamic structure. Some virtual organizations will be more dynamic than others, e.g., a group of scientists may form a grid to analyze data. The duration of such a virtual organization could last for years, and only have a few users entering or leaving, while other virtual organizations may be far more dynamic, e.g., a virtual organization could grant access to every student at a certain institute. This dynamic nature imposes certain requirements upon the infrastructure. It must support an easy and automated way of registering new users and resources, while being able to cope with disappearance of users and resources.

Virtual organizations will most certainly play an important role in the future of grids, since they concern two of the most important things in grids: Users and the sharing of resources. Making virtual organizations easy to create and maintain are an important aspect, among many, to have in a grid infrastructure, if they are to become widely spread.

From this short explanation of a line of fundamental computational grid concepts, we now examine the fundamental architecture of computational grids.

2.3 Grid Architectures

This section discuss various grid architectures and considerations that arise when building grids. We start by looking at a general model of a grid architecture and move on to

issues that must be considered when building a grid. We finish by giving examples of existing grid environments. The discussion in this section is based on the discussion in the previous chapter along with the models and discussions presented in [30, 31, 35]. It outlines a set of basic properties and capabilities that a grid architecture must provide.

One of the things to be aware of when designing a grid architecture, is the entities in a grid environment. In principle the main entities of interest in a grid are users, resources, and jobs. These entities have characteristics and requirements which must be taken into considerations.

- **Users** – are geographically spread, they are in complex sharing relationships with organizations and other users. They require ease-of-use, authorization, authentication, trust, and needs access to several grids, assured a certain quality of service.
- **Resources** – are heterogeneous, dynamic, and geographically spread. They are not necessarily dedicated to grid jobs and requires fine grained access control.
- **Jobs** – belongs to different users and may consist of mobile code, foreign to the computing element. They may have secret content imposing security needs and may need specific runtime environments to be installed.

The above list is only some of the characteristics and needs, of the entities involved in a running computational grid, but they illustrate the complexity of such a system.

Looking at the definitions from the previous chapter, it is obvious that some of the key elements in a grid is sharing, decentralization of control, and heterogeneous resources. As an example of the problems and complexities when running in a grid environment, a thing as the simple operation of executing a program on a grid is nontrivial, as the input, output, and data has to be set up prior to the execution [35]. Fundamental requirements for computational grids are.

- **Scalable** – The fundamental idea behind computational grids are that they are, or at least will become very large. Thus it is important that the architecture and technology upon which grids are build scale very well.
- **Robust** – As for other distributed systems the architecture must be robust. A grid must be fault tolerant and cope gracefully with network and resource failures, providing consistent and dependable quality of service.
- **Secure** – A grid architecture raises almost any security issue conceivable. Since a grid has no central control, and may span over several administrative domains, the security requirements upon the architecture are important. This is emphasized by the fact that the resource owners allows users to execute foreign code their resources.
- **Pervasive access** – This covers several issues of grid access. Access must be easy, with respect to user credentials (e.g., single sign on). Access must be provided from a wide range of computational devices and must be inexpensive.
- **Accounting** – There must be a reliable accounting system keeping track of resource usage, making it possible to charge the users. This includes accounting with respect to resource usage and contribution. Accounting and payment raises issues about quality of service, and a grid should facilitate a way to measure and assure this.

These are only some of the issues and list goes on, other important issues are: control, interoperability, open protocols, consistency of service, and scheduling policies.

The main purpose for the architecture are to conceal the heterogeneity and complexity of the underlying resources. Foster and Kesselman [30] points out that a grid architecture is first and foremost a protocol architecture and its goal is to ensure interoperability. Furthermore the architecture must facilitate fine grained access control over the sharing of resources. There are discussions [31, 35] of whether new programming models is needed and if these should be implemented in therms of basic grid protocols.

2.4 The need for Interoperability

Since users and resources can be members of several virtual organizations, there is a clear need to have common protocols [31]. Having separate protocols for each grid middleware is not a feasible option if virtual organizations are to be created quickly and maintained easily. Furthermore a user or resource could be member of several virtual organization, making interoperability almost impossible without common protocols. Additionally these protocols should be standardized such that different grid middlewares can be created, while still being able to talk together; much like the IP protocol works today.

Writing such a middleware is not an easy task, so most grid users will use an existing middleware to create their grid application. Some users probably want to extend their middleware, with some specialized services or access to uncommon resources. Such users should not have to create their own middleware, but rather extend existing middleware to fit their needs. This means that the code, for at least some middlewares, should be open.

Given that there will exist several middlewares for grid applications, these should be accessible in a uniform way. This means that they should provide a similar API to the application programmer. However, as described above, middlewares will differ in functionality. Due to these differences it would be impractical for all middlewares to provide the same API. Instead they should aim to provide the basic API. This would mean that grid applications could be ported between different middlewares, without too much effort. This may not be a realistic goal since there are many different ways of solving the grid problems as we will see in the next chapter. However standards are being developed which should ensure interoperability.

2.5 A Generic Model for a Grid Architecture

To address the issues just described, a grid environment can be described as a set of abstract levels. These can roughly be divided into three parts: Core grid, services and user interface [35]. The core grid consists of the resources and applications running on the resources. The services provides a homogeneous interface to the resources, as well as facilitating discovery and resource brokering. The user interface supplies the user with a way to interact with the grid services. This interaction can be facilitated in many ways, ranging from a standard Unix shell augmented to support the functionality of the grid, to a web portal interfacing to the grid services. Furthermore a grid environment should fulfill two main functions: Provide user-side programming and control the user interaction.

We will now take a look at a more detailed model of a grid architecture, and identify the features necessary at the different levels. The model was originally presented in the article “The Anatomy of the grid” [31] and can be seen on Figure 2.5. This model is

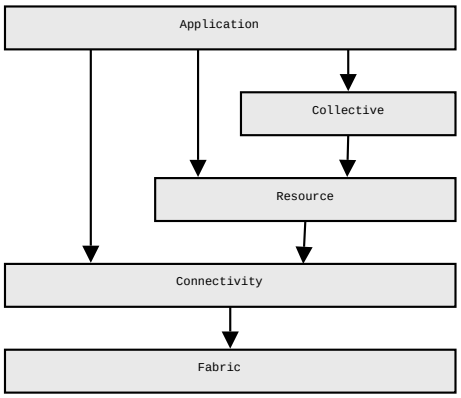


Figure 2.2: A generic model of a grid architecture showing the different abstraction layers common for many grid architectures.

made up of a set of abstraction levels as previously discussed. Starting at the bottom we have the Fabric. This is the level that interfaces with the local resources and provides shared access. The resources in this level may be either a physical or a logical entity, e.g., a cluster or distributed file system. The fabric level supports local resource specific operations which are dependent on the operations of the higher levels of the model. There is a trade off concerning functionality on this level. A richer set of functionality may make advanced sharing functionalities available for the higher level, but at the same time making deployment of new resources more complex. As a minimum, this level should support a mechanism enabling discovery of services and capabilities and a resource management mechanism delivering some control over the quality of service.

The connectivity layer defines the grid related protocols providing the necessary grid specific functionality like communication and authentication protocols. The communication protocols should enable exchange of data between fabric resources, including routing, naming, and transport. Much of this can be achieved trough existing protocols, e.g., TCP/IP, and DNS. The authentications protocols should also, due to complexity and security issues, rely on existing protocols and provide support for virtual organizations, supporting single sign on, trust relationships, and delegation. The authentication protocols should integrate well with existing protocols and systems.

The resource layer is relying on the connectivity layer supplying protocols for negotiation, monitoring, control, and accounting operations on the individual resources. The functionality of this layer can be split into two classes; information protocol and management protocol. The protocol layers form a bottleneck in the model and should therefore be kept as small and simple as possible while still supplying the needed functionality.

The collective layer are focused on the global resource view and contains services and protocols not associated with any single resource. The collective deals with relationships and interactions between resources. This layer implements services, including directory services, scheduling, monitoring, and accounting. These protocols are general in nature and rely on the services of the underlying layers to implement the

needed functionality with respect to the individual resources. The functions in this layer can be implemented as services with associated protocols, or as software development kits with associated APIs. The collective can be developed for specific use (e.g., specific VO requirements) or for a more general purpose

The application layer is the applications that run within a specific VO environment and applications at this level makes use of the services of the lower levels by means of well defined protocols and APIs. We can now specify what we mean when we use the term grid, we mean:

> *An infrastructure that enables controlled sharing of computational resources across sites and trust boundaries. Between users from different organizations and institutions which may be geographically spread. The resources belongs to the institutions and users who remain in control of their own resources. The users of the grid have the possibility to securely run jobs on the shared resources and the resource owners has the ability to charge the users for usage of the resources.*

Ultimately all of these requirements are needed in order to be talking about grid, However if a fairly large subset are met, e.g., support for accounting could be absent, we still use the term grid. This is reasonable because not many – if any – grid environments support all the requirement mentioned.

2.6 Grid Environments

This section examines existing types of grid toolkits that are in use today. We have selected examples among the many that exists today. The reason for selecting the ones described in this section is, that they represent some of the major different approaches for constructing grids. For a more extensive list and a description of the grid environments available today, we refer to [35].

Before discussing the various environments, we start by briefly describing the Open Grid Server Architecture. Even though there are much discussion about standards for computational grid protocol, there are surprisingly few. One of them is the Open Grid Service Architecture (OGSA), which is an attempt to establish a common standard for grid architectures. OGSA is based on web services and the grid services provided by OGSA follows the Open Grid Service Infrastructure (OGSI) [74], meaning that every service is a web service[1]. Many projects seems to embrace the standard and develop their toolkits accordingly [48, 56]. The newest in this family is the *Web Service Resource Framework* (WRSF) which has just recently been proposed, it should be seen as a successor to OGSA/OGSI, but the specifications has yet to be finished.

2.6.1 The Globus Alliance

The Globus Alliance is a research and development project with participation by several universities and large companies around the world [57]. They do not produce a grid environment, but instead the main focus of the project is to develop fundamental grid technologies needed to build a grid. The result of the project is a grid toolkit called the

[1] Web services provide a standard means of inter operating between different software applications, running on a variety of platforms and/or frameworks. The interaction is made possible by using protocols and technologies such as SOAP and XML [13].

Globus Toolkit. This toolkit is a set of building blocks for creating grid middleware, meaning that it is not a complete grid solution, but rather a framework for building grid applications and solutions.

At the time of writing, there exists two major versions of the Globus Toolkit; version 2 and 3. Lately a fourth version has been announced, but it has yet to reach a stable incarnation [3]. The toolkits and the code for them are freely available on The Globus Alliance web page[2]. In the following sections, the different version of the Globus Toolkits is described.

2.6.2 The Globus Toolkit 2

The Globus Toolkit 2 was a continuation of version 1, and it has had major revisions in several areas [56]. The toolkit defines its own set of communication protocols, meaning that it cannot easy communicate with other grid middleware. However over the past six years the Globus Toolkit 2, has evolved into becoming the de facto standard for computational grids [29]. This is likely due the large number of institutions which has build their grid solutions on the Globus Toolkit and today it is one of the most mature grid toolkits. The reasons for basing a grid solution on the Globus Toolkit are, that the toolkit can be downloaded for free, and that the code is freely available, making it possible to tailor it to suit your needs.

The toolkit it build of three major parts [59]: Resource management, information services and data management. Resource management is concerned with allocation and management of resources. Information services is the part that provides information about the resources in a grid, making it possible to query the information needed. The last part, data management, is concerned with access and management of data. As of this writing the Globus Toolkit 2, is in version 2.4.3, which was released September 11^{th}, 2003.

2.6.3 The Globus Toolkit 3

The Globus Toolkit 3 is relatively new as its first release was in July 2003. This version is a major redesign of the previous Globus Toolkit. It was redesigned to create an implementation compliant with OGSA standard [28]. The functionality of the services provided by Globus Toolkit 3 corresponds to the services provided by the Globus Toolkit 2, but they are implemented as web services in order to comply with the standard. The reason for implementing grid services as web services are extensibility and manageability in contrast to the services in Globus Toolkit 2, which are separated and independent. This means that it takes a significant amount of work to implement a new service or change an existing one. The Globus Toolkit 3 provides a framework for this, so existing OGSI services can be modified more easily and new services can be created faster [58]. Furthermore using and managing grid services has become uniform through the use of web services.

To accommodate migration from Globus Toolkit 2, several steps has been taken. Globus Toolkit 3 contains the same components as version 2, and has API compatibility and the same form of authentication is used[3]. This makes it possible for existing authentication and authorization mechanisms and credentials to be used.

[2] Homepage at http://www.globus.org

[3] X.509A certificates, which is a widely used certificate standard. [27]

Globus Toolkit 3, is currently in version 3.2. The additions of this release compared to 3.0 is bug fixes, performance improvements, new features and a new documentation structure [4].

Even though Globus Toolkit 3 has been released, there is a continued development on Globus Toolkit 2. This is due to the many existing projects which has been based on version 2, and that version 3 is still relatively new. However the links to Globus Toolkit 2, on the Globus homepage, are becoming increasingly difficult to find, indicating a desire to move people from Globus Toolkit 2 to Globus Toolkit 3 (or more likely; Globus Toolkit 4).

2.6.4 The Globus Toolkit 4

On January 20^{th}, 2004, the WS-Resource Framework (WSRF) was introduced [3]. WSRF is basically a refactored version of OGSI, an addition of some new features in web services [26]. Furthermore the specification has been split into six parts, where drafts exist for three of them [6]. Although WSRF is heavily inspired by OGSI it is not compatible with it [6]. The Globus Alliance has not yet produced a working toolkit for the WSRF, but work is underway to port the Globus Toolkit 3 from OGSI to WSRF. We believe that the introduction of WS-Resource Framework will keep grid projects away from using the Globus Toolkit 3, since OGSI is essentially dead after the introduction of WSRF, although the Globus Alliance says that this is not the case [6].

Whether or not grid computing will converge to web services, using OGSI, WSRF or a third possibility, still remains an open question. However, if the Globus Alliance wants their Toolkit to succeed they will surely need to settle on a standard and produce a stable toolkit.

2.6.5 Enabling Grids for E-science in Europe

Enabling Grids for E-science in Europe[4] (EGEE) is the successor of the European Datagrid (EDG), which is an example of a grid environment based on the Globus Toolkit 2. EDG was an initiative, doing research in building a common European grid solution. It was funded by the European union and has a budget in the range of 10 million euro. EDG was led by CERN, and it was a collaborative effort with several European research agencies including the European Space Agency and national agencies from several European countries [21]. EDG was trying to construct, not just a toolkit, but a complete grid suite and did work on applications in several areas. The development was divided into four major areas: Testbed and Infrastructure, Applications, Computational and Data Grid Middleware, and Management and Dissemination. The application areas that EDG was aiming at is High Energy Physics, Biology and Medical Image processing, and Earth Observations.

The EDG project was finished as of March 2004 and many of the technologies have passed to EGEE, who are creating a grid infrastructure to support the research area. The EGEE solution is not based on any existing middleware and relies on a SOAP interface defined by the EGEE.

[4] www.eu-egee.org

2.6.6 The NorduGrid ARC

The NorduGrid Advanced Resource Connector[5] (NorduGrid ARC) will be examined in detail in the next chapter, but for the sake of completeness a short description follows. The NorduGrid ARC is a grid solution based on Globus Toolkit 2. It is the result of a collaboration between the Scandinavian countries to create and operate a Nordic computational grid. The purpose of the NorduGrid project was to create a testbed for a Nordic grid infrastructure and is a collaborative effort with participating research centers from Denmark, Norway, Sweden, and Finland. The focus was to create an infrastructure for future high energy physics experiments, but this goal has been expanded over time. Even though the main focus still is high energy physics, the possibility of running other applications on NorduGrid is being examined. At the time of writing NorduGrid consists of clusters located at the participating organizations, but more and more sites and countries are participating. Since the summer of 2003 up till today the number of CPUs on the grid monitor has gone up from below 1000 to more than 2400. In this time frame, NorduGrid has moved from being a testbed to a production grid.

The NorduGrid approach is to base the toolkit on Globus Toolkit 2, reusing as many components as possible. However it has been necessary to extend some of the Globus components and replace others in order to get the desired functionality. Contrary to many other grid toolkits in existence NorduGrid is not planning a move to Globus 3, although it has been discussed.

Due to the collaboration and the creation of the NorduGrid ARC, the local grid research centers in the Nordic countries, including Danish Center for Grid Computing (DCGC) and Swedish National Computational Resources (SweGRID), are basing their research and work on the NorduGrid ARC. This has an impact on our choice of toolkit, making NorduGrid the natural choice, since we can get access to resources running the NorduGrid ARC through DCGC.

After having looked at Globus and two toolkits based upon it, we move on to looking at Legion, representing a radically different approach to building a grid .

2.6.7 Legion

Legion[6] is described as a world wide virtual computer. The Legion project is based at the University of Virginia; the goal of the project is:

> *"Users working on their home machines see the illusion of a single computer, with access to all kinds of data and physical resources, such as digital libraries, physical simulations, cameras, linear accelerators, and video streams. Groups of users can construct shared virtual work spaces, to collaborate research and exchange information."*

This is very much in accordance with the original vision of The Grid, making many computers act as one supercomputer. As opposed to Globus, Legion is trying to created an integrated solution and does not use preexisting services and technologies. Instead Legion are using an object oriented philosophy toward designing and constructing a grid. Legion is build on top of a unified object model developed specifically for Legion [54]. In Legion everything is an object, and Legion defines the message format and high-level protocol for object interaction, but not the programming language or

[5] NorduGrid ARC was formerly known as the NorduGrid Toolkit, but it was feared that this name would prevent it from getting acceptance outside Scandinavia.

[6] Homepage at http://legion.virginia.edu.

the communications protocols. It is possible for users to provide their own classes, since the common services are implemented by core objects. Additionally Legion offers PVM and MPI libraries which applications can be compiled against in order for the application to take advance of the infrastructure.

The main objectives of the Legion project has a lot in common with other grid projects. The aim is to create a scalable and fault tolerant architecture that takes care of the management and utilization of resource heterogeneity. When running Legion the user has, what Legion refers to, as a context space in which all applications, belonging to that user is executed. That, along with a virtual file system and a resource management system allows the user to run processes on the grid. Legion does this by extending the basic capabilities of the Unix shell to work with the distributed object file system [35].

The project lists a set of constraints under which the Legion software must run. These are very common for grid projects. For instance the host operating systems cannot be replaced, as well as changes to the interconnection network cannot be legislated by Legion. Furthermore Legion cannot be required to have superuser privileges and Legion should work while keeping site autonomy and ensure security for users and resource owners. Legion works on top of the users operations system and negotiates security and scheduling policies with the different sites according to their policies. These requirements ensures that site autonomy are kept by maintaining and adhering to the local policies, whether it is being security, resource, or other policies.

These objectives and constraints are very similar to many most grid projects, and when examining NorduGrid design philosophy in section 3.1 we will see that many of the same objectives and constraints are valid for NorduGrid as well. The main difference is that the approach to solving the problems that differs and not the fundamental goals as many of these goals are paramount when dealing with distributed systems that cross trust boundaries.

2.7 Summary

We have been looking at several different grid environments and toolkits. They each use different approaches trying to solve “the grid problem”. These approaches has their advantages and disadvantages. One of the main differences between Globus 2 and Legion is that Globus 2 does not have an underlying component architecture [53]. This approach yields advantages as well as disadvantages.

The main problem with the Legion approach is that every piece of software ever to run on Legion must be build specifically for Legion or otherwise ported and linked against specific Legion libraries. Legion tries to accommodate this by providing special Legion enabled versions of a number of well know APIs such as MPI. However having a unified name space and resource abstraction where everything is an object makes grid specific application development easier. It also serves as a basis for tighter integration between the applications, and the grid middleware making this approach closer to becoming “The Grid”. This closer integration should also make the construction of interactive grid applications possible. A thing that is very difficult within Globus based grid environments, however this comes at the price of more complex deployment.

The Globus Toolkit however, is not without advantages. Looking at NorduGrid it it in principle a very large batch system and resource broker. This batch like behavior is enough for a number of applications, especially in the area of traditional high performance computing, where the problems and applications can easily be distributed, e.g.,

parameter studies. These types of applications does not need human interaction, but raw computational power. In this case the advantage is, that once the job is submitted to the grid, no extra communication is needed, and the communication overhead in tighter integrated system is reduced. Additionally a lot of existing applications can be brought to run on this type of grid without significant modifications.

Thirdly there is the web service model as introduced by OGSA and used in Globus Toolkit 3 and 4. This model can in some way be considered the middle ground between the two other toolkit models. It offers tighter integration than Globus 2, by supplying an object model. This model is not as tightly integrated as Legion, because the interactions between the components is defined on a lower level using standard protocols (SOAP) and common data descriptions (XML). The looser integration compared to Legion can be seen by, e.g., the lack of a distributed file system and lack of a uniform name space.

It is hard to say if there are a “right” way to build grids, and we believe that the world is big enough for all three types of models, especially since they address different type of user and application needs. In this project we work with NorduGrid ARC due to reasons already explained. Therefore we will describe this toolkit in further detail in the next chapter.

Chapter 3

The NorduGrid Advanced Resource Connector

This section takes a deeper look into the NorduGrid Advanced Resource Connector (ARC), and examines the individual components, in order to get a better understanding of how the toolkit works.

The NorduGrid ARC was first known as the *Nordic Testbed for Wide Area Computing and Data Handling*, or the shorter name NorduGrid. The toolkit developed by NorduGrid, the NorduGrid ARC, is the grid middleware, which we have chosen as the basis for our work in this project. NorduGrid started as a testbed, but have since gone through reorganization of the organizational structure in order to become a production grid rather than a testbed. The NorduGrid organization has changed its name to Nordic Data Grid Facility (NDGF) and the NorduGrid Toolkit has changed its name to NorduGrid Advanced Resource Connector (NorduGrid ARC). NDGF is part of North European Grid Consortium, its job is to coordinate connection and usage of the Nordic grid, and control the agreements with organizations who wish to use the Nordic grid [11]. Internally NDGF coordinates the contributions to the Nordic grid by the various national grid facilities and serve as certification authority handling authentication, authorization, and accounting issues. The group maintaining and developing ARC is still be called NorduGrid [52].

The purpose of the NorduGrid project was to create a testbed for a Nordic grid infrastructure, and is a collaborative effort with participating research centers from Denmark, Norway, Sweden, and Finland. At the time of writing NorduGrid consists of clusters located at the participating organizations in the different countries. The main purpose was to create an infrastructure able to handle and analyze data from high-energy physics experiments.

The NorduGrid project was started in May 2001, in response to the ATLAS Data Challenge. The ATLAS Data Challenge (DC) is the name of the first Large Hadron Collider (LHC) application to be executed in a computational grid environment. It is a series of challenges to test the computing infrastructure. The ATLAS Data Challenge was created in order to prepare for the intaking of data from the LHC being build at CERN, when it goes into commission in April 2007. The LHC being build at CERN is the largest of it's kind and sets new standards for the amount of data generated by high-energy physics experiments. When operational the LHC is expected to generate data in the magnitude from 100 Mb to 1 Gb/sec adding up to more than 1 Pb/year [50].

The first two parts, the ATLAS DC0 and DC1, have already been completed. DC1 was initiated in July 2002 and ran through the first part of 2003. During this time more than 2 TB input data was processed and more than 2.5 TB output data was produced by more than 4750 grid jobs [24]. DC2 have been delayed but it is at the time of writing in preparation and expected to start during June 2004.

3.1 The NorduGrid Design Philosophy

We start by outlining the fundamental design philosophy behind NorduGrid, as it was formulated when the project was started. NorduGrid should start out by being build on tools and technologies that actually works and proceed from there, in an effort to construct a scalable grid architecture, without single points of failure. For NorduGrid to be dynamic in creation of VOs, and to run on sites that is not dedicated to grid jobs, it is important that the owners of the respective resources retains full control over their own resources, local policies and configurations. To achieve this, the NorduGrid middleware should impose as few site requirements as possible, i.e., there should be no dictation of cluster configuration or install method. Furthermore no dependencies on particular hardware should exist and NorduGrid should reuse the existing system installation as much as possible. The computational unit of choice in NorduGrid is the cluster. The NorduGrid ARC software should only be required on front end machines and the computing nodes nodes should not be required to be on a public accessible network [50]. To summarize the goals.

- Avoid single points of failure.
- The architecture should be scalable and able to cope with a highly dynamic resource pool.
- Resource owners should retain full control over their resources.

In the initial phase of the project, existing grid middleware packages was examined. The two main candidates where Globus Toolkit 2 and EDGs Data Grid. These were analyzed further, and both of them found to be inadequate. The Globus Toolkit 2 did not support resource brokering, and it also lacked the middleware for staging large input and output data files. The EU Data Grid seemed to address these issues, but was at the time (early 2002) considered to premature to be the basis for NorduGrid. In addition it had a centralized resource broker which was seen as a bottleneck and a single point of failure.

In light of the result of the analysis it was decided to build the grid infrastructure from scratch. In practice the developers have been using the Globus Toolkit 2 as the basis of the development, addressing the various issues, and adding components that either replace, extends, or complement existing Globus components.

3.2 Task Flow in the NorduGrid ARC

This section describes how the NorduGrid ARC is usually operated from the users perspective. This is done to give the reader a “feel” for how the toolkit works. We will go through the preparation, submission, and retrieval of jobs and job data. The first section goes quickly through the usage of the NorduGrid ARC, without explaining in

detail how the elements work. This is done later in this chapter. Even though we will not go through the installation procedure, we will note that the installation of the entire Globus Toolkit is necessary for the NorduGrid user interface to work.

3.2.1 Job Preparation

The first thing needed in order to submit a job to the grid, is to have access to one or more resources, i.e., the user must be a member of a virtual organization with access to a set of resources. The access is based on a user certificate issued by the certificate authority.

In order to execute a job to a cluster connected by the NorduGrid ARC middleware, the user must first prepare a job description. This description supplies information to the user interface, which is used to locate a cluster on which to execute the job. The description contains information needed to run the job on a cluster, including name of the executable, input data, location of input data, filenames, and location of output data, as well as other requirements, e.g., libraries required to run the job. The description is created in a language, the Extended Resource Specification Language (xRSL), designed for describing grid jobs. Since there are no screen or terminal to display input or output, when executing the job on the grid, all input files must be specified as files, and all output must be redirected to files. The job description specifies the input files and to which files the output must be redirected.

3.2.2 Job Submission

Before submitting a job to the grid, the user interface needs access to the proxy certificate, to be able to authenticate the users against the different grid services. This is done by running the program `grid-proxy-init`. Contrary to what may be suspected by the name, it does not start a program (a proxy), but generates an X.509 proxy certificate which expires after a predefined amount of time. The proxy certificate is used to sign the job description to determine the identity, and credentials of the user. By using a time limited certificate the severity of a compromised certificate is lessened, because it will eventually expire. The job is submitted via a command line user interface, which locates a suitable cluster for the job, i.e., one which fulfills the jobs requirements, and where the user has the privileges to execute jobs. When a cluster is found, the job is submitted. The submission process consist of uploading the job description and any local input files, and the user interfaces terminates.

3.2.3 Job Processing

Once the job description is uploaded to the cluster, the Grid Manager checks every two minutes (default) for the arrival of new jobs. The grid manager submits the job to the Local Resource Management System (LRMS), and waits for it to finish. When a job is completed or failed, the user can choose to be notified by email, or can check the status of the job manually, either through the grid monitor on the NorduGrid website, or through the command line interface, which queries the information system. When the job is completed, the grid manager does the post processing of the job data according to the job description and optionally moves the output files to a storage element for later retrieval by the user. The task flow in the job submission process is as follows:

1. The user creates a job description in xRSL.

2. The User Interface interprets the job description in order to perform resource brokering. It queries the information system to locate a resource to which the job can be submitted.

3. The job description is submitted to the grid manager on the chosen cluster via GridFTP.

4. The grid manager creates a session directory for the job data on the cluster.

5. The grid manager handles the preprocessing of the job data. If the job description states that data should be fetched from a storage element, the grid manager fetches the data, and makes them available within the session directory.

6. The job is submitted to the local resource management system (LRMS) for execution.

7. The grid manager performs post processing of the output data. The grid manager can optionally register the data with a Replica Manager. Upon job completion the user can be notified by email.

8. The user downloads the data from the cluster, using the User Interface or by GridFTP.

9. The grid manager deletes the job within a given time frame, if the user has not removed it.

On Figure 3.1 task flown and the various protocols in the communications in the NorduGrid ARC are illustrated. Along with the protocols used for the different task. The details of the components will be discussed in detail in the rest of this chapter.

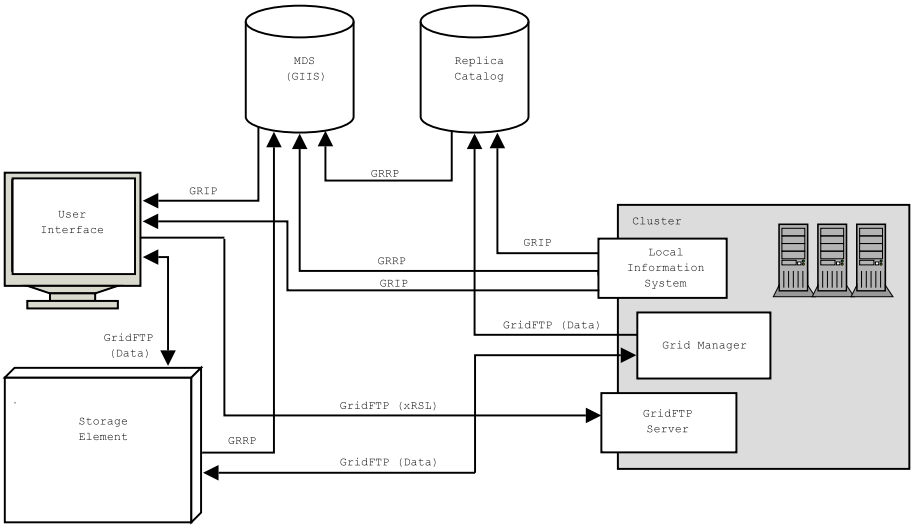


Figure 3.1: The communication and the various protocols handling the communication in NorduGrid. GRRP is the protocol used by resources for registering contact information with the information system (MDS), and GRIP is the protocol used for querying resources for status information. GridFTP is used for both transfer of job data, and the job description itself.

3.3 NorduGrid Middleware Components

In this section the components of the NorduGrid toolkit is explained. This is done in the same order as they show up in the task flow just described. We begin by looking at the extended resource specification language.

3.3.1 Extended Resource Specification Language

Extended Resource Specification Language (xRSL) is an extension to the Resource Specification Language supplied by Globus. It has been extended for use with NorduGrid and not all Globus attributes are supported. This mostly pertains to attributes concerning Globus Resource Allocation Manager (GRAM), which is replaced in the NorduGrid ARC with the Grid Manager. The language is divided into two levels: User-side xRSL and grid manager-side xRSL [68]. The user side part of the language is the description which is prepared by the user and send to the user interface. This part describes the job and its attributes. An example of such a description can be seen below.

```
&(executable=/bin/uname)
 (arguments=-a)
 (stdout="out.txt")
 (stderr="err.txt")
 (outputfiles=
    ("out.txt" "" )
    ("err.txt" "" ))
```

This description shows a very simple job, executing the Unix command `uname` that prints various information about the node it is executed on. The first attribute is the name of the executable, and the second is the parameters it should be executed with. If the executable is not native to the cluster and has to be transferred, this can also be specified. The next two attributes states where the output from the program should be redirected and the last two are the files that the user would retrieve after the job is finished, or optionally that the Grid Manager should upload to a storage element or register with a replica catalog. Apart from the attributes in the example, other attributes concerning disk space, runtime environment, middleware version, cluster, and many other attributes can be described [68].

The grid manager side of xRSL is used internally when the user interface submits the job to a cluster. It specifies attributes pertaining to the network, the submitting user, etc. Some of these attributes can also be specified by the user, though, this is not advised.

3.3.2 The User Interface

The User Interface is the major new component added by the NorduGrid ARC. It delivers the high level functionality needed by NorduGrid, but which is not supplied by Globus. The user supplies functionality for handling resource discovery, resource brokering, job submission, status querying, retrieval of job data, job control, and other necessary functions for interacting with the grid. When a job is submitted through the user interface, it parses the accompanying xRSL job description in order to locate a suitable cluster to submit the job to. After retrieving a list of available clusters from the

information system, the user interface queries the information system, to check if the user are allowed to submit a job to the cluster and if the cluster fits the job requirements. Then the user interface determines which cluster to submit the job to, using an internal scheduling algorithm, based on the number of total and free CPUs on the clusters.

When a suitable cluster is found, the xRSL job description is stripped for user interface information, and Grid Manager-side information is added to the description. After this, the job is uploaded to the cluster using GridFTP. There is no need for additional services in order for resource brokering to work. Optionally extra data can be uploaded by the user interface, or it can be left to the Grid Manager to fetch it from a storage element, by writing this in the job description.

3.3.3 Information System

For a system as complex as the NorduGrid ARC to work, it is important to have a robust, scalable and reliable information system, to store information about users, resources, and jobs. The information system in NorduGrid is implemented as a distributed service, serving information to the other NorduGrid services and components. It is build upon the Monitoring and Discovery Service (MDS), an information system framework supplied by the Globus Toolkit. The information system is essential to the NorduGrid ARC, and it takes care of all information related tasks. MDS is an extensible framework, provided by Globus, for creating grid information systems based on OpenLDAP. The information system consists of the following.

- An information model described by a LDAP schema.
- Local information providers.
- Local databases.
- Soft registration mechanisms.
- Information indicies.

The information model supplied by Globus is single machine oriented, and not suited to describe clusters very well[1], so an information model was created specially for NorduGrid. The NorduGrid information model is a mirror of the architecture, and it describes the main components of the grid, i.e., clusters, jobs, and users. These elements are mapped onto an LDAP tree that forms a hierarchical structure of queues where every user and every job has an entry. Replica managers and storage elements are described similar, but in a simplistic manner.

The Information System consist of a dynamic set of distributed databases which are coupled to information providers residing on the clusters. In NorduGrid a single MDS service is run per resource, and the task of this service is to provide status information about the specific resource on which it is located. Each resource operates its own Grid Resource Information Service (GRIS). These resources can be grouped together in order to form a virtual organization. This VO structure is called an MDS-tree and it is the actual mapping of the resources in the grid onto the information service.

The information providers are small programs that generates LDAP entries in the database upon search request. The NorduGrid ARC provides its own information

[1]The EDG information model was also considered, because it was better at describing clusters, but there were doubts about its practical use.

providers. They serve as interfaces to local systems, collecting information about the status of a job from the clusters LRMS and the grid manager. This information can be used to find information about the resource, such as available CPUs, disk space, and effective queue length. NorduGrid provides access to two queues: NorduGrid-authuser and NorduGrid-jobs. Authuser contains information about the CPUs available for the user, disk space and effective queue length. The job queue describes the jobs submitted to the cluster, i.e., status, job id, certificate, and owner. An example of an MDS tree containing information about users and jobs queues is depicted on Figure 3.2.

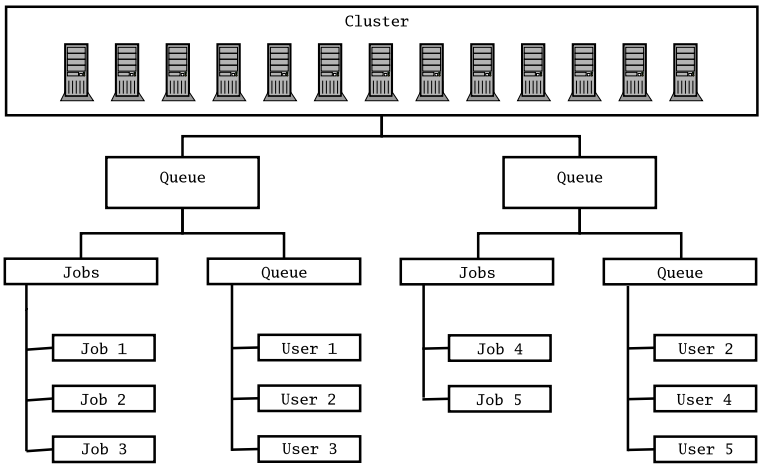


Figure 3.2: The organization of a LDAP MDS-subtree for a cluster, showing different queues of a cluster.

The information is gathered via LDAP queries, either through the NorduGrid Web Interface[2] or the user interface, this information is used to serve the necessary information to the brokering functions of the user interface. The requested information is generated locally on the resources, but it can optionally be cached for subsequent use.

NorduGrid has an indexing service, used to get the contact information for the resources in the grid. Even though, Globus provides higher levels of caching, this function is not used in the NorduGrid ARC, where the indexing service consists of simple dynamic link catalogs. The main function of these lists is to reduce the overall load on the information system. The resource information is ordered, in a topology, according to their national and geographical locations. This structure is depicted on Figure 3.3.

The last part of the information system is the soft state registration mechanism which is used by the local resources to register their contact information. Soft state is necessary because the amount of resources are not constant and thus no constant database of resources can exist. The resources must register themselves with the service as they appear on the grid, and they must subsequently keep registering themselves continually, otherwise the monitoring system purges the contact information.

[2]located at http://www.nordugrid.org

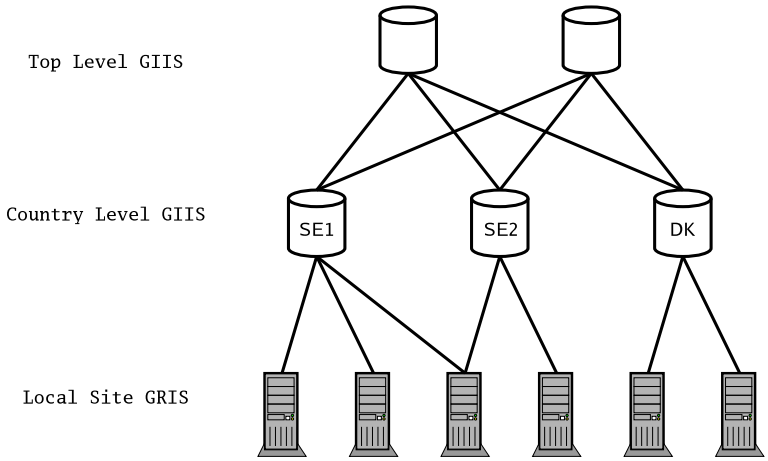


Figure 3.3: The structure of the GIIS topology in the NorduGrid ARC. The local information providers (GRIS) on the clusters registers with the country level indexing service (GIIS) which in turn registers the information with the top level indexing services.

3.3.4 Grid Manager

The Grid Manager is the gatekeeper on the local resource. It runs on the front-end of a cluster where it handles incoming job submissions and handles the interaction between the grid and the local resource management system. The Grid Manager is implemented as a layer above the Globus toolkit. It replaces GRAM delivered with Globus in order to provide additional functionality which were not supported by GRAM. This is primarily job and data pre- and post-staging functionality but also integrated support for Replica Catalogs and sharing of cached files among users.

The main responsibility of the grid manager is to process input and output data from jobs and submit the jobs to the local batch system. Figure 3.4 shows the interaction between the Grid Manager and the cluster software, which is described in the following. When a job is submitted to a cluster, a session directory is created. This directory holds all the files associated with the job. The grid manager checks the session directories, at a certain interval, to see if new jobs have been uploaded. If new jobs have arrived, the job description is parsed to see if any additional data is needed. It is the responsibility of the grid manager to gather all the data necessary for the job to be executed. This data can be uploaded by the user or downloaded from a storage element, by the grid manager. When the needed data is downloaded, it is placed in the session directory for the job. Optionally any downloaded data can be registered with a replica catalog.

When all input data is collected the grid manager submits the job to the LRMS running on the cluster[3]. Once submitted, the grid manager, periodically checks to see if the job has finished. When the job is finished, the grid manager collects the output data. The user is notified by email if this is stated in the job description, otherwise the job status can be monitored by using the user interface or the web interface. Furthermore data can automatically be uploaded to a storage element and registered with a replica catalog. When a job is submitted to a cluster it can be in one of several states [44].

[3]Currently the NorduGrid ARC only supports the batch systems Open PBS, Scalable PBS, and PBS Pro, but support for others are being planned.

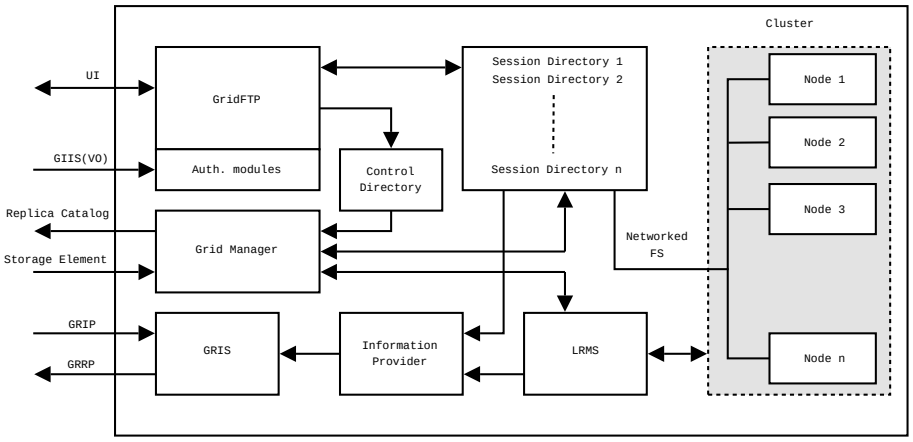


Figure 3.4: Closeup of the front end and nodes of a cluster, and how the middleware interacts. When the job description is uploaded by GridFTP, it is placed in the control directory. The grid manager reads the job description, an creates a session directory, and downloads any data needed. The job is then submitted to the LRMS and executed on the nodes, which have access to the session directory through a distributed file system. The information providers collects information, about the jobs, from the LRMS and session directories.

- **ACCEPTED** - The job has been submitted and accepted, but no processing have been done.
- **PREPARING** - The Grid Manager is collecting the data needed for the job to run.
- **SUBMITTING** - The job is being submitted to the local batch system.
- **InLRMS** - The job is submitted to the Local Resource Management System. When the job is in the LRMS it can be in a number of sub-states such as queued or running.
- **FINISHING** - The output data is being processed and optionally moved to a storage element or registered with a replica catalog.
- **FINISHED** - The job is finished and the user can download the data. The data are deleted after a certain amount of time.

The Grid Manager handles jobs by creating a separate directory where it stores the input files. There is no single point where all jobs in the grid has to pass and thus no single point of failure.

3.3.5 GridFTP Server

A job is submitted by uploading the job description to a cluster using GridFTP which is used for almost all data transfer within the NorduGrid ARC. GridFTP is a modified FTP server provided by the Globus Toolkit, however there is also a special NorduGrid implementation of the software that has been extended for better interoperability with NorduGrid.

The main differences compared to the Globus supplied GridFTP software is that it supports a virtual directory tree, that can be configured per user and that local access is implemented through plug-ins. These plug-ins come in two types: A local files system access plug-in and a job submission plug-in that has an interface for submission. Additionally there is support for GACL, a scratch area for temporary data accessible through GridFTP via a plug-in.

3.3.6 Computational Cluster

A cluster is the main computational element in the NorduGrid architecture, even though other computational resources can run on the grid, as log as they run a batch system[4]. A cluster consist of several machines, where one is the front-end dividing the work between the other computational nodes, often through a batch system. In NorduGrid all clusters run some flavor of Linux[5].

In order to add a cluster to the grid, the NorduGrid software has to be installed on the front end machine and have permission to interact with the local batch system. It is not necessary for the other nodes to run the NorduGrid software, but there must exist some form of distributed file system within the cluster in order for the nodes to get the job data. This goes a long way of making NorduGrid an add-on system and it respects local security and configuration policies.

3.3.7 Storage Element

A Storage Element is, as the name states, an element that stores data in the grid. A job description can specify that some data for the job should be fetched from a storage element. It is implemented as a GridFTP server; either as the GridFTP server delivered as part of The Globus Toolkit or as the GridFTP server supplied by NorduGrid.

At the moment, a smart storage element is being developed. This type of storage element is planned to support of data replication, automatic registration of incoming content, automatic up and download of data, and failure recovery [43].

3.3.8 Replica Catalog

The Replica Catalog is used for registering and locating data sources. NorduGrid uses the replica catalog supplied by the Globus Toolkit, but with minor changes to improve functionality. The information contained in the replica catalog is primarily used and maintained by the grid managers, but it can also be used by the user interface for the purpose of resource brokering. The replica catalog is based on OpenLDAP and is used without modifications, other than patching it to better cope with transferring of large files and adding the possibility to perform securely authenticated connections based on the Globus Security Infrastructure (GSI).

3.4 The Future of the NorduGrid ARC

On the NorduGrid website, is a list of things that the developers of the NorduGrid ARC would like to see implemented. Several of these tasks are already being developed

[4] There is actually a VCR on the grid, where jobs can be submitted to record TV-programs, which can the be downloaded.

[5] Support for other Unixes is underway.

while others are just suggestions. From this list it is obvious that there is still a lot of work to be done in order for the NorduGrid ARC to be "complete", even though, it is considered ready for production by the developers. It has been used with success in the first ATLAS data challenge, where it completed up to 15% of the total jobs completed, even though the participating NorduGrid sites only supplied 4.7% of the total CPU time in the second data challenge [49]. This demonstrates that NorduGrid works well, but DC1 also showed some problems with NorduGrid ARC. The main lesson was that a lot of time was spent babysitting jobs and resubmitting failed jobs to ensure that all the jobs completed correctly. This was done manually by the developers and physicists, giving to a lot of extra work. This lead to the conclusion that the NorduGrid ARC need a way of providing more automated control of the entire job submission and execution process to scale to the level needed by high energy physics. It also demonstrates the need for an automated production system for all the grids contributing to the ATLAS Data Challenge.

At the 6^{th} NorduGrid Workshop, a talk was given by Brian Vinter about the future usage of NorduGrid. A discussion of how to get users to the embrace the toolkit was initiated. At the moment there are initiatives to get applications other than HPC applications to run on the toolkit. These are applications in the areas of biology, chemistry, and visualizations. Even if the applications are developed to run on the grid, one of the challenges NorduGrid faces, is the problem of getting users to adapt the middleware. There are several reasons for this:

- The middleware is not necessarily compatible with the users existing systems.
- At the moment the users are not getting anything from using the NorduGrid middleware, which they do not already have. And why install some middleware which could have an impact on the stability of their production system. A lightweight front end is being developed for non Linux/PBS systems to accommodate this problem.
- There is no real HPC resources available on the grid run by NDGF. Even though there are two former top 500 machines on the grid, inspection reveals that only a limited number of CPUs are available to NorduGrid.
- It is only possible to run the grid manager as a root user, and each job runs as the same local user; having an impact on both security and secrecy and scares the security aware system administrator.
- Accounting, Accommodation, and Authentication, are stated as the three most important requirements in order to get new users to the toolkit. Accounting are still missing, so the grid is basically "paid" for by the resource owners who contribute it.

To get users interested in grid, the plan is to "gridify" some applications which could benefit from computational grids. Examples of applications being ported to the NorduGrid ARC are: Dalton, a chemistry tool creating huge jobs. The Povray raytracer as an example of something that is easy to explain to people not knowing anything about grid and HPC in general. Finally BLAST, a genome sequencing application, which has a lot of security requirements as it is dealing with datasets that is worth a lot of money and thus secrecy is a must. These examples are not only relevant in a NorduGrid context, but for most of the grid research today.

Chapter 4

Initial Considerations

We have already been looking at some of the reasons for developing a production system for the NorduGrid ARC. This chapter provides a deeper discussion of what an actual production system should provide. Concepts and ideas surrounding a production system are introduced by identifying the necessary features such a system should provide. This chapter also provides considerations regarding the design and implementation. We start by looking at the general aspects of the system we are developing.

The main function of the system proposed and developed in the project, is to provide tools to enhance the management of jobs being executed on the grid, thus we choose the name Job Manager for the system.

The idea of creating a job manager is not new. In the introduction three projects that have some parallels of the system we are proposing, was mentioned. We will look closer on these projects to get inspiration and to steer clear of common pitfalls.

The PROGRESS project is a Polish grid project aiming at creating a complete grid solution. It is build on Globus and the grid engine from SUN [8]. The method for interacting with the grid is through portals. These portal communicate with the grid through a Grid Service Provider, which mediates between the portal and the grid [9]. It does not provide the autonomous job control functions that the Job Manager does. The common goal between PROGRESS and this project is the goal of creating more flexible user interfaces for the grid by introducing an extra layer to into the grid architecture.

Nimrod is a tool for parametric modeling[1]. Nimrod/G is a grid enabled version of Nimrod, and it addresses issues related with running Nimrod in grid environments, e.g., coping with a dynamic resource pool and access to a multitude of grid environments (Globus, Legion, and Condor). Nimrod/G is a layer in top of the grid middleware and it is similar to our Job Manager in some aspects. They both sit on top of the grid middleware, where it manages the access, dispatch, and execution of jobs on the underlying grid toolkits. However Nimrod/G is limited to parametric modeling, and therefore not as flexible as the Job Manager. Nimrod runs on the workstation where pre- and post-processing of the jobs are handled. Nimrod/G also works as a resource broker and supports different schedulers and accounting [1].

Lastly we will look at the EDG. The EDG middleware had had a central resource broker, in order to handle scheduling better. Furthermore it was hiding the underlying protocols, providing a well defined interface to program against. The problem was that the central resource broker was a point where all jobs and data should pass through

[1] An application which is run several times, but with different input parameters.

when being submitted. This was a bottleneck in the submission process and did never fully work[2] and should be avoided in the Job Manager.

The difference is that the above mentioned applications all are made with a specific purpose in mind. Either for a specific application or it is not a job manager in the sense we use it in this project.

4.1 The Need for a Job Manager

In the introduction, Chapter 1, some of the reasons for developing a more advanced application interface to the grid, where outlined. This section takes a deeper look into the most important reasons for creating a Job Manager which provides such an interface.

The current user interface in the NorduGrid ARC is a suite of command line tools called from a standard Unix shell. These commands are described in Appendix B. This interface along with the grid monitor on the web is used to interface with the grid. However this interface is difficult to use internally in the applications and applications that are intended to use the grid has to do so by calling the command line interface. This approach is not suitable for application use and makes it hard for applications to use internally. An example of this is; when submitting a job using *ngsub* an error code, indicating whether the job submission was successful or not, is returned. If the application wants to know the job id of the submitted job, it will have to parse the output of *ngsub*, which is rather inelegant. Some applications using this technique have been developed for NorduGrid ARC. One of these is a frontend for the well known raytracer Povray [3] but it primarily used to demonstrate the possibilities of running different types of software on the NorduGrid ARC.

The existing user interface fills the need for a simple interface that can be used to quickly submit jobs from the command line, but it became apparent during the first Atlas Data Challenge that it lacks the capabilities needed of a large scale production system. One of the things missing is the ability to monitor changes in the job. Fundamentally there is a need to monitor the entities in the grid and react to changes when they occur.

There are several reasons for creating the Job Manager. One of them is, as just described, to extend the functionality of the interaction with the grid for users and applications. To reduce the complexity of grid usage there is also a need to automate trivial tasks and provide functionality to assist the user or application with complicated tasks. By providing these functions the usage and development becomes simpler and should facilitate a faster adoption of grid technology. There are several trivial tasks concerning job preparation that could be automated in order to ease the usage of the grid for the user/application.

Another reason is that of high level control over the execution of jobs on the grid. Examples of this, is resubmission or movement of jobs, support for different schedulers, and data management. This said, the problem is not only to have easy access to the grid. The problems must also be suitable to a grid solution, i.e., it must be able to parallelize them and they must be solvable without to much user interaction.

The Job Manager is being designed as a separate application, opposed to a software development kit with libraries to link against. The main reason for this choice is

[2] This claim has been discussed on the NorduGrid discussion mailing list and there seem to be consensus about this (http://mail.nordugrid.org/mailman/private/nordugrid-discuss/2004q1/012396.html).

[3] Homepage at http://www.povray.org/

flexibility. There are of course advantages of both approaches, and it may be that the functionality of the Job Manager should also be provided as a library for developers to develop against. Software development kits have the disadvantage of being bound to one particular language in contrast to a client-server model with a specified protocol can be used by several languages[4].

The flexibility of having the Job Manager as a separate application manifests itself in different areas. From an application programmers perspective it is difficult to implement grid functionality into the applications since no software development kit for NorduGrid ARC exists. Development of a SDK does not solve all the problems since many applications would have to implement the same functions for job control leading to duplicate program logic across applications. Another important aspect of the separation of the Job Manager from the application, is that it makes it easier to adapt to changes in the underlying grid software without having to change the application. This is important as development for grid is somewhat a moving target although NorduGrid is more stable than most. The fact that grid development is a moving target is obvious from the number of changes in names and technology during the latest years, but this will probably change in the future as the grid technologies matures and gets adopted by a larger base of users.

From a user perspective embedding the job control in the application leads to other problems. In order to do resubmission and other tasks, it would require that the workstation running the application should be kept turned on, and have the application running, as long as jobs are running. In many cases this may not be desirable. The separation of job management into a separate application, would make it possible to access the jobs from different workstations and in different ways, e.g., through an application or a web portal. It also makes it possible to start managed jobs from foreign computers, e.g., from a web cafe or through a portal and have them managed by a Job Manager running remotely.

4.2 Design Philosophy

When creating NG Proxy one of our goals was to change as little as possible in the NorduGrid code, and only to extend the functionality. The Job Manager is not just an extension, although backward compatibility is retained, but is a new interface for applications to use.

In the development of the Job Manager we intend to create a complete design. The focus of the implementation should be on the Job Manager and not to change major components of the NorduGrid ARC. This is to help adaptation of the Job Manager in the NorduGrid ARC, for previewing and testing purposes. The reason is that it will make it harder to get acceptance of the Job Manager if it can only be used on sites that run a modified version of the NorduGrid ARC. This is only a preliminary requirement and the necessary changes should be made to the NorduGrid ARC if the Job Manager proves to be a success. Keeping this focus will hopefully yield a general Job Manager, which may be less difficult to adapt to changes in the NorduGrid ARC in the future. This is important since the NorduGrid ARC is in a state of active development. To further accommodate this, the design of the Job Manager should be as modular and extensible as possible, limiting the assumptions and restrictions and dependencies on other components in the middleware.

[4] In theory this has the same implications as with an SDK, but by leveraging an widely adopted protocol we do not have to implement it in several languages.

With this in mind, we start by considering the features the Job Manager should have and in the next section, the necessary features, components, their design, and intended functionality, are discussed.

4.3 Features of the Job Manager

To design the Job Manager we need to determine what features the Job Manager should provide. One way to go about this is to examine the current user interface to see what functionality it provides. It is important for the adaption of the Job Manager, that it is able to perform all of the tasks the existing user interface is capable of, along with the added functionality.

The existing user interface, i.e., the ng family, along with GridFTP is currently the only way which the user currently can interact with the grid[5]. The functionality of the existing user interface is listed in Appendix B. These functions can roughly be divided into three categories: Job control, information querying, and data management. The Job Manager must provide a set of capabilities, in which these categories are represented. Those sets are presented below:

- **Job control –** The functions in this section deals with submission of a job, it must be possible to specify cluster and queue. Furthermore it must be possible to cancel and clean jobs.
- **Information querying and retrieval –** This has more to do with the grid as a whole, i.e., new and departing clusters, storage elements, replica catalogs. Cluster specific information, i.e., cluster load, queue length and information about a specific job, e.g., status.
- **Data management –** The transfer of data to and from local machine and between storage elements but also registering files at replica catalogs and deleting files.

The list is used as the basis for identifying the basic functionality of the Job Manager. However the Job Manager will provide more functionality than is provided by the existing user interface tools. This functionality should be used as a basis for the protocol designed to support applications use.

4.3.1 Application Interface

We have already discussed the implications of choosing different types of interfaces between an application and the Job Manager. The Job Manager separates the functionality of the user interface and the grid, by creating a protocol between the two. This makes it possible to have several frontends using the same Job Manager and functionality without having to implement it themselves. In turn this should make it easier to develop several frontends, e.g., a portal, or an application.

An example of a better application interface for *ngsub* would be to return the job id, or raise an exception if an error occurs during the job submission. However not all languages supports exceptions, and languages each have their quirks and way of doing things. Making an API that is consistent between all languages is therefore not a very feasible option. Preferable a protocol between the Job Manager and application is a better solution and it allows the Job Manager and application to be at different

[5]There is also the grid monitor, but it delivers only information and does provide any means of interaction

machines. What is needed for the application protocol, and what technologies should it be used is discussed later in this report, see Section 6.3.

4.3.2 Extended Features

Some of the features wanted for NG Proxy, e.g., the ability to automatically fetch output files from a finished a job to the machine on which the Job Manager is running on, does not have any direct relevance for an automatic production system. However it is a desirable feature for a scientist to have on his or her workstation. One can imagine other domain specific extensions could be desirable as well, from a users point of view.

Extended functionality is possible since the Job Manager has the ability to continuously monitor the grid and most importantly, react to changes in it. The ability to autonomously react to changes, e.g., the status of a job, is what enables the Job Manager to automatize tasks for the user. This is not limited to resubmitting jobs if they fail or to automatically retrieve output data upon job completion. Another example of functionality is to move a job, if a new and better suited cluster appear.

Having the ability to continuously monitor the grid allows the Job Manager to react statefully because the Job Manager has knowledge about about the job which is not known by the existing interface. An example is the ability to keep track of jobs through a series of submissions and resubmissions.

As we have already discussed other areas than high energy physics applications may benefit from the grid. One of the reasons that high energy physics is one of the areas where grid has been adopted is because it is an area that are used to using HPC resources and the shift to grids are not that a big step. In order to attract new users to grid - and in this case, NorduGrid - the complexity of writing applications and running jobs on the grid must be reduced. The difference in user needs and application types, gives rise to new issues. Since grid is a complex topic it is not possible to predict every users need. In order to prevent imposing limitations on the user, the Job Manager should be extensible, by providing some form of plug-in structure that allows users to customize the behavior of the Job Manager if this is needed. An important aspect is that users can use not only the existing plug-ins in the Job Manager, but is able to write their own, and extend the Job Manager. Examples of plug-ins are job resubmission in the case of failure or fetching output files from finished jobs.

4.3.3 Failure Handling

In order to prevent the Job Manager becoming a single point of failure, measures to prevent a crashed Job Manager leading to failure must be taken. These measures could range from a cron job monitoring the Job Manager and restart it if it fails, to making a distributed system of Job Managers managing the jobs, thereby making it possible is to have Several Job Managers working together managing jobs. This feature is mostly geared toward production systems where large portions of jobs are handled, and failure of a Job Manager would result in jobs not being submitted or not being monitored. Making it possible for Job Managers to take over from each other can greatly reduce human intervention in the case of a crash.

The first solution is a bad idea as it does not help if the host on which the Job Manager is running on crashes. The last solution of distributing the Job Manager provides a much more fault tolerant system, and is discussed in detail in Chapter 8.

4.3.4 Multi User Job Manager

It should be possible for the Job Manager to support multiple users. This makes it possible to run a Job Manager, managing the jobs for several users. Having the Job Manager to run on behalf of several users would be a nice addition, since it could dramatically reduce the number of Job Managers running, because users can share them.

This features is however hard to obtain for two reasons. The first is that it would become hard for users to write their own plug-ins and extending the Job Manager. This would mean that they could possibly have access to information about other peoples job and could also jeopardize the stability of the Job Manager by using their own plug-ins, making it harder for people to expect that the Job Manager would just work.

The other reason is that the Globus and NorduGrid Toolkit is not geared toward shifting users at runtime. Usually the toolkits look for the file `/tmp/x509up_uUID`, where `UID` is the users Unix user id [42] as the proxy certificate. Alternatively the user can specify the location by setting the X509_USER_PROXY environment variable. However when the user proxy certificate has first been selected, it is being used implicitly, and there are no direct way of changing it. Even if it could be changed at runtime it would only be possible to submit jobs from one user at a time, since submitting jobs on behalf of another user would require a change of proxy certificates. Changing the toolkits to allow changing user proxy certificates or being able to handle several of them simultaneously would require a significant amount of changes to Globus. Due to the these two obstacles, we have chosen not to make Job Manager support multiple users per instance, but if the dependency upon Globus would be removed, the possibility of a multiuser Job Manager should be investigated further.

4.3.5 Feature List

The examination of the current user interface, along with the other features and needs previously discussed, have lead to the following list of features.

- **Provide interface to applications –** Deliver a clean and easy to use interface to application developers.
- **Provide the same possibilities as the old user interface –** This is important as the functions are necessary when using the NorduGrid ARC. Furthermore it is also important if we want users to start using the Job Manager.
- **Automatize tasks for the user/application –** To automate tasks that do not need user interaction, like fetch data, prepare jobs.
- **Extend the Job Manager –** Making it possible to extend the functionality using plug-ins.
- **Provide high availability –** Implement mechanisms to prevent the Job Manager to become a single point of failure.
- **Multiuser –** make it possible for several users to use the same Job Manager to monitor jobs without giving access to other users jobs on the manager.

This list is the basis for the design of the Job Manager. Another question is whether to continue the development of NG Proxy, or to start over. It is possible that NG Proxy

could be extended, but integrating all these features into it would quickly turn it into a giant mess because NG Proxy was never designed with this purpose in mind. As a consequence it is better to start over and rethink the whole design. Needless to say, things which are working in NG Proxy should be reused when possible, but the focus should be on a complete design and not the extension of NG Proxy to support the new features.

4.4 Language Choice

After having implemented NG Proxy in C++ [69], our experience was that a lot of development time had gone into investigating bugs and dealing with interesting perks of the language. While C++ is certainly a powerful language, we found development in it rather cumbersome and unnecessarily complex for the task. Therefore it was decided to implement the Job Manager in a high level language. By implementing the Job Manager in a high level language we expect to reduce the development time and making it easier to modify and for other to understand.

When implementing in high level languages speed is often a concern[6]. The data and data structures on which the Job Manager should work on are expected to be relative simple, and the performance requirements on the Job Manager are relatively small. Furthermore the Job Manager will primarily be I/O bound, i.e., waiting for network and not be CPU bound. This means that the Job Manager will spend most of its time waiting for events to happen, since none of its tasks include anything CPU intensive. As performance is not an issue we have the luxury of being able to decide the implementation language freely. The only constraint is that is must be possible to use C and C++ code; either directly or through bindings, otherwise we would not be able to use the existing NorduGrid ARC code base, meaning that we would have re-implement a lot of functionality, which should not be necessary.

For our implementation language we decided to use Python [32] to create the Job Manager. Python is a interpreted high-level object oriented language, supporting multiple paradigms. The language is known for combining remarkable power with very clear syntax [34]. It it possible for Python to interface with C and C++, by creating wrappers, as will be explained in section 5.2.3. Code written Python usually runs on all the platforms on which the Python interpreter runs, making it highly portable. Portability is a desirable feature in a grid environment, since it is easier to support and create heterogeneous grids. Finally development in Python is usually magnitudes faster than developing in low level languages such as C and C++.

4.5 Other Considerations

There may also be problems introducing a new layer, as it adds to the overall complexity, but by making it optional, this can somewhat be circumvented. It follows the modularity of the rest of the grid. Another concern could be the impact that the Job Manager have on the rest of the NorduGrid ARC components in terms of added communication and queries. However, this should not be a problem if we rely on the information system for the distribution of information. But it demonstrates that the communication introduced by the Job Manager should be kept on a minimum and the possibility of caching should be explored.

[6] Although this concern is usually more cultural than technical.

4.6 Summary

This chapter has presented the considerations made before the construction of the Job Manager. Figure 4.1 illustrates how we envision the Job Manager would work in a grid, giving the reader a feel for the Job Manager in the big picture, before presenting an overview of the Job Manager in the next chapter.

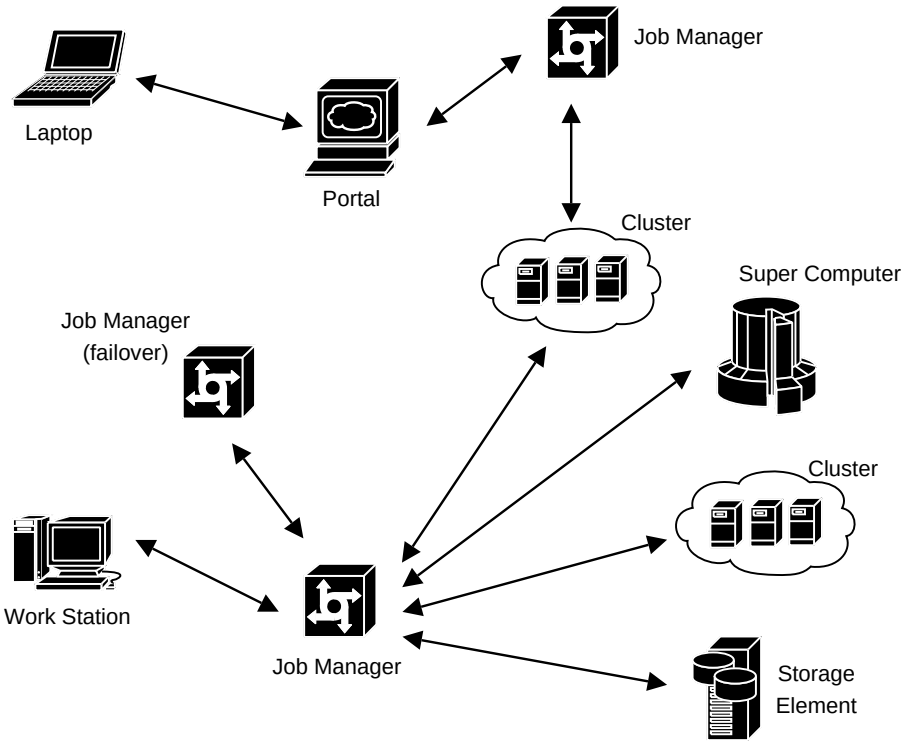


Figure 4.1: How the Job Manager interacts with the grid, acting as middle layer between application and the grid. Here a Job Manager acts a backend for a portal, one as a production system for a work system, with a fail over Job Manager.

Chapter 5

The Job Manager

The chapter describes the Job Manager. It starts by giving an introduction to the Job Manager, outlining the concept of it. Hereafter an overview of the construction and which modules it contains is given. After this the dependencies and modules will be briefly described; giving the reader a feel for the workings of the Job Manager.

As described in Chapter 1 the Job Manager introduces an additional layer between the grid middleware and applications using the grid. This is illustrated in Figure 5.1. The purpose of this layer is to hide the complexity of using the grid by providing the application with a clean API to facilitate the use of the grid, while also removing grid code from the applications.

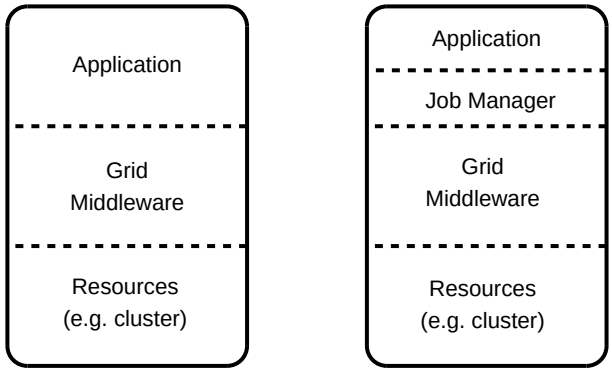


Figure 5.1: The Job Manager introduces an extra layer between the grid middleware and application, hiding complexity of grid usage from the application.

Besides providing easier access to the grid the Job Manager can also aid the application by assisting it with various tasks, such as building jobs and automatically retrieving output data from finished jobs.

The Job Manager can also simplify the use of the grid and aiding the application in solving tasks on the grid. It also features handlers, which are plug-ins that allows the user to change the functionality of the Job Manager. Finally it has the ability to communicate with other Job Managers, making it able to handle fail over, in the case of a crashed Job Manager. This feature is desirable to have in production systems where a crash would often result in manual reconstructing of a list containing which jobs that had finished, failed, or still running.

5.1 Job Manager Overview

On Figure 5.2 an overview of the Job Manager is depicted. The functionality has been divided into logical modules illustrated as boxes within the Job Manager. Each of these modules covers a specific aspect of the Job Manager, and will be discussed in the following chapters. To make the figure less complicated no effort has been made to illustrate dependencies or information flow within the modules. The figure merely serves as a reference point for the further discussion of the modules and their function.

Starting from the bottom, the Job Manager has several dependencies, for which it relies for its functionality. The two major dependencies are the NorduGrid ARC [10] and Globus 2 [59] toolkits. Since the NorduGrid ARC is dependent on the Globus Toolkit, it lays on top of Globus. The Globus and NorduGrid ARC are written in C [41] and C++ respectively, whereas the Job Manager is written Python. As Python does not interface directly with these languages, wrappers has to be created. Fortunately we only needed wrappers to the NorduGrid ARC, since the Job Manager does not access the Globus library directly. The creation of these wrappers is described in Section 5.2.3, later in this chapter, and in Appendix C. Also the Job Manager depends on M2Crypto [66] for its RPC server. Moving up the Job Manager it consists of several modules, each representing a functionality aspect. In the following, each module will be briefly described, starting with the information system.

The information system module deals with querying the grid for information, usually getting cluster lists from the top GIIS servers, or getting information from clusters. Additionally it is able to cache the information it retrieves for a certain time interval. Data management handles data related tasks such as the movement of data to and from storage elements, and registration data with replica catalogs. The RPC server is the module which talks to applications. It does authentication and authorization of incoming requests, and translates messages into function calls. Submission, cancellation and other job manipulation is done from the job management module. This module also deals with scheduling of jobs and is able to help the application build jobs. The JM communication module handles the communication with other Job Managers, i.e., it coordinates the replication, and handover of jobs, so if a Job Manager fails the jobs will continue being monitored by another Job Manager. Even though a Job Manager is meant to be run continuously, it must also be able to save its configuration and session data; the last being list of jobs being monitored. This is necessary if the Job Manager must be shutdown and started again. This saving and restoring is done by the configuration and session management module. The Job Manager also features a logging capability, whereto it logs its events, decisions, and so. Finally the handler module deals with configuration and plugging in any handlers which the user plugs in the Job Manager.

Some of the modules are self contained, while other rely on other modules to complete their functionality. The RPC server, data management and informations system are self contained, i.e, they do not rely on any other modules. The job control module is dependent on the information system to find the resources it need, and furthermore needs the data management module to move data. To report to other Job Managers the JM communication needs to gather information from the job management module, and it needs the data management module to transfer data. To save the configuration and session, the configuration and session management module is dependent on all the other modules, even though it does not rely on them for its functionality. The logging module is independent of any other modules, but requires the Configuration module to be started. Finally the handler module is dependent on the configuration and session

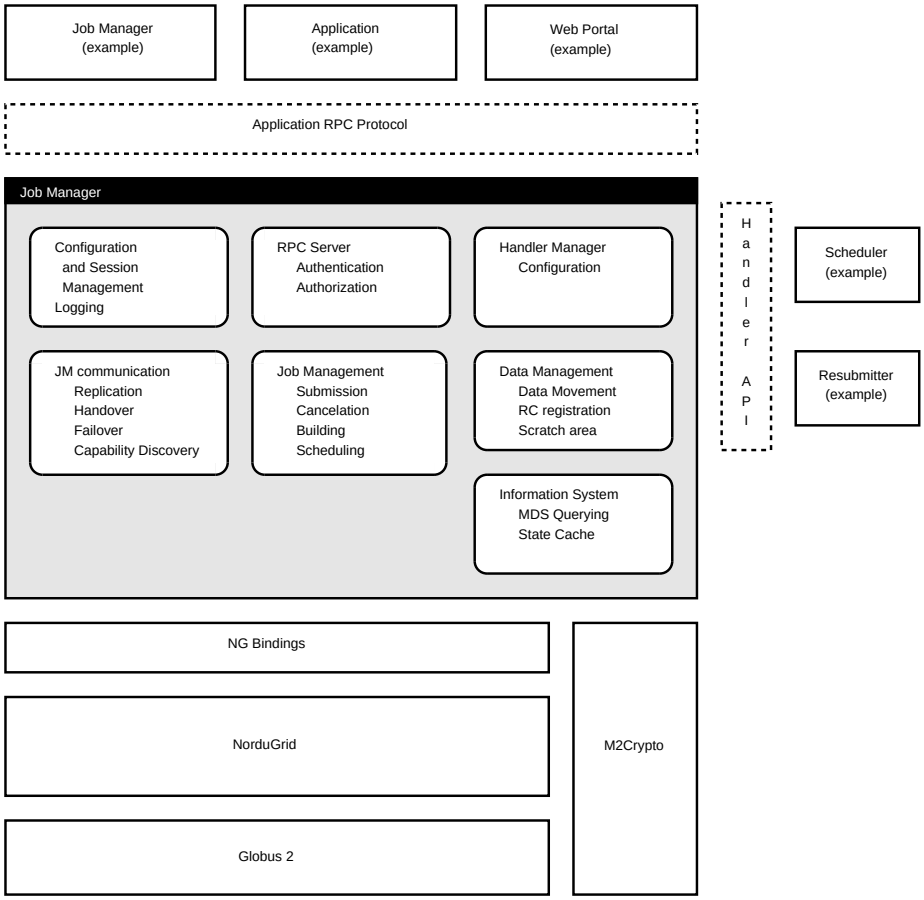


Figure 5.2: Overview of the Job Manager showing the different modules it consists of, the placement of applications and handlers, and external dependencies.

management, but not directly dependent on any other modules. They will however use other modules since they are plug-ins and need access to some of the Job Manager internal functions to perform work.

5.2 External Job Manager Dependencies

The Job Manager depends on the functionality on other software packages. On Figure 5.2 the dependencies of the Job Manager is displayed below it. The most notable dependency is the NorduGrid ARC, from which the Job Manager uses much functionality for its basic operations. The NorduGrid ARC again depends on the Globus Toolkit for much of its functionality.

This next section describes the external dependencies of the Job Manager, starting with M2Crypto. Hereafter the dependency of the NorduGrid ARC is described, and finally the wrappers for this.

5.2.1 M2Crypto

The Job Manager needs a way to authenticate its clients, i.e., establish their identity. In NorduGrid ARC this is done using using X.509 certificates [27]. To make the Job Manager easy to use, it should also be capable of doing authentication using these certificates. Fortunately X.509 certificates is a widely used standard, and is supported by the SSL protocol [36]. The OpenSSL [61] library is an implementation of the SSL protocol, supports X.509 certificates and comes with most Linux distributions. Therefore it was a natural choice to use for authentication protocol. Unfortunately Python did not have direct support OpenSSL, and therefore no support for using X.509 certificates for authentication. Fortunately the M2Crypto project delivers these [66]. M2Crypto delivers an object oriented interface to most of the OpenSSL API, including the ability to do authentication using X.509 certificates. Therefore M2Crypto is a dependency on the Job Manager, as it is needed for our authentication scheme. The use of M2Crypto is covered in Section 6.3.

5.2.2 The NorduGrid ARC

The Job Manager uses much of the functionality from the existing the user interface in the NorduGrid ARC. As mentioned in Section 4.3, the user interface functionality can roughly be divided into three parts: Job control, information retrieval and data management. The Job Manager uses functionality from all these parts, to build its own enhanced functionality, which it provides to applications.

5.2.3 Wrappers

Since the user interface in NorduGrid is written in C++, wrappers must be created for interfacing with it from Python. Wrappers for the ng commands already exist, and are used by the NorduGrid Executor[1]. However since the Job Manager provides an API which surpasses the ng commands, access to the internal API in the NorduGrid ARC is needed. Therefore it is necessary to create bindings for functionality needed in NorduGrid. Writing such wrappers by hand is relatively easy, but tedious for a large

[1] No reference for this exists, it is part of the production system for the Atlas Data Challenges

amount of code, as much of it is of a repetitive nature and consists mostly of type and error checking.

Hence we decided to auto generate the wrappers needed for the Job Manager. For this we choose the Simplified Wrapper and Interface Generator (SWIG) [25]. The reason for using SWIG is that it is able to generate wrappers for functions and classes for C++ classes, making it possible to instantiate objects from C++ classes in Python. Furthermore it is possible define templates so that vectors and other C/C++ data structures can be accesses from within Python. This gives some inelegant interfaces around calls to the NorduGrid ARC functionality, but wrappers to better interfaces are easily created in Python. For an in depth explanations of how to create SWIG bindings see Appendix C.

Having given an overview of the Job Manager and its external dependencies, the next chapter will describe the small modules in the Job Manager.

Chapter 6

Job Manager Modules

This chapter describes the smaller modules in the Job Manager, i.e., the modules which are not big or complex enough to justify giving them a separate chapter. These modules also form the the basis of much the functionality in the Job Manager. The modules that will be described in the chapter are, in order: Configuration and session management, Logging, RPC server, Information system, and Data management.

6.1 Configuration and Session Management

This module handles the loading and saving of configuration and session during startup and shutdown of the Job Manager. Configuration management deals with saving and loading configuration options, e.g., which port to use for the listener. Session management handles saving and restoring of job information, so the Job Manager can continue monitoring jobs after having been shut down and started.

While configuration and session management are somewhat orthogonal topics, as they handle the settings of the Job Manager and its internal state. they make it possible for the Job Manager to regain it previous state when restarted. Configuration change is not a common event, so it can be saved each time a change is made. For session management saving the session each time a change occurs is a more daunting task, since the status of jobs can change quite often. It is not necessary to save the session every time a status change occur, since this information can be queried from the information system. The only time when it is necessary to save the session is when a client submits a new job to the Job Manager, since this information cannot be found elsewhere. In addition the session should be saved at a regular interval so changes to the meta data of a job is not lost.

The next two sections describes how the configuration and session works, presents their API, and how they are used.

6.1.1 Configuration

The configuration module, handles the user defined settings for the Job Manager and is similar to configuration handling in other applications. The main purpose is to read and write changes to the configuration and settings of the Job Manager. The user must be able to change the settings by changing the configuration file, as well as change some settings through the application interface.

The configuration consists of several sections each specifying settings for various areas of the Job Manager. The configuration file has a series of sections which holds the configuration for the different areas of the Job Manager. The settings which it should be able to change are the following.

- **User settings** – Username and credentials (password or certificate) and if it should be possible to poll the Job Manager for information anonymously (useful for portals).
- **Connection settings** – The port on which the Job Manager should listen.
- **Job Manager Group** – Group which the manager belongs to. Needed to find failover managers
- **Logging and paths** – Location of log file and log level. Paths to session directory.
- **Handlers** – Which handlers should be enabled along with handler specific configuration. The configuration should contain sections that holds individual settings for the handlers.
- **Miscellaneous** – Settings that does not fit anywhere else, e.g., what should be done with jobs on shutdown.
- **Internal settings** – Settings that the user may not need to change, but may be necessary to change for debugging purposes or for future adaption to changes. Examples are: Local timeouts and bind attempts, what method to use when encountering duplicate jobs.

Apart from reading and writing the configuration there should be methods for updating the configuration system wide. This is used when the Job Manager starts and all the modules should be configured, but it is also necessary if the user needs to have the configuration reloaded without stopping the Job Manager.

The module itself is based on the Python module `ConfigParser`[1] which provides basic configuration handling. There are other modules that provide this, but ConfigParser is part of the Python standard library and should be present on most installations. The module is implemented by extending the ConfigParser to handle Job Manager specific functions.

When the Job Manager starts, the first module to be activated is the configuration module. When initializing it tries to locate the configuration file in the following order. First it checks if the location has been explicitly specified on the command line. If this is not the case it reads the environment variables for the variable `$JM_HOME` for the location of the configuration file, if this variable is not set then `$HOME` is consulted and the default location `$HOME/.jm`. If no configuration file can be found in the locations above, a default configuration file is created and the user warned. If none of the locations or environment variables above is found, the Job Manager fails, and the user must correct the situation in order to start it again.

When the configuration file is located it is possible to set up and start the logger, as the location of the log file can read from the configuration file, or it can be placed in the default location. In order for the configuration module to function properly, it is necessary to add some additional functions to the ConfigParser interface.

[1] http://docs.python.org/lib/module-ConfigParser.html

- `__init__(filename = None)`
  Initializes the configuration module and tries to read the configuration from the configuration file and sets up logging.

- `reload_config()`
  This function is called if the Job Manager receives a call to reload its configuration and restart. This is typical behavior in Unix if the signal `SIGHUP` is received.

- `set_defaults()`
  When called this function sets some reasonable defaults for the configuration. It is used for generating a new configuration file if it does not exist.

A part from these methods, the standard methods and data structures from `ConfigParser` is used for the other modules to read the configuration from the respective sections. If the Job Manager receives a signal to reread the configuration, the main thread must make sure that `reload_config` is called, and that all modules call the configuration module and reads the new configuration settings.

6.1.2 Session

The session module keeps track of the state of the Job Manager. It is the responsibility of the session module to return the Job Manager to a consistent state after a shutdown or a crash. If it is not possible to return to a consistent state, the manager should be started as a new manager, forgetting about its id and previously managed jobs, in order to prevent jobs from becoming orphans, see the discussion in Section 8.5.

In order to restore a session, there are some data that must be saved to nonvolatile storage. This includes information about: The id of the Job Manager, the jobs currently being managed, and meta data associated with the jobs. The rest of the information is static and could be taken from the configuration file. This can lead to problems in the case where the user edits the configuration before restarting the Job Manager. But only a few options can have and impact on the session, and therefore it is the responsibility of the user to exhibit caution when editing the configuration between executions. For this reason it should be possible for the user to start the Job Manager without restoring the previous setting.

In order to prevent jobs from disappearing in case of a failure, job information must be written to disk before the information is propagated to the failover managers. All the information should be written to a session directory.

The session handler is implemented as a simple module capable of storing the job data structure, along with the data structures of the Job Manager communication module. It is called when a new job is submitted to the Job Manager but before it is registered at the failover managers. The interface to the Session module is rather simple, as it has only two functions:

- `update_session()`
  Updates and saves the session to the session directory.

- `restore_session()`
  Restores the previously saved session.

6.2 Logger

Since the Job Manager is a daemon, and runs continuously, is not feasible to have a human monitor it. Especially if used as a production system, the quantity of jobs may be large, making it infeasible to supervise it. However at certain times it may be necessary to inspect the events that has happened and the decisions made, i.e., history of these must be kept. Usually this is done by writing this history to a file, which can be inspected if necessary. Therefore the Job Manager must provide a logging function.

The Job Manager features a logging mechanism, whereto it logs information. Examples of this information could be events such as a job submission, incoming RPC calls or any errors that could occur. The module uses the logging module that comes with Python [33], from which all the functionality in the module, except the setup, comes from. The API to use the logging feature in the Job Manager is:

- `SetupLogger(logger_file)` This functions sets up the logger. The only argument is the location of which file to log to, which must be passed along. If not set, an exception will be raised.

- `Logger()` Returns a logging object, which can be used for logging events. If the logger has not yet been setup an exception will be raised.

The logging object returned by `Logger()` has several methods which can be used for logging. Each of these methods represents a category which the logged information should adhere to. Furthermore each of these methods also has a logging level and when creating the logger, a desired logging level should be set. Setting this will cause any logging information below that level to be omitted when logging. The logging module in the Job Manager outputs all logging information. The logging methods all take an arbitrary numbers of strings and are as follows:

- `debug()` Used to emit debug information, such as contents of variables and entering certain parts of the program, making it possible to use this information to debug the program with. The debug method has the lowest logging level, i.e., the one that will be filtered away first.

- `exception()` The exception method is used for logging exceptions, and should only be used within exception handlers. Normally only unhandled exceptions should be logged since exceptions are a normal part of the program flow in Python. The method has the same logging level as debug.

- `info()` Info is used for normal events, such as a successfully job submission or an incoming connection from a client, i.e, information which represent the intended behavior of the program. The logging level for info is above debug and exception.

- `warning()` Warnings should be issued when an unintended behavior arises, such as the inability to contact a cluster, but was handled gracefully and the program can continue working. Warnings are the level above info.

- `error()` The error logging level should be used for errors which cannot be handled gracefully, such as assertion errors or unhandled exceptions. The error level follows the warning level.

- `critical()` Critical logging events should be used when a fatal condition arises within the program, that disallows it to continue doing some or all of it work. A critical logging event would often be followed by an action such as restarting or quitting part, of, or the whole program.

The above descriptions of how to use the logging levels are subjective to the developer, e.g., no clear limit exists between issuing warning and error messages, but consistency should be enforced by logging the same types of events to the same log level in the entire Job Manager.

Having explained the logging module, the RPC server, i.e., the module which accepts incoming remote procedure calls to the Job Manager will be explained.

6.3 RPC Server

The Job Manager must feature a way to communicate with applications. Since we cannot assume that the Job Manager and application are on the same machine, the communication must be able to work on a network, and in a secure manner. Secure is a rather vague definition; in this context is means that the communication must be confidential, integrity must be assured and it must support authentication and authorization. Confidentiality is that the message is private, i.e., only the receiver can decrypt the contents. Integrity, means that no one can tamper with the message. Authentication is to establish the identity of the peer and authorization is deciding if the peer should have access. Furthermore is important the user can authenticate to the Job Manager as other resources in the also identify themselves. The standard way to establish the identity in NorduGrid is to create a X.509 proxy certificate by using a public/private key pair, where the public key has been signed by a NorduGrid certificate authority. This means that X.509 certificates should be used for authentication. The protocol used for communication must be standardized, since creating a new protocol, for which no libraries exists will not help move code out the application. Furthermore the protocol should support a relatively high level of abstraction; after all, grid use is supposed to become easier. A common concept for high level protocols is remote procedure calls (RPC), i.e., communication happens transparently by making a simple procedure call. This abstraction has proved to be a good way to do high level communication, therefore a RPC protocol should be used.

To summarize the requirement for the server and the protocol used in the Job Manager:

- It must be secure, i.e., confidentiality, integrity, authenticated and authorization.
- Authentication must be based on the X.509 certificate.
- It should use a standard RPC protocol with a high level of abstraction.

Given the two first requirements it is almost apparent that SSL [36] or something build on top of it, will have to be used. This is since the SSL is regarded as the standard for doing secure communication on the Internet and supports authentication using X.509 certificates. Also it provides a standard BSD socket interface [7], meaning that it does not dictate the protocol and basically any protocol using TCP can be based upon it. This gives us freedom to chose the protocol that we want, without having to worry about security aspects, since these concerns can be separated.

6.3.1 RPC-Protocols

For RPC we considered several options. The two notable was XML-RPC [75] and SOAP [14]. XML-RPC is a simple and lightweight way of doing RPC. As its name suggest it uses XML for encoding its data. Furthermore it embeds its request and replies within a HTTP header making it possible to use it through web proxies. What shines about XML-RPC is that it is really simple. Its specification [75] is seven pages long; it supports only six basic data types along with structures and arrays. In [63], a comparison of XML-RPC and SOAP it is said that:

> *Any competent programmer*[2] *should find no difficulty whatsoever in implementing XML-RPC in their software after reading its spec.*

SOAP reminds a lot of XML-RPC, since it also based on XML and is capable of doing RPC, however it is really a protocol to transfer objects. The latest SOAP specification [12], is about 40 pages, and presents a rather complex object system. SOAP makes it possible for users to define their own types, and includes esoteric features such as sparse and partial arrays. SOAP is transport independent, meaning that it does not dictate how the underlaying system should work. This means that a SOAP object can be transferred over almost anything, e.g., over SMS transfer or embedded into HTTP. Finally SOAP is a part of web services, which has gained some momentum within grid software lately (see Section 2.6.3 and 2.6.4). We will not debate on whether web services it the thing or not for grids, since this out of scope for this report.

When choosing between XML-RPC and SOAP it is important to look at what one needs. If there is a need to be able to define data types, SOAP is the choice. However if a simple system supporting the most basic stuff is enough, XML-RPC should fulfill the need. Since our need for the Job Manager was relatively simple, we choose XML-RPC as our primary candidate for an RPC protocol.

As mentioned we considered primarily XML-RPC and SOAP. Other candidates to do RPC could Java RMI [70] or CORBA [51]. However as the Job Manager is written in Python, Java RMI is not really an option, since it is Java specific. CORBA is much more than just an RPC system, and is everything but lightweight and simple. Therefore we decided not to go with any of these.

6.3.2 Selecting a Protocol

Even though XML-RPC was our first choice for an RPC protocol, the first RPC server we used was a SOAP server. The reason for this was that the pyGlobus [39], which provides a Python interface to Globus, had a SOAP server. The SOAP server was based SOAPpy [62] which is a SOAP [14] RPC implementation in Python. The SOAPpy module in pyGlobus has been augmented with GSI [5] support. GSI is an extension to SSL making it easier to use in grid contexts, mostly be integrating it into various application. This meant that we could easily create a SOAP server for the Job Manager to use, since it was possible to do authenticated RPC calls over SOAP using X.509 [27] proxy certificates.

Therefore SOAP became our natural choice for doing communication since it integrated nicely with GSI. Also SOAP is a good abstraction for doing distributed communication, since it just resembles a normal function call. Unfortunately, during the development of the Job Manager, we discovered what appeared to be deadlock, when

[2] Whatever that means.

using callback from Globus. An analysis of this is given in Appendix F. The result of this was that we either had to throw out NorduGrid ARC or pyGlobus. Dropping NorduGrid ARC was not really an option, since the Job Manager is highly dependent on it, so pyGlobus had to go. This meant that we had find another way of doing RPC. However, first we had to find another way to use SSL within Python. There are two libraries which provides SSL interfaces to Python: M2Crypto [66] and pyOpenSSL [67]. Both are based on OpenSSL library [61]. Since the last update to pyOpenSSL was in 2002 [67], we choose to use M2Crypto since this project was actively maintained. For RPC protocol we selected XML-RPC [75], since this is already supported in Python and M2Crypto provided the necessary extensions to use XML-RPC over SSL.

Even though M2Crypto has some support for XML-RPC, some work must be done to create the server. Most notably one must construct the server and setup the OpenSSL context [60] oneself. With knowledge of Python the construction of the server is relatively straight forward. Constructing a proper SSL-context however is not trivial, and requires in-depth knowledge of OpenSSL to set up correctly [76]. Fortunately the XML-RPC server worked well, after having been setup properly. Switching to XML-RPC also removed our biggest dependency on pyGlobus, and after having rewritten some code in the Job Manager we where able to remove the dependency completely. For the client this means that the Globus libraries are not needed, since GSI is not used. This means that a client using the Job Manager only needs an XML-RPC client capable of using SSL as transport. This makes it possible to write clients to a multitude of platforms.

The API of the XML-RPC server, is rather large, due to deep inheritance, however the following methods should explain how to use it, and in normal cases all that is required.

- `__init__(addr, ssl_context)`
  ăConstructor for the XML-RPC server. *addr* is the address to bind, specified in a tuple containing host name and port, e.g. `('localhost', 9443)`. `ssl_context` is the SSL context for the server and specifies host certificate, private key, certificate requirements for client, etc.

- `register_function(function)`
  ăRegisters a function into the server making it available for clients to call. Note that Python supports the functional paradigm, making it possible to treat functions as data. Everything regarding types and number of arguments is handled by the introspection[3] capabilities of Python. This makes it easy to extend the server since all that is needed to add an additional function, is a single line telling the server to make the function available.

- `register_instance(object)`
  Does the same thing as `register_function` except that this method takes an object and makes all of the methods on the object available on the server.

- `register_introspection_methods()`
  ăThis method makes three additional function available on the server: `listMethods`, `methodHelp` and

[3]The ability to "look into" objects at runtime, getting contextual information, such as name of functions and methods available on an object.

`methodSignature`. These functions allows the client to do some simple introspection of the server, e.g., to get the available functions on the server, giving a simple form of capability discovery.

- `serve_forever_thread()`
  Starts a new thread, which starts the server. The object representing the thread is returned to the caller, which regains control after the call, allowing it to continue working.

Summarizing these calls, a typical use of the RPC server is as follows:

```
server = SecureXMLRPCServer(('localhost', 9443), ssl_context)
server.register_function(submitJob)
server.serve_forever_thread()
```

After having explained how the XML-RPC server of Job Manager works, and which interfaces it has, the focus will shift toward the information system in the Job Manager.

6.4 Information System

To make decisions such as to which cluster to submit a job to, the Job Manager must have knowledge about the state of grid. To get this knowledge the Job Manager, must query the information system of grid. The information system of NorduGrid is covered in section 3.3.3. It is the purpose of the information system module to provide an API to access the information system. In NorduGrid ARC, the information system is consists of an LDAP [38] server running on each of the clusters. Examples of queries are listing of jobs which the users is currently running, or retrieving cluster information for job submission. The information system module must query the clusters and transform the received data into data structures, making it easy to use the information system. These data structures must represent the information queried. Therefore it makes sense to map the LDAP schema, depicted on Figure 3.2, on page 25, into objects, which can be used by the caller of the information system. These objects are: Cluster, queue, and job, each containing information about their respective entity in the grid. These objects are then created, packaged together and returned. Fortunately this functionality is already in the NorduGrid ARC, and the classes has been wrapped to Python, making them usable by the Job Manager. When the module has returned these it is up the caller, to do any further abstraction.

The API of the module reminds a lot of the one already existing in the user interface of the NorduGrid ARC. The main changes are the hiding of the GIIS concept, and the introduction of caching. The reason for hiding the GIIS away, is that it does not matter, for the user of the information system how the cluster list is retrieved, just that it is retrieved. The next section discusses the issues surrounding the cache. The API for the information system module is:

- `GetGiis(use_cache=True)` Returns a list of the top GIIS servers. This function is usually not needed since the `GetClusters` call retrieves this automatically, hiding the GIIS concept away from the programmer. However if the developer for some reason need a list of the GIIS servers, it is possible to get it. The `use_cache` flag sets whether a cached version should be used. Note that even if use_cache is set to True, and it has expired, a new list will still be retrieved.

- `GetClusters(mds_filter='JOB_SUBMISSION',anonymous=True, timeout=40,debug=0,sn=None,use_cache=True)` This returns a list of clusters, each containing a list of queue objects, each of those containing a list of job objects. The `mds_filter` flag specifies what kind of query that should be done. There are four possible values: `CUSTER_INFO`, `JOB_INFO`, `JOB_SUBMISSION`, and `JOB_MANIPULATION`. Depending on which value is passed along, different information is returned. For `CLUSTER_INFO` only information about the clusters is retrieved, for `JOB_INFO` information about jobs are returned. For `JOB_SUBMISSION` relevant information about clusters and queues for submitting jobs. Finally, for `JOB_MANIPULATION` only jobs that can by manipulated, i.e., the ones the caller owns, are returned. If an invalid value is passed along, an exception will be raised. The remaining parameters are more straightforward. The anonymous flag specifies whether or not to use authenticated queries. Timeout specifies the maximum interval in which the cluster should respond, before the query is aborted. Debug increases the verbosity of the function. The `sn` argument specifies the subject name used for certain queries, e.g., job manipulation where only the jobs of certain user is wanted. Finally the `use_cache` flag allows the caller to bypass the information system cache, which is used by default.

- `GetValidTargets(clusters=None,debug=0)` Returns a list of targets which jobs can be submitted too. A target is composed of a cluster and a queue. This is used as a convenience function when submitting jobs, since we are interested in targets in that case.

  The criteria for a target to be valid is:

  - Must have public keys, which are signed by a trusted authority.
  - Queue must have active status.
  - User must be authorized.
  - Queue must be non-full.
  - Queue must have CPUs available.

  Note that this does not say anything about having enough CPUs, or having the correct runtime environment for a job. The reason for this is that such requirements are specific from job to job. Creating a list of valid targets, but not worrying about whether it meets a job requirement allows the list to be reused when dispatching several jobs. It also separates the two things. Filtering targets which does not meet jobs requirements are of course done, but later in the process of a job submission.

- `ClearCache()` Clears the cache in the information system.

6.4.1 State Cache

Doing queries to all clusters within a grid can take a substantial amount of time since all the clusters must be contacted. This is especially a problem when a lot of information is updated sequentially, e.g., updating the status of several jobs. A solution to this is to retrieve the information once and used this for all the updates. This can be done in the following way (pseudo code).

```
job_info_list = RetriveAllJobInfo()
foreach job_info in job_info_list
    if job_info in job_list
        update status for job in job_list
```

Which is good, since it saves a lot of queries. However the resulting code is suboptimal, since it is not really intuitive; at least not compared to:

```
foreach job in job_list
    job.state = GetState(job.id)
```

Which is more obvious. However if the `GetState` function contacts the cluster each time, such an update can take quite a while. A way to solve this problem is to let `GetState` cache the results for reuse. Doing caching transparently leads to less and more readable code than doing explicitly.

Caching allows certain things, like sequential job submissions to be done significantly faster, but caching is also a delicate task, as there are several issues to take into account. Not all information about the grid can be cached for the same amount of time. E.g., a list of clusters are usually usable for longer amount of time than information about queue status or jobs. One of downsides of using cache is that the information used is older, than if it would be retrieved right before use. However, such retrieval can take some time, if there are non responsive servers, making the Job Manager wait for the information. It should be noted that information about the grid will always be a bit old and inconsistent, since it is not realistically possible to get a consistent snapshot of the state of the grid, since this would require an enormous synchronization effort between the clusters. Since it is not possible to get fully consistent information from grid, it might as well be cached for a while. What is important is to make sure that the information is purged from the cache before it gets too inconsistent.

The Job Manager does transparent caching, as descried in the previous, thereby hiding it away from the users of the information system module. The caching is done on two functions: `GetGiis` and `GetClusters`. It is not done on `GetValidTargets` since this uses `GetClusters` to build its list, and doing caching here would lead to double bookkeeping. Even though there are only cache on two functions, there are five caches internally in the module. One for the `GetGiis`, and four for `GetClusters`. One for each of the `mds_filter` options.

Cache expiration is done by having different expire times for the two functions. For the GIIS list it is one hour, and for cluster information 30 seconds. Even though the information system in NorduGrid has a valid-from and valid-to fields which marks the validity period of the data, it is not used. In an ideal situation this would be used to mark the end of life for the information, however it is not easily retrievable through the NorduGrid interface, so it was decided to use a simple timeout. If necessary the cache can be cleared manually by calling the `ClearCache` function. If the cache has expired it is first retrieved when a one of the functions using caching is called. This decreases load on the grid, since information is only retrieved when needed, instead of having a thread retrieve the information at a constant interval.

There are certain downsides of caching information this way. The cache system has no idea what data the caller want, making this operation more expensive than the same task performed by `ngstat` in some cases. An example is that for retrieving the status of a single job, will result in `JOB_INFO` queries to all clusters in the grid. However we have chosen to do it this way, since it is simpler, and provides a simple API to use.

After having explained the workings of the information system modules, the focus will shift toward the final module in this chapter; data management.

6.5 Data Management

The task of Data Management module is to provide the Job Manager with an API to handle the transferring of data. Data transfers can happen in several different ways: From the Job Manager to the grid, from the grid to the Job Manager, and finally from the grid to the grid. Since the NorduGrid ARC does not provide a "copying" service, copying from a grid location to another must happen by copying the file to the Job Manager and to the wanted destination. This functionality is basically the same as *ngcopy* provides, i.e., copying from one URL to another, and the module uses the same functions as this.

Data Management does not have anything to do with moving data to or from the client. Moving data to or from the client is done over XML-RPC since the Job Manager should not force the application to support grid protocols. When transferring files from the client and to the grid, or the other way around, the Job Manager will work like a proxy. To avoid overloading the Job Manager only small files should be transferred this way, since the whole file will be kept in memory in the Job Manager when using XML-RPC.

As mentioned the Data Management module has the same functionality as *ngcopy*. Therefore the API reminds a lot of the command. The Data Management module only supports a single function, displayed here:

- `URLCopy(from_url, to_url, confidential = False, blocking = True)`
  Copies a file from one location to another. It supports several protocols, which are listed here:
  - `file://` Local file.
  - `rc://` Replica catalog.
  - `rls://` Replica Location Service.
  - `se://` Storage Element.
  - `gsiftp://` GridFTP.
  - `ftp://` FTP.

  The copying uses message integrity as default, but is not encrypted. Encryption can be used by setting the `confidential` argument to `True`. Note that using encrypted transfers will cause a heavy load on the CPU. Furthermore it has the ability to act as a "copying service" by specifying the *blocking* parameter to false. This makes the Job Manager start a new thread and return control to the application before the copying is done.

The Data Management module does not feature advances features like creating a collection in Replicate catalogs and such. These are not often used, and has therefore been not implemented.

6.6 Summary

This chapter has presented some of the smaller modules of the Job Manager. These modules are, regardless of their size, necessary for the Job Manager to function properly, however due their size they did not warrant their own chapter. The two next chapters describes the modules that did; the managing of jobs and distribution of the Job Manager.

Chapter 7

Managing Jobs

Managing jobs is the heart of the Job Manager (hence its name). Making management of jobs robust and flexible is a critical aspect if the Job Manager is to become a usable tool. This chapter describes the considerations taken, its design and the construction of job management in the Job Manager. First an overview of the requirements for job management is presented. Hereafter the different parts of the Job Management module will be identified.

7.1 Considerations

As explained in Section 5.1, the user of the Job Manager should be able to extend or override the way jobs are dealt with, by the use of handlers. Even though handlers and job management are separate modules, as illustrated on Figure 5.2, they are closely connected, since handlers must be able to decide what happens to a job. Therefore the modules are closely related, and their interaction should be carefully considered, since this is an important aspect of the Job Manager. So far the handler concept has been described as a plug-in concept to the Job Manager, which allows the users to redefine and extend the way jobs are handled. Although this concept provides the user with much flexibility and power, it is also vague in its details. To overcome this we will start by analyzing how handlers should work and what implications they have on the rest of the Job Manager.

7.1.1 Handlers and Their Implications

To identify what handlers should be capable of, and how they should work, three examples of handlers will be analyzed. These three examples are: Another scheduler when submitting jobs, resubmitting jobs in case of failures, and automatic fetching of files from a completed job. These three examples all have that in common that they should be invoked when a certain state of the job arises. For the submitter it is the presence of the job in the Job Manager that should trigger it. The two last should be invoked when the job has finished. Of course the resubmitter should only be invoked if the job has failed, and the fetcher only if the job has finished successfully. Generalizing this; a handler should be invoked when a certain status of a job arises. For the handler interface to be as general as possible it should be invoked every time a change in the state of jobs happens. Since handlers should be invoked when a change occurs

in the state of the job, something external from the handlers will have to monitor to the jobs, and invoke the handlers when necessary. Basically there are two changes that can occur: The application calling the Job Manager or the grid updates information about the job. The first happens when the application makes an RPC call to the Job Manager, e.g., a job submission or cancellation. Since the RPC calls are translated into functions, these will have to invoke the right handler. Updates to the job in grid, will usually be updated job statuses. Unlike RPC calls, the Job Manager will not be notified of these updates, it will have to fetch them from the information system. Therefore the Job Manager should fetch the status of the jobs it is managing at a regular interval and invoke handlers if those jobs whose status has been updated. Summarizing the conclusions so far: Handlers are plug-ins that should be invoked when a change in a job occurs. This change can either be an RPC call from an application or an update to the status of the job. When one of these are invoked the Job Management modules will invoke the handler. This is depicted on Figure 7.1.

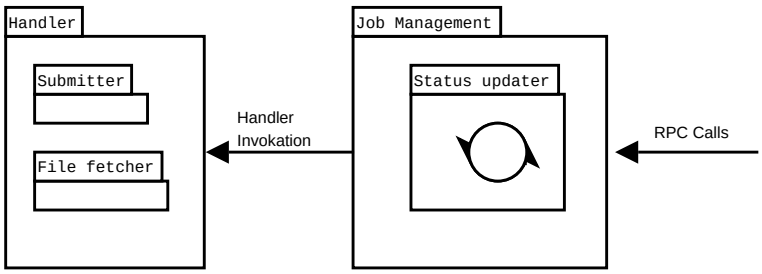


Figure 7.1: Overview of the handler invocation from the Job Management modules. When an RPC call is received or the status of a job is updated the handler is invoked. The boxes within the handler are examples.

Since handlers should be able to overwrite the default behavior of the Job Manager, the default behavior should be implemented as handlers. This will make them easier to change or overwrite, since they can be overwritten by writing new handlers, or even be taken out.

For now the focus has been on the interaction between Job Management and handlers and when they should be invoked. Now we will turn toward how handlers should be constructed in the Job Manager. Since handlers are plug-ins, we will start by looking at other plug-in structures. Drawing programs such as Skencil [37] and The Gimp [72] feature plug-ins allowing the user to create various effects to be applied to the picture. These plug-ins usually work by either loading the plug-in at startup or runtime. The plug-ins are then invoked when requested by the user, and a context is passed along. This context usually represents the canvas. The plug-in then manipulates the canvas, and returns control to the caller. Retrofitting this to the Job Manager, the context to pass along is of course the job description, e.g., an object containing information about the job, such as jobid and xrsl. Unfortunately this description alone is not enough. The handler will need to have access to the grid and be able to manipulate instantiations of job running on the grid. This is the difference between the plug-in structure and handlers, handlers causes side effects on the grid, they do not just operate on a description. To control job on the grid a library containing functionality for this is needed. As explained in Section 4.1 there exist no proper API for applications to use the NorduGrid ARC. Therefore a reimplementation of the job controlling functions is necessary. Another reason for such a reimplementation is that it is possible to separate the scheduling

from the rest of the job controlling functions. Such a separation makes it far easier to make a handler which overwrites the submission to use a new scheduler. Therefore this separation should be done when constructing the new job control library. Creating this library would allow handlers to manipulate the job, by giving them the necessary tools to work.

When handlers has the functionality to manipulate jobs, it also means that they can cause certain side effects when doing this. An example of this is a resubmission handler. Such a handler will resubmit a failed job, and thereby give it a new jobid. Since the handler has access to the job description it must update the jobid within, however if other parts of the Job Manager or the application are referring to the job using the jobid they will not be able locate it. This means that a jobids cannot be used as a consistent way of identifying jobs, since they are dependent on where the job is executing. Therefore an alternative way to identify jobs uniquely must be created. Such a representation are allowed to change during instantiations between jobs. A consistent way to identify jobs, which are not directly related to anything on the grid, will allow handlers to manipulate jobs without having to worry about side effects. Such an identification mechanism is therefore necessary if handlers are to work properly.

The plug-in structure have far reaching consequences as to how the module should work, and dictates much of how the structure should be in the module. Also a need to identify jobs consistently between instantiations was introduced. This need will be addressed shortly; first the parts of the Job Management will be identified in the next section. There it will be clarified how the modules interact; clarifying the requirements and concepts presented in this section.

7.1.2 Identifying Modules

This section will identify the modules that are needed for job management. From the previous it is already clear that two modules are needed: Job Management and Handlers. Furthermore a need for a job controlling library was introduced, which constitutes a third module. Finally a separate scheduling module should be created, to keep it separate from job submission. The job control library and scheduler are both invoked from the handler module, while handlers are invoked from job management. Starting from the bottom, we will explain the modules, starting with the Job Control module.

The task of the Job Control module is to provide job controlling functionality to handlers, making them able to manipulate jobs on the grid. The Job Control module will basically be a reimplementation of job control functions in NorduGrid ARC, and will provide an API instead of a command line interface. The module will provide functionality such as job submission and cancellation, i.e., functions that performs a specific actions on a job. The module will not do anything automatically, nor will it do any kind of bookkeeping, this is entirely left to callers of the modules. The Job Control module does not do scheduling; this is left to the scheduling module.

The reason for separating scheduling from job control is to make it easier for handlers to provide and invoke their own scheduler. Therefore a proper interface for scheduling will have to be created. It is also our hope that by separating the scheduler, will make it easier to experiment with scheduling. Currently this is rather hard, because the scheduling functions in NorduGrid ARC are buried deep in the job submission code. Since the scheduling module will only contain schedulers and perhaps some helper functions, the module will be relatively small, but provides important functionality.

The handler module will contain several handlers each specialized to perform one action, e.g, submission, resubmission, or fetching of output files. The handler will use the Job Control module to control its jobs, while the handlers will be invoked from the Job Management modules. This makes the handling modules the middle layer between bookkeeping (Job Management) and the actions (Job Control), and the handler module ties these layers together. This, combined with the ability to override or extend the handlers makes this layer very powerful, and critical for the success of the Job Manager. Updates to jobs are most likely to come in chunks, e.g., the status of all the jobs has been updated, or the application has submitted a set of jobs. Handling all these jobs at once may cause the Job Manager, to become heavily loaded for a while, if dealing with large sets of jobs. Therefore the Handler module should feature some form of queue making it possible to enqueue work, and allow the handler module to do the work as it sees fits. Additionally the handler module could form sub-queues for, e.g., submission or resubmission if needed.

The task of the Job Management module is to do bookkeeping of jobs and to invoke the handler when an update to a job occurs. The module contains a list of all the jobs and a description of each jobs. In this description information such as jobid and xRSL description should be stored. Furthermore since the use of handlers cannot be predicted each job should have a data structure which handlers may use to store and manipulate their information in, e.g., resubmission attempts. Such a data structure could be a dictionary or list.

As mentioned earlier updates to jobs can happen in two ways: By an RPC call from an application or by an updated status on the grid. Since the Job Manager is only notified when receiving RPC calls, it will have to pull the status of the jobs it manages at a regular interval. If any change occurs in the status of the job or an RPC call is received, the module will invoke the handler, which will threat the job accordingly. When invoking the handler the job description will need to passed along, and the handler will need to update it, e.g., if doing a resubmission, a new jobid will have to be set and the old saved. This means that bookkeeping is not done entirely in this modules, but also in the handlers. However it is necessary since the Job Management module does not know what the handlers does.

Summarizing this section, there is the Job Management module which does bookkeeping of jobs. It provides functions for the RPC server and also updates the status of the jobs at a regular interval. When a status of a job is updated, the job is pushed into a queue at the handler, which then goes over each job, checks if anything must be done with it, and dispatches the job accordingly, e.g., submits it. To control jobs the Handler module uses the Job Control and Scheduling Modules. This setting is illustrated on Figure 7.2.

7.2 Introducing Job Tags

Having explained the modules, we will turn the problem of uniquely identifying jobs and our solutions to it: Job tags. In NorduGrid terminology a job represents a single execution of an executable. Each job is described by an xRSL description which is uploaded to the cluster to execute the job. When submitting a job, a unique id is returned as an identifier of the job, this is called a job id. An example is:

```
gsiftp://benedict.aau.dk:2811/jobs/21268217461189425804
```

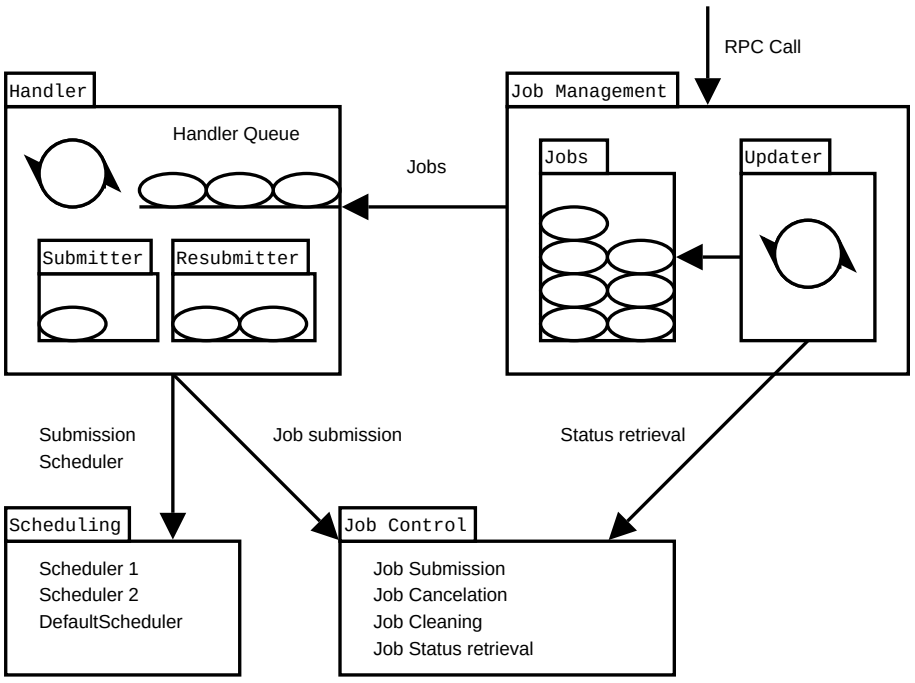


Figure 7.2: Overview of the module in job management and their interaction. The ellipses are jobs, and the circles with arrows threads. It is illustrated how jobs are pushed from Job Management to the Handler module which manipulates the job on the grid.

From this id, the cluster on which the job is being executed, as well, as the location of output data can be extracted. This representation works well with the NorduGrid ARC, however it is less suited within the context of the Job Manager. If a job is resubmitted, or moved to another cluster, its job id will change, i.e., it loses any reference to the previous job, even though it is the same job, only in another context.

Therefore, to support the features of the Job Manager, another way of representing jobs will have to developed. This representation must remain constant during several executions. In order to do so, the concept of a *job tag* is introduced. A job tag is an unique identifier for a job, which remains constant during instantiations of the job. This makes communication between the Job Manager and applications more consistent, as nothing needs to be changed if the job is resubmitted. Remember that an application can be also be another Job Manager. The job still has the same representation for the application. If the application needs the jobid of a job, it can retrieve it by doing a call to the Job Manager. Communication between Job Managers also happen using tags, to identify the jobs that they, e.g., monitor for each other. This communication will be described in Chapter 8. On Figure 7.3, it is illustrated where the Job Manager uses tags and ids respectively for communication.

Since a tag must identify a job it is important it is unique. Therefore when creating a job tag, it should be ensured that it is as unique as possible. To this random and unique data are collected for entropy. From this data a SHA1 hash [23] is created. The values used for entropy are: The pid of the job manager, user id of the user which the job manager runs, the host name of the machine, the current time, and a random number. Generating a hash that collides with this is statistically very improbable [22].

Figure 7.3: Where the Job Manager uses tags and ids in communication. Tags are used between Job Managers and application making job representation consistent between job instances. Ids are used when communicating with grid.

We decided not to embed any information into the tag, since such information would be likely to change, requiring a change to the tag, as it would be with, e.g., resubmission attempts. It is tempting to include information such as Job Manager contact information into the tag, but since a job can change between Job Managers, e.g, during a failover, thereby rendering the information wrong. Therefore the best solution is to simply generate a hash to represent the job. If any additional information must follow the job, it must be in another way than embedding it into the tag.

The tag must follow each instantiation of the job into the information system, making it possible to do external queries to get the tag. Unfortunately the information system in NorduGrid has no support for tags, or extensions that can be used.

Fortunately there are certain job attributes in the information system that can be set in the job description when uploading the job. These attributes can be queried through the information system. This means that there certain fields that can be controlled by the Job Manager. One of these is the job name, which is suitable for the purpose. When a Job Manager receives a job, it modifies the name of the job by appending three # and the tag hash generated as mention in the previous. This is the tag of the job. An example of a tag is listed here:

```
My_job_name###3dbba9cddce6c55526af4a2353e632a6b126f660
```

Setting the tag to be the name of the job, makes it possible to get and query for tag through the information system. However job ids are still the primary abstraction when manipulating jobs on the grid, since job names are not guarantied to be be unique in the information system. However when creating tags as described, hash collisions should be very unlikely. This does not assure that a tag cannot be re-created since a tag is just a job name in the information system. Therefore when searching and comparing tags in the information system, subject name of the submitter should be compared as well, to ensure that only the jobs of the right users is considered. If this comparison is not done, it opens the system to “tag poisoning” where jobs with identical job tags are submitted into the system.

Finally it should be noted that tags are not replacements for jobids, but complements it. Using only tags it is not possible, e.g., to identify a certain instance of a job, which is possible using jobids. The argument can be reversed to favor tags. Additionally an extension to the Information System, where the tag could be kept, could be considered. However to make it more useful than the name replacement, there would have to be a guarantee to keep the tag unique for every instantiations of the job. This would certainly add to the complexity of the information, so it is not fully clear if this would be an advantage. After having explained tags, the modules in job management will be discussed; starting with Job Control.

7.3 Job Control

The purpose of Job Control is to provide low level control of single jobs. Basically the module does four things: Job submission, cancellation, cleaning, and retrieval of status. This functionality is the same as the command line interface offer for controlling jobs, and the module is basically a reimplementation of the functionality in this. However the API has been reworked, making use of, e.g., exceptions for handling failures. Also, instead of the functions spanning hundreds of lines, they have converted to small functions, where the different functionality has been split into different functions. An example of this is that it is easy to replace the scheduler used in a job submission, by writing a new scheduler in a different function and changing a single line where job submission is invoked. The scheduler interface will be covered fully in Section 7.4. Such separation has been done throughout the code to make it more comprehensible and modular. Hopefully this modularization will make it easier for people new to the NorduGrid ARC or the Job Manager to understand the code, and make it easier to replace part of the code, thereby making it easier to do development and experiments.

7.3.1 Job Control API

Since the Job Control module is a reimplementation of code in NorduGrid ARC, jobids are the primary abstraction. Keeping tags out of the module means that the module can be used as a replacement for some of the existing code in NorduGrid ARC, provided that a command line interface is build to use the API of the module. This is strengthened by the fact that the module does no bookkeeping; it must be done be the caller. The design and implementation of the Job Control is relatively straight forward since it is mostly a reimplementation with the purpose of providing an API for controlling jobs. The API is:

- `SubmitJob(xrsl, scheduler = None, dryrun = False, dumpxrsl = False, debug = 0)`
  This is the high level function that submits a job. It takes an `xrsl` description. The functions retrieves a list of clusters in the grid, creates a list of valid targets. Hereafter it uses `GetSubmittableTargets` to get the targets which fulfills the requirements of job. Hereafter `DispatchJob` is called with the new target list and xrsl description. Furthermore the `scheduling`, `dryrun`, `dumpxrsl` and `debug` parameters are passed along to it. The meaning of these arguments will be explained in the following.

  - `GetSubmittableTargets(target_list, xrsl, timeout = 40, debug = 0)`
    Given a `target_list`, i.e., targets that have passed the validity test described in Section 6.4 and an `xrsl` object, this function returns a new list of the targets which fulfills the requirements of the jobs. Examples of requirements are runtime environments, number of CPUs, disk space or a specific architecture. `Timeout` is only used in an internal function in the NorduGrid ARC, which calculates the needed file size, and does querying. The `debug` parameter make the function more verbose.
  - `DispatchJob(target_list, xrsl, scheduler = None, dryrun = False, dumpxrsl = False, debug = 0)`
    This function takes a list of targets and an xrsl description. The target list

will then be sorted, either using the scheduler provided or using the default scheduler if none is provided. After this, submission will be attempted for each target in the list, halting if the submission succeeds. If submission fails to all targets an exception is raised. If the `dumpxrsl` is set, the xrsl to be uploaded will be printed to standard out. The functionality of `dryrun` is explained in the next.

- `PrepareSubmissionTarget(target, xrsl, dryrun = False)`
  Given a target and an xrsl description, this functions creates a new xrsl object within the target. This xrsl object is prepared for submission, i.e., the relevant attributes from the xrsl are added, along with information such as which queue the job should go to. The reason for creating an xrsl object for each attempted target is that each target is different and the information to embed into the xrsl is different. Modifying the original description would destroy its state. If `dryrun` is set, an attribute will added to the xrsl, indicating that the job should not be started when received by the Grid Manager.
- `UploadJob(target, debug = 0)`
  Uploads an xrsl description to the given target. The xrsl description is in the target object already. If the upload fails an exception is raised. Figure 7.4 illustrates the calls made to submit a job.

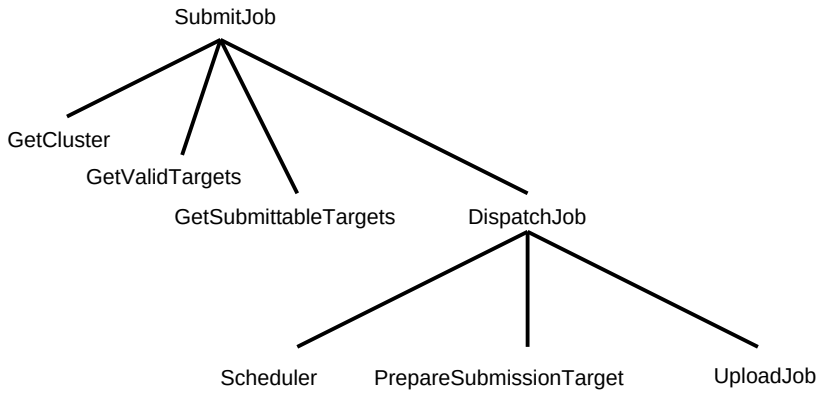


Figure 7.4: The call graph for SubmitJob, showing the API calls made during a job submission.

- `GetJobStatus(jobid, debug = 0)`
  This function returns a string containing the status of the job, given a `jobid`. If `debug` is set, the function is more verbose. If the job is not found, an exception is raised.

- `CancelJob(jobid, debug = 0)`
  Given a `jobid`, the function will cancel the job, i.e., kill it. If the job is not found or is no longer running an exception is raised. Note that the job may already have finished, even though information system says otherwise. Therefore the exception is merely a notification to the caller. If `debug` is set, the function is more verbose.

- `CleanJob(jobid, debug = 0)`
  After a job has finished, its session catalog will contain various files, i.e., output data. Given a `jobid` this function will clean up the session catalog of the job, i.e., remove everything in its session catalog. If the job has not yet finished an exception is raised (the job is not canceled). `Debug` make the function more verbose.

Having discussed the Job Control module and explained its API, we will turn to the other module used by the handlers; scheduling.

7.4 Scheduling

The scheduling module is the smallest of the four modules in Job Management. Its only task is to provide a scheduling interface and at least one scheduler, to use as default. In Section 7.3 it was already outlined how scheduling works. The `DispatchJob` function is provided with a scheduling function as its arguments. This function is called with the target list as argument, and must return it, where the first target is believed to be the best and the last the worst. `DispatchJob` will try each target in this list, starting with the best. This means that writing a scheduler corresponds to writing a function that sorts a list of targets in descending order.

7.4.1 A Scheduling Example

The Scheduling module does not have any real API, since it is just a collection of functions. It must however contain an object called `DefaultScheduler` containing the default scheduling function. Instead of an API, an example of a simple scheduler is given, i.e., one that sorts the target list after how many free CPUs the target has:

```
def MostFreeCPUsScheduler(target_list):
    def MostFreeCPUs(target1, target2):
        cpus_free1 = target1.queue.GetUserFreeCpus()
        cpus_free2 = target2.queue.GetUserFreeCpus()

        if cpus_free1 > cpus_free2:
            return -1
        elif cpus_free1 == cpus_free2:
            return 0
        else:
            return 1

    target_list.sort(MostFreeCPUs)


DefaultScheduler = MostFreeCPUsScheduler
```

The example contains a nested function, which is used to compare targets and return a value indicating which is the best. The built-in sort routine on `target_list` will then use this function to sort the list. Finally the `DefaultScheduler` is set to point at the `MostFreeCPUsScheduler` function, making it the default scheduler. It might seem tempting to always use a comparing function like the one in this example since it is a rather nice way to sort the targets. This however, would limit the flexibility of the scheduler functions, and we do not wish to dictate how the sorting should work. Returning to `DispatchJob` the target list will now be sorted, and job submission can begin.

This form of scheduling is only applicable to one job at a time. Therefore when submitting multiple jobs, the scheduler must be invoked each time a job is submitted. This is not a problem since an average job submission still takes a lot longer time than sorting a target list[1]. It might be tempting to construct a mass scheduler, when submitting several jobs. However, jobs has different requirements and will therefore only work when the jobs have similar requirement, i.e, parametric modeling. Instead of making a scheduler handling several jobs, the current interface can be expanded to handle multiple submissions in a better way. With the above example the target with the most free CPUs are always put first, i.e., the best target comes first. However, remember that the Job Manager uses caching in its information system, and that this information will be reused when submitting jobs sequentially. Given the scheduler in the previous example, all the jobs will be submitted to the same target. Therefore a bit of randomness should be added to a scheduler function. The best target should have the highest probability of coming first, however other targets should have a chance of getting first as well. This probability should fit accordingly to how "good" the target is. Adding this randomness will make a scheduler distribute jobs more evenly when submitting multiple jobs.

Another way of solving this problem is to modify the data to schedule by after a submission. For instance when submitting a job requiring two CPUs, the target to which the job gets submitted will have two subtracted from its free CPU count. This method has two problems. First of all it is hard to do in a consistent manner, since some CPUs may appear on several targets, making it not obvious which targets to manipulate. Secondly it only accounts for the submissions done by one self. Other submissions cannot be accounted for, unless pulling from the information system again. And when received, the information is still a outdated. Therefore it is doubtful whether this manipulation can provide something that the previous method cannot. A better solution would be to monitor the grid, and move jobs between queues as new clusters appears as queue skew becomes apparent.

Having introduced the scheduler interface, given an example of a scheduler, and explained for its workings, the focus will shift toward are more complicated matter within job management; Handlers.

7.5 Handlers

The Handler module is the glue that binds job management together. Beneath it is the Job Control and Scheduling modules which it it uses to control jobs on the grid. Over it, sits the Job Management module which invokes the handlers. The task of the Handler module is to decide what to do, when invoked. An example of this is when a job appears in the Job Manager, Job Management will invoke the handler, and the handler will submit the job to the grid.

7.5.1 Handler Invocation

As already mentioned, there are two kind of events that make the Job Management module invoke the handler: RPC calls and new status of jobs. It was also mentioned that a status updater was needed, since the Job Manager is not notified of this; it must be pulled from the information system. In this section it will by identified how the

[1] At least as long as the target list remains under a couple of thousand entities.

Handler module will work. To do this we will start by looking at the when and how the Handler should be invoked.

- **Updated Job Status –** When the updater queries the status of the jobs from the information system, it should update the status in the job description and then invoke the handler system with this job. The handler reads the new status of the job, and handles it accordingly.
- **Application RPC call –** Whenever an application calls the Job Manager, and the call manipulates a job, the handler system will need to be invoked. There exists three such calls, listed here:
  - **Submit job request –** When the Job Manager receives a job submission request the Job Management module should setup the necessary job description, and call the handler. The normal case for this request would be to just submit the job, however if no suitable resources are available, a handler may decide to postpone the submission until a useful resource appear.
  - **Cancel job request –** When a job cancellation call is received the Job Manager should cancel the job on grid. This situation is complicated, since the Job Manager may be doing something else with the job, e.g., resubmitting it. Also if the job has just been submitted, it may not yet have appeared in the information system. If this is the case the Job Manager will have to wait for this to happen before canceling the job.
  - **Clean job request –** If the application request that the job should be cleaned, i.e, have its output files and session catalogs deleted, the handler should be invoked. Such a request can also be complicated to carry out under certain conditions, e.g., if a handler is downloading the files from a finished job, the request would have to wait before being carried out.

At each invocation it is the task of Handler module to decide what should be done with the job. As mentioned, many of these invocations can lead to race conditions, e.g., what happens if a cancellation request arrives before the job was submitted. The implication of this is that the Handler module will have to keep track of what is currently happening to a job, and either cancel that action or wait for the handler to finish, before performing the requested action. These conditions are likely to be a concern when a cancel or clean request is received, since the Job Manager will have to stop what it is doing and perform the request. A job submission call is not prone to this, since it is not a change to a job, but the creation of one. Related to this is what happens when the application asks for the state of the job, and the job has not yet been submitted or appeared in the information system. To solve this problem the Job Manager introduces to new statuses: `IN_JOB_MANAGER` and `SUBMITTED:cluster@queue`. The first marks that the job has entered the Job Manager, but has not yet been submitted. The second indicates that the job has been submitted, but has not yet appeared in the information system. Furthermore it displays to which cluster and queue to the job has been submitted, i.e., a cluster and a queue name. These two states only exists between the Job Manager and application, no extension to the information system is necessary. This addition removes the problem of getting a status for a job, which has not yet gotten a status from the information system, a problem which the `ngstat` command line tool is suffering from.

In Section 7.1.2 it was mentioned that a lot of updates or subsequent RPC calls, would invoke a lot of handlers at once. To solve this problem the concept of a handler queue was introduced. Having such a queue means the Handler module will have at least one thread to do work, since it is not directly called by Job Management. Having such a thread would also allow the caller of Handler module to continue without having to wait for the Handler module to finish. This is a good, since some of the handlers can potentially take substantial amount of time to complete, e.g., fetching of output files. Giving the Handler module its own thread might not even be enough, since the thread can only handle the jobs in the queue sequentially. If the Handler module is to fetch output files from a set of finished jobs, this will probably congest. Therefore the Handler module should spawn new threads for tasks that may take a significant amount of time. This will allow to main handler thread to continue its work. It might also be feasible to create sub-queues within the module, e.g., for submission. However each thread added, will add to the complexity of the module, so adding threads should only be done in the case of congestion problems. Even though the module will contain several threads, simplicity of use should be kept, making the sure that the internal complexity of the module is not exposed. An example of a handler is illustrated on Figure 7.5.

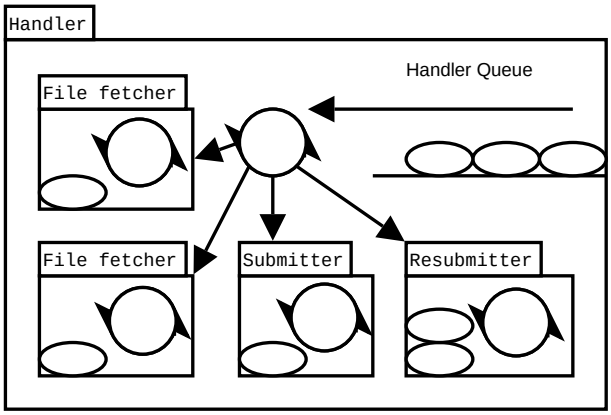


Figure 7.5: The Handler module with a thread handling jobs. Two file fetchers have been started and two sub queues are started. One for submission and one for resubmission. The ellipses are jobs, and the circles with arrows threads.

Moving on to the construction of the module we will look at which principles should be used when constructing the handlers. There are several requirements for what should be possible:

- **Creating new handlers –** The user should be able to create his or her own handlers and incorporate them into the module.
- **Overriding existing handlers –** It should be possible to override the default handlers with other handlers.
- **Selecting between existing handlers –** The user should be able to mix and match the handlers, both the handlers provided with the Job Manager and externally provided.

Given the two first requirements it is natural suggest that a handler should be implemented as a class. From this class the user can create a new class, which inherits from the provided class. This makes the user capable of creating a new handler by writing a method in the class, which are called when a certain type of handler is invoked. It would also allow to user to overwrite the existing handlers by using polymorphism. For the third requirement, something more flexible than inheritance and polymorphism is needed: Mixins[2]. Mixin allows each handler to be implemented in a separate class, containing only one method which name corresponds to what will be invoked for a certain condition, e.g., a job submission. Putting each handler in a separate class, allows the user to select the handlers that should be used by inheriting from the desired classes into a new class using multiple inheritance. For this to work there needs to be a super class, which the users class should inherit from. This class should define the interface for Job Management to use, along with setting up the work queue. The methods provided to the Job Management module should then invoke a method depending on which action should be taken. The superclass should contain empty methods for each of these actions, since these are the methods meant be overwritten by subclasses, the real handlers.

After having given an explanation of how the Handler module will be implemented, the section will be finalized by listing the API of the handler module:

- `StatusUpdate(job)`
  Used when a job has received a new status. The status of the job will be read and the handler for this status invoked. Usually called by the updater, after updating the status of a job, and the job had a different status than last. Should also be invoked for new jobs, since their `IN_JOB_MANAGER` status will make them go to the submitter handler.

- `CancelJob(job)`
  Should be called when a request for job cancellation has been made. If the job is in any if the handler queues it will be removed from there. Handlers running with this job will either be killed, or the function will wait for them to complete.

- `CleanJob(job)`
  When invoked the session catalog of the job will be deleted. If any handlers with the job is running, the method will allow them to finish, since they could be using the session catalog, e.g., downloading from it.

Note that the complexity of handler has been hidden by a small interface which should be simple to use for the caller. Having explained the API of the handler module, we will turn to the last module in described in this chapter: Job Management.

7.6 Job Management

The task of the Job Management module is to do bookkeeping of jobs and invoke the handler module when necessary. Furthermore it provides high level functions concerning jobs, many of them provided directly to the RPC server. This means that the module binds together RPC calls with the internal data structures representing the jobs. The module does not decide how to threat the jobs; this is the task of the handler module.

[2]Mixins is a concept using multiple inheritance to augment the functionality of other inherited classes; the difference between regular classes and mixin classes are that mixin classes cannot stand by them selves.

This section will start by explaining how jobs are described and how these are kept track of. Hereafter the job status updater will be described, and finally the API of the Job Manager will be listed.

7.6.1 Describing Jobs

Previous in this chapter, there has been several hints about what a job description should contain. The basic contents of a job description are: job tag, the xrsl description, its status, and a jobid, if submitted. Furthermore handlers must have a way to store various informations about the job, e.g., resubmission attempts and previous clusters. Since this data is related to the object, it should be stored along with it. Therefore a job description should have a container for handlers to use. Also, a flag indicating whether or not the job is currently is being handled is needed. As the Job Manager is highly threaded a lock to protect access to the job is also needed. This lock could also be used to protect access to the object while it is being handled, however this would cause the lock to be held for long periods of time. To overcome this it would be necessary to attempt lock grabbing in a non blocking manner. Doing this would complicate much of the code, making the Job Manager harder to understand and more prone to errors. Therefore both the lock and handled flag should be kept. The description, should naturally be kept in an object, which ties the data together. The code for the class is listen here:

```
class JobInfo:
    def __init__(self, tag, xrsl_string, jobid = ""):

        self.tag = tag
        self.lock = threading.Lock()
        self.xrsl_string = xrsl_string
        self.xrsl = Xrsl(xrsl_string)

        self.jobid = jobid
        self.status = NOT_SUBMITTED_STATUS
        self.handled = False

        self.local_input_files = {}
        self.handler_info = {}
```

The code has a few extra items not mentioned. Since the xrsl job description is send as a string to the Job Manager [3], it converts the string to an xrsl object, and the string is kept, should be needed later. Also a dictionary[4] of input files are created. This is used when one or more of the input files to the job are local. Since it is the Job Manager that submits the job, these are needed. The `handler_info` is for the handlers to store information in. Finally, note that the status of the job is set to a `NOT_SUBMITTED_STATUS`. This makes the handler module submit the job when invoked.

Having a suitable job description is not enough; there must also be kept track of these descriptions. For this the Job Manager uses a dictionary, where the tags are used as indexing keys, and a job object as the value. Since access to this is dictionary also happen concurrently a lock must be acquired before reading or modifying it. Given that each job and the tag dictionary has a lock, it is clear that some constraints for doing locking is needed to avoid deadlocks within the Job Manager. These locking constraints are:

- One must grab the lock of a job / the tag dictionary before reading or modifying

[3] Remember that XML-RPC does not support user defined types

[4] An associative array in Python

it. This ensures consistency of the object, so that an object is not modified and read at the same time or being modified by more than one thread at a time.

- If the handler flag is set, you are allowed to read from the object, but not modify it, except if you set the flag. The reason for this constraint is that when a handler is running it has the exclusive rights to the object.
- You are allowed to grab one job lock anytime, however if two or more job locks are needed you must drop any job lock holding, grab the tag dictionary lock, and grab the lock of the jobs. If more than one thread is allowed to grab more than one job lock at a time without going through tag dictionary lock, the Job Manager can deadlock. This constraints ensures that it does not.

These constraints ensures that the data in the Job Manager remains consistent, and that the threads does not disturb each other or deadlock. Therefore it is critical that these constraints are not broken. We will now turn to other big task of the Job Management module: Updating the statuses of jobs.

7.6.2 The Status Updater

When a job receives a new status, e.g., it is marked as `FINSIHED`, the Job Manager should invoke its handler. However the Job Manager is not notified of these updates, it must pull them from the information system. These updates must be done at a regular interval, since the status of jobs is updated continuously in the information system. In the Job Manager, this updating is done by status updater, which is a part of the Job Management module. The updater is a thread, which will update the statuses of the jobs in the Job Manager and sleep for a while. This process is then repeated. It uses the following procedure for updating the statuses of the jobs:

1. Collect the jobids of the jobs in the Job Manager. If the job is being handled or the status of it indicating that is has already finished, the job is skipped.
2. The status of the jobids representing the jobs is fetched from the information system.
3. The updater will reiterate over the jobs, comparing the jobids of the jobs with the ones that it fetched statuses from the information systems. This comparison is necessary since the job lock is not held during the polling from the information system. If the jobid is exists, the status of the job id updated. A special condition is for finished jobs, where the status will be set to either `FINISHED_ERROR` and `FINISHED_SUCESSFULLY`. These states are necessary for the handler to recognize whether the job was successfully finished or not. Finally if the status is updated, the handler will be invoked.

Continuing doing this the updater will update the statuses of the jobs, making the Job Manager able to react to changes regarding the job.

7.6.3 Job Management API

This section explains the API of the Job Management module. Much of this API is exported directly to the server, and a large part of also used by the Job Manager communication module. The API is:

- `NewJob(xrsl_string, input_files = None)`
  This creates a new job descriptions with a new tag. The xrsl description is sent as a string as explained in Section 7.6.1. Furthermore more the input files are sent a long as well. The `input_file` should be a dictionary, with the paths as keys and the files as values. From this a job description is created, which is inserted into the tag dictionary and the `StatusUpdate` method on the handler is called.

- `CancelJob(tag)`
  Given a tag, and that a job instance is running, this job will be canceled. The job description is not removed. If the tag does not exits or the job has already finished, an exception is raised.

- `ClearJob(tag)`
  Cleans up after a job on the grid, i.e., removes its session catalog. If the job has not finished or the tag is invalid an exception is raised.

- `InjectJob(job)`
  This function injects a job into the system, i.e., the job is inserted into the tag dictionary, but nothing more will be done. The updater will threat this as any other job though.

- `RemoveJob(tag)`
  Given a tag, the job description representing this tag will be removed and any running handlers, will be killed or be allowed complete. If the job is running on the grid, it is not canceled. The job removed will be returned. An exception will be raised if the tag does not exists.

- `GetJobStatus(tag)`
  Returns the status of a job, given its tag. If the tag does not exist an exception is raised.

- `GetJobidFromTag(tag)`
  Given a job tag this function returns the jobid, if any, to the caller. If the tag does not exist an exceptions is raised.

These functions provide a high level interface for job management for application to use. While the functions correspond much to those in the NorduGrid ARC command line interface, the jobs are managed, e.g., they have handler which can resubmit them if failed.

7.7 Summary

This chapter has described, what is perhaps the most critical part of the Job Manager, Job Management. Initially an analysis of how handler should work was made. The result of this analysis was that handlers would dictate most of how the construction of the Job Management should be. The analysis made it clear that the Job Management was not just one module, but four: Job Control, Scheduling, Handler, and Job Management. This separation works well since it provided much better separation of concern, and made it easier replacing parts.

Chapter 8

Distributing the Job Manager

This chapter discusses how to make several Job Managers work together managing jobs. The purpose of distributing the Job Manager is the same as for many other distributed systems, where the system needs to keep working even when failure occurs. We need to cope gracefully with failure. Therefore the Job Manager should deliver failure transparency. A way to achieve failure transparency, is to replicate the data and/or services in order to avoid a single point of failure [17, 18]. Depending on the method, distribution of the Job Manager could also lead to improvement in availability. Ideally total location and failure transparency, from the users/applications perspective, is preferable, but due to constraints discussed in this chapter this will not be achieved in full.

To make the Job Manager resilient to failure there should be no single point of failure. Distributing the Job Manager allows for the user to introduce redundancy by starting several Job Managers to manage the jobs. This leads to several interesting problems which will be discussed in this chapter. The problems discussed are already known within the realm of distributed systems so we start by looking at some general issues.

8.1 Issues

In order to distribute the Job Manager, several questions and issues must be resolved. The items on the list spring from the requirements and the features of the Job Manager and they need to be resolved for the Job Manager to function as wanted. The list are, for the sake of readability, divided into two sections. The first category has to do with failure, location, and replication transparency.

- **Failover** – How to make the other managers aware of a failure in one of the managers, and decide which manager has to take over and how this is achieved, e.g., election algorithms, discovery, and soft registration. Furthermore a failure detector is needed and the requirements and type should be determined.

- **Handover** – The ability to explicit push a job to another Job Manager, for instance, if the load on the manager gets to high or if the Job Manager in question does not have the capabilities to manage a particular job. To be able to hand a job over to another Job Manager, it must be possible to transfer all information and data the job needs to execute properly.

- **Availability of data** – The data and information necessary to manage a job needs to be available to all Job Managers, depending on the failover mechanism, for failover to function properly. A scheme for making this data available has to be developed.
- **Job Manager Discovery** – It is necessary to have a mechanism for new Job Managers to locate other Job Managers already running, in order to locate failover managers. This concerns the location of data and the arrangement of the Job Managers, e.g., MDS, hierarchy or P2P, explicit list of managers, user provided list.
- **Unique identification of jobs** – It is necessary to be able to identify jobs uniquely over time, even when job ids change due to resubmission, clusters disappear, etc. This has already been discussed in Section 7.2.

The rest of the problems are of a more general nature, and several of the pertains to the integration of the Job Manager into the NorduGrid ARC.

- **User authentication and transfer of credentials** – Depends on whether multiple users are able to run on the same Job Manager. If so there is an explicit need for credential handling. The issue of transferring credential to a new manager could be explored, as well as the security implications such a scheme would have.
- **Scalability and availability** – The solution should be scalable, at least to a certain degree, depending on the expected usage. If the Job Manager is started per user, scalability is not a big issue, as most users probably only needs a few Job Managers to satisfy their need for failover capabilities. If a Job Manager can serve several users the scalability requirements are larger, as many Job Managers will have several users and handover managers. Thus failure detection may include any number of Job Managers.

The problems regarding multiple users will be ignored since the Job Manager is used on a per user basis as explained in Section 4.3.4. Furthermore the scalability issues may not be severe, as it is unlikely that more than a few Job Managers will be started by each user. However, if Globus is replaced, making it possible for the Job Manager to handle several identities and manage jobs for several users, then all of the above mentioned issues becomes concerns regarding the scalability and robustness of the Job Manager.

All of the issues listed have an impact on the choice of communication protocol between the Job Managers, and it helps determine the type of communication and protocol should be used for the different tasks. In the next sections we take a deeper look at the issues listed in this section.

8.2 Models for Distribution

There are several ways to distribute the Job Manager. In this section we look at two general ways of providing failure transparency and discuss their pros and cons with respect to the Job Manager.

Every Job Manager holds some information about the jobs which it manages. In order to make Job Managers work together, this information must be propagated to the

other managers. The objective is to determine the best and most simple solution that are working with the rest of the NorduGrid ARC and delivers the necessary failsafe mechanisms. There are several schemes for handling the communication and replication of data between the Job Managers. Two possible solutions is presented below.

- **Active replication** – A job is managed collectively by a pool of job managers. When a request arrives from a client or the job changes its status, the request is multicast to call managers which handles the request. The application receives the result from the fastest manager.
- **Primary-Backup / Passive Replication** – A job is managed by one Job Manager which remains in control of the job unless the manager fails. This means that control is only transferred automatically to another Job Manager in the case of failure.

There are pros and cons for each approach. Common for both of these models are the need for some form of mechanism to detect if a particular Job Manager has failed. One type of failure detector is some form of registration protocol with timeouts, a push model. Another failure detector is a pull model, where the Job Manager asks the other Job Managers if they are alive. In a asynchronous distributed system like the Internet, is impossible to implement a reliable failure detector solely by message passing [17]. The choice of failure detector will be discussed further later in this chapter.

8.2.1 Active Replication

Active replication provides a high level of fault tolerance and in addition it also provides high availability since all the Job Managers can respond to requests from the client. Basically every request is processed by all managers and in this way a consistent state is maintained in every manager. In order for active replication to work there is a need for total ordered multicast, to ensure that every manager reaches the same state [17].

The main argument for using active replication is robustness and availability since the bottle neck of having one manager to handle all jobs does not exist in this model. There are two problems with the approach in the context of the Job Manager. First of all we cannot deliver total ordered multicast since the network is asynchronous. Thereby we cannot ensure that all managers agree on a consistent state since the requests can arrive in a different order at the different managers. Secondly for this approach to work all operations on jobs would have to be idempotent in order to prevent multiple submissions of jobs from the managers. There are no obvious way to implement job submission as an idempotent operation in the NorduGrid ARC. Because of these inherent problems of using this model in our context we look at passive replication instead.

8.2.2 Passive Replication

A solution using passive replication imposes a ordering between the Job Managers. One manager is the primary, accepting request and performing the job management. It is only in the case where it fails, that another Job Manager takes over.

Passive replication is traditionally associated with a relative large overhead, since data is replicated between the primary and the backup, whereas the active replication performs the same functions concurrently. The overhead of data replication between

the primary and the backups, is small because it is only the main job information, e.g., the job description, that has to be distributed, and not the data associated with the job.

The small amount of data that needs to be replicated, makes passive replication preferable. Furthermore it is simpler to implement. The drawback is that this method does not increase availability as active replication does. However since the Job Manager is expected to spend most of the time sleeping, while jobs are being executed at a cluster, this is not a major issue.

This model is not free of communication between the backup managers since this model requires explicit failure detection. If the primary manager fail there is an explicit need to reach consensus of which manager should take over. In order to do this, the failover managers should continuously be able to check if the master is running. Additionally the failover managers should know of each other in order to reach consensus regarding the new primary manager.

8.2.3 Discussion

The two models just discussed are focused on high availability of data. A lot of the requirements that makes it difficult to use them in our context stem from this. In the job managers case a lot of the data are actually stored outside the Job Manager. In addition to this, is the fact that most of the data stored at the Job Manager does not change once the job has been submitted. An example of this is the xRSL job description, which does not change once it has been transferred to the Job Manager. The data that are subject to changes, are the meta data concerning the job, i.e., resubmission attempts and previous clusters. Even if the meta data are inconsistent, it is not a catastrophe that prevents the jobs from being managed[1]. The important job information such as the current cluster the job is running on, is accessible through the information system. This means that the requirements of consistency can be relaxed a bit and this makes the distribution of the Job Manager simpler.

8.2.4 Job Manager Failure Considerations

Generally the Job Manager has to cope with two types of failure in this type of system: Job Manager failure and network failure. Job Manager failure means that the Job Manager or the host on which it is running fails or crash.

Networks can fail, e.g., a router can crash causing the network to becomes partitioned. In a partitioned network the route between some nodes in disappears. It is important to note that network failures can be both intransitive and non-commutative with respect to routes between nodes in the network [17].

A similar problem is network congestion. The network is working but it is very slow preventing packages from reaching their destination before timeout. Though this is not an error in the strictest sense, it is impossible to determine if the network is slow or a host has crashed. This observation illustrates a fundamental problem with asynchronous distributed systems and explains why there are no way to implement a reliable failure detector by the means of message passing alone. Even though we use TCP, which is a reliable transport and the message is almost guaranteed to reach the manager eventually, because TCP cannot recover from massive network failure [71]. This have an impact on the use of passive replication.

[1] It may actually not be preferable to merge the meta data if the management of jobs have branched, due to network failure, since the meta data should follow the individual executions.

Since we cannot prevent failure, we will examine three types of inconsistent states failure can result in. The three problems are lost jobs, job duplication, and jobs that are managed by multiple managers. These will be discussed more in depth in the actual model used for distributing the Job Manager.

The failure conditions means that additional measures should be implemented to detect and correct these problems, i.e., there should be some mechanism for detecting duplicated jobs. Depending on the method, we run the risk of killing all instances of a job in case of failure. A solution could be to check if the job is already running, before submitting making job submission somewhat idempotent. A job can be identified by the job tag, and therefore duplicates can be identified using the tag.

The main function is the soft-state registration mechanism. The manager must register with its failover managers, or the failover managers must check the master, periodically. Furthermore the master must push any updates in the jobs meta data to the failovers. In order for another Job Manager to take over the management of a job, it needs to have the job information.

In this section we disregard the byzantine failures, causing the Job Manager lies about the jobs which it is managing. This have we done for two reasons; the mangers get most of their info from the information system, and thus it is considered correct. Furthermore, if the Job Manager lies about the job tag, the job cannot be located, and a manager cannot do anything about jobs it cannot locate. This will result in identical jobs running under different tags, an undesirable situation but difficult to prevent.

8.3 Job Information and Job Data

This section discusses how to ensure that the information and data associated with a job is propagated between the managers. For a manager to be able to take control over a job originating at another Job Manager it must have access to the job information, i.e., job tag, job description (xRSL), and current job id. Furthermore, the manager must have access to job meta information (submission attempts, etc.) and job data.

The main job data should be available as it has already been discussed, but the manager also needs the job description, along with the meta data. Strictly speaking the meta data is not necessary, since it only for information purposes and has no impact on the possibility to submit jobs. However in the case of failure it would be preferable to the user if this had been transfered along with the job description.

It is not possible to keep a consistent “state” of the grid [17]. As discussed in Section 8.2, the only information in the current information system[2] is the job id, and there are no support for job tags. Furthermore all meta information regarding a job is not available in the information system either. There are several ways to solve this problem. The information system could be modified to provide a way to acquire the needed information. The main problem with changing the information system, is that the information system is a core component of NorduGrid ARC, and since NorduGrid is a production grid, it is unlikely that changes to the information system will be accepted. This imposes severe restraints on our possibility to test the Job Manager and thus forcing us to consider this during the design phase. One solution to this predicament is to let the Job Manager assume the role of information provider outside the scope of the information system, for the purpose of job specific information. The problem with this solution is, that it leads to ambiguity with respect to the location of information

[2] Actually the information is stored on the resources in the grid. The information system merely holds the contact information of the contending resources

in NorduGrid, as there are suddenly two places to locate the information. The proper solution would be to extend the information system, but since this is not possible the necessary job information should be transferred between the Job Managers by other means. One way of doing this is to transfer the information upon registration, and then subsequent push updates to the failover Job Managers upon changes in the jobs meta data.

If the information was to be included in the information system, an LDAP schema describing the Job Manager should be developed. As for all other resources in the NorduGrid ARC, the information system would only hold the contact information and the Job Manager should implement a service provider to answer queries about the Job Manager as well as queries about jobs meta data.

8.3.1 Storage of Job Data

Another issue is the storage and availability of data associated with a job. For the Job Manager to be able to submit a job it needs to have access to the input data needed by the job. This has to be ensured. Furthermore since the job is submitted by the Job Manager all data residing on the client should be transferred to the Job Manager or the grid before submitting. Several solutions to this problem exists:

- Have a storage area at every Job Manager where job data is stored. Such a storage area may be necessary anyway for storing temporary data. Additionally the Job Manager could register itself as a storage element and this way allowing data to be transferred using the normal operations.
- Use some alternate mean outside the grid for storing data, e.g., Freenet, Pastry, or Coda.
- Always use SE/RC to store job data, including input and output data, thereby removing the problem of having the Job Manager manage the data and letting the Grid Manager fetch the data.

All of these solutions raise some concerns. A storage area at every Job Manager is necessary under all circumstances, to store temporary data, when submitting jobs. But registering it as a storage element and using it when submitting data does not improve fault tolerance, since other Job Managers are not able to submit the job if the Job Manager, holding the input files, crash. Also, since applications executed in a grid environment can have rather large input data sets, it is also clear that job data cannot be distributed between every Job Manager, as it would put unacceptable high load on the network and require a lot of storage and bandwidth.

By using a storage element to store job data, the Job Manager becomes reliant on another entity in the grid, thus substituting on single point of failure with another. If the storage element becomes unavailable, for some reason, the Job Manager is unable to submit jobs until the storage element is available again. This problem can be countered by replicating the data on several locations in the grid, at the cost of additional traffic and space.

The solution of going outside the standard grid element and using a foreign storage partially solves the availability problems are solved. This idea poses yet another problem, as the grid software becomes dependent on other services, such as a peer to peer network, like Pastry or the Coda filesystem. It is important to consider quality of service of the external services, e.g., the two examples both guarantees availability of

data and upper limits for searches, whereas Freenet does not [17, 64]. But the idea of doing this outside the grid is not preferable, especially if existing grid resources can already be used.

Replication is already possible in NorduGrid ARC, by means of the replica catalog, the only drawback is that this method requires the explicit replication of data. However, work is under way to develop a Smart Storage Element (SSE), which should provide support for automatic replication [45]. Though the timeframe for the development is not known.

In our opinion, the best solution is to use a storage element. Doing so prevents the Job Manager of becoming dependent on other services outside of NorduGrid, especially since the needed replication is planned. This solution leads to relatively high availability, but it has a problem since the storage element becomes a single point of failure. This does however not lead to big concerns since most of the storage elements are high performance computing resources, which have designated administrators, and therefore the availability can be expected to be high. None the less it is still a single point of failure and are subject to failure. As already described it is possible to replicate the data manually on several storage elements using the replica catalog. Furthermore the advent of the smart storage element delivers a solution to this problem and thus it would be a waste of resources to solve this problem twice.

This imposes further requirements on the Job Manager, in order for this solution to work, the Job Manager must check the job description to see if all the needed data are located at a storage element. If this is not the case, it is the responsibility of the Job Manager to upload the files, if available, and modify the job description accordingly.

8.3.2 Two Models

In this section we examine two models for distributing the Job Manager. The first model takes some of the ideas described under active replication, though it uses a primary-backup solution for replicating data.

In this model all Job Managers can handle request from the client. It differs from the active replication, since the request are not submitted to all the mangers to keep a consistent state. Instead data are replicated between the Job Managers. This makes the model more communication intensive and raises concerns about the consistency of data, since the data is not replicated instantly, i.e., the Job Managers does not have a complete picture of the state of the grid. The model illustrated on Figure 8.1, is a rather “naive” approach and it is unnecessarily ineffective but none the less it serves to illustrate the basic idea.

This model is rather communication intensive since job meta data of every job must be propagated to every Job Manager running, every time a change occur. Since the requests can only be handled by one manager, to prevent jobs being submitted multiple times, the model requires that the managers coordinate every action performed on a job. This requires that every Job Manager knows of each other and they have a way of reaching consensus. Even though consensus cannot theoretically be reached [17], it should be possible to implement a satisfactory solution in our case, since the duplicated jobs can be handled by other mechanisms.

However, this scheme also has problems in larger perspective since it does not scale well. The problem is that any decision about a job must be coordinated between all the managers. In addition to the distribution of the job data. The problem with the coordination of job management may not appear to be a problem since jobs often require no attention between the time when they are submitted until they are finished. This usage

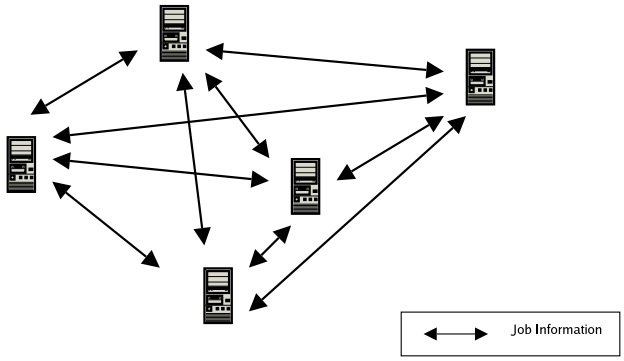


Figure 8.1: A naive network of Job Managers. All information about jobs and Job Managers are shared among all of the nodes in the network, resulting in a unnecessary high network load.

pattern may, however, change, as the introduction of a Job Manager, which provides more control of the management process. A rise in the actual need for managing jobs may lead to problems with management overhead.

A way to minimize the data transfer between the Job Managers it to use a model based on passive replication. Furthermore we can also take advantage of the information system. In this model we introduce the concept of a master manger. In this model the master manger is the only one accepting requests from the client, it then distributes the data to the failover managers on the network.

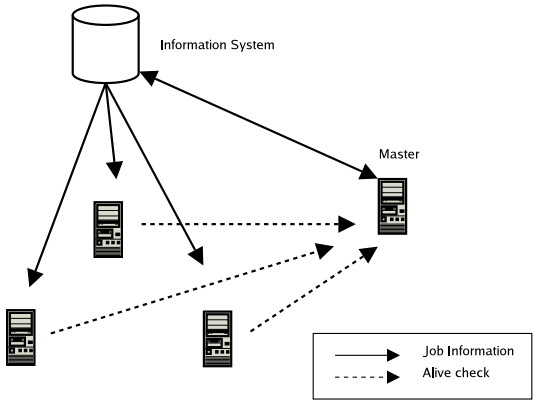


Figure 8.2: The master is the frontend, which the clients contact. The information about the jobs are propagated to the clients through the information system (some in formation has to be pushed explicitly, e.g., the xRSL description, in the current information system). The failover managers checks periodically if the master is alive.

By using the latter scheme, the scalability issues are less severe. Though the replication is still the same amount, there is only need for coordination if the master fails. Furthermore, by taking advantage of the information system the amount of job information that has to be transferred directly between the managers can be brought further down, by only distributing the most necessary data continuously (e.g., list of managed jobs), and let the failover get the rest from the information system only when needed. This model is illustrated on Figure 8.2.

There are some problems with the approach of using the information system. The first is a practical obstacle: The NorduGrid information system does not support all the information necessary, e.g., there are no support for submission attempts, old clusters, job tags, and location of data. In order to support this model, the information system therefore has to be extended. Changing the information system will have a tremendous impact on the entire NorduGrid, and it will be hard to convince the developers and users that this change is necessary without having demonstrated that the Job Manager approach works.

This leaves us in a situation where the information system only hold job names and current job status. The rest of the information needs to be propagated between the master and the failover managers. If this scheme is used techniques to reduce the amount of traffic between the Job Managers, should be used. One possible technique is to lessen the number of failover managers which a particular master registers with. This comes at the price of lesser robustness, all though this may not be noticeable in practice. This modification brings us closer to the previous model.

The fact that the improvements considered brings the models close together, makes us consider a hybrid model combining the two, could be considered, where a pool of manages work together following the first scheme. At a higher abstraction level, the pool is treated as one manager and the second scheme is used, this model could have the advantages of both models, and make the system more scalable.

8.4 Discovery Methods

In order to achieve location transparency, mechanisms that support automatic location and discovery of the Job Managers should be in place. This issue have two sides: Mechanisms for the application to locate a Job Manager when it is started, and methods for the Job Managers to locate each other.

8.4.1 Application

From the clients perspective, the problem is the bootstrapping process of locating the first Job Manager. When the first Job Manager is located rest can be automated. Basically the discovery can be done in two ways, either for the user to explicitly identify the Job Manager, by address or hostname, or to query the grid resources.

The first approach is by far the simplest and it has some advantages, as well as some disadvantages. The advantages are that it is likely that the user know where the Job Manager is running and therefore it is not a problem. This is of course a disadvantage if the user do not know where the Job Manager is located, or if the Job Manager has crashed and the user does not know of any other managers. The user may also have some explicit knowledge of the capabilities (e.g., handles) of a particular manager, and thus wants to have the jobs managed by this manager and not have one selected automatically.

The disadvantage is that this solution does not provide total location transparency for neither the user or the application. This problem is not a big issue since the Job Manager is most likely to be started by the user who uses it and therefor knows the location of it. The Job Manager have the mechanisms for automatically locate other Job Managers, so when the application has located and contacted a Job Manager, a list of known Job Managers can be fetched the application. This list could be stored and used subsequently to automatically locate the new master in case of failure. It is

important that the client locates the master manager, since it is the only one which are able to make decisions about jobs running on the grid.

If the client should be able to locate the first Job Manager automatically, it would need some way of querying the information system, to locate managers running on behalf of the user. This solution constitutes a problem since it goes against the idea of separating the grid functionality from the application by putting it into the Job Manager. To be able to query the information system, the application becomes highly dependent on Globus and NorduGrid ARC, which is not desirable. Another solution could be to implement an LDAP module able to talk to the information system and have the application link against it. The last solution my be feasible, but would require a lot of work with relatively little gain.

8.4.2 Job Managers

The Job Manager should be able to run without user intervention, and support automatic failover. Therefore it is necessary that Job Managers can discover and locate each other automatically.

When a new Job Manager is started, it needs some way to locate other managers belonging to the user. In Section 8.3 several solutions to the problem of information sharing was proposed, one was the storing of contact information in the information system. This solution would solve the problem, as a new Job Manager could query the information system, getting a list of job managers it could register with.

This solution is not without problems. One of them, the implications of modifying the information system have already been discussed. Additionally the selection of Job Manager to register with may have consequences of how failsafe the system becomes. As an example, it is not a good idea to register with a Job Manager running on the same host as the registering Job Manager [3] but also to have the manager running at different geographical locations. This illustrates that the network topology is important, but it most likely not possible to ensure a geographical distribution automatically. Some simple heuristics taking domain names and ip-ranges into account could be developed, but the user should have the possibility of overriding the default behavior of the registration process.

If the Job Manager is to be used in a production grid, the information system should be changed to be able to describe the Job Manager. This is not possible for test purposes, but we have been able to place the contact information in the information system in a slightly odd way, as we will see in the following.

8.4.3 Job Manager Contact Information

This section describes two things: The contact information needed for a Job Manager, and how we will to use the information system for Job Manager contact information.

To contact a Job Manager the location, in the form of a hostname and a port number, is needed. This is the least amount of information needed but additional information to make selection process simpler should also be provided. This information includes, the user which is running the Job Manager and which Job Manager group it belongs to, see Section 8.5, and possibly a list of capabilities. Furthermore some additional information regarding multiple users and certificates may be necessary in the future. This includes, the users allowed and the certificate of the Job Manager.

[3] The idea of running two managers at the same host may seem like a bad idea, but there can be arguments for doing it, e.g., different capabilities to support special demands of some jobs.

If the Job Manager was to be regarded as the other resources in NorduGrid it would be described in an LDAP schema in the information system. It would serve as a part of the information system responding to queries and providing contact information to the information system. Since this is not possible without modifications, we push the information provider responsibility to the information provider on a random resource.

To do so, we exploit the fact that job information can be queried from the information system, until the jobs session directory is cleaned on the cluster. This is done automatically by the Grid Manager after a specified time, usually 24 hours. By having the Job Manager submit a "contact job" to the grid as one of the first actions it performs when it is started, the information abut this job is placed on the cluster to which the job was submitted and can be queried through the information system. The job itself does nothing, e.g., sleeps for one second and exits, but in the same way as job tags, see Section 7.2, the name carries the information needed to contact the Job Manager.

To register with a Job Manager a query is sent to the information system to locate contact jobs. Since job information can live for a long time, it is important that the contact jobs are time stamped in order to identify the newest of two or more identical jobs. Additionally the Job Manager should clean up its old contact information when possible. This means that old contact jobs should be cleaned when shutting down, and when reregistering after a crash, if possible.

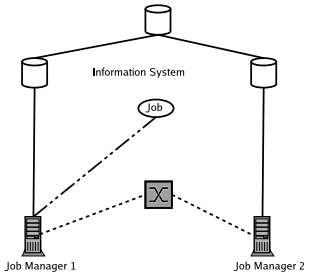
This method is only for testing purposes but should work, as long as the cluster holding the job information is still on the grid.

8.5 Model for Distributing the Job Manager

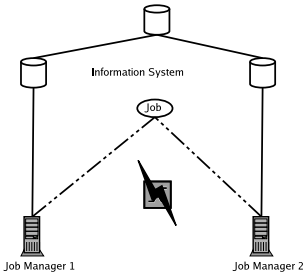
We now look at the specific model used for distributing the Job Manager. This model takes the previous discussion into consideration and it uses the information system as the basis for discovering other Job Managers. The Job Managers are grouped together and one manager in the group is designated as the master. The master accepts and handles all request. If the master should fail another Job Manager takes over the management of the jobs.

Without a reliable failure detector there is a risk of job duplication. As an example, consider the situation where a manager is suspected to have failed, but it has in fact been separated from some of the others due to a network partition or congestion preventing the registration message to reach the failover in due time. We disregard that a separation would likely also have separated some the Job Managers from the grid and the jobs being managed. A network failure can be both intransitive and non-commutative, causing an election to be called on the one side of the partitioning. This election would result in a new manager in charge of the jobs. There are two outcomes of this scenario, see Figure 8.3. Either the same job is being managed by two managers, which can see the job but not each other. The other situation is that a duplicate job is being spawned by the new manager, because it cannot locate the job, and it is thus determined to have failed.

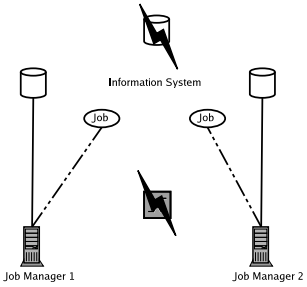
It may be unlikely, but possible that the Job Manager and resource could be separated from the other Job Managers. In order for this to happen there has to be a GIIS on every side of the network partition. An example could be to have a Job Manager running in Sweden and one in Denmark. If the connection between Denmark and Sweden breaks down, there is still a GIIS on both sides of the split. The jobs managed by the manager is duplicated on both sides of the partition. No matter what side the client runs on it still have access to the job, though one of them have been resubmitted.



(a) Before



(b) One job multiple managers



(c) Job duplicated

Figure 8.3: A network partitioning can have different results. (a) shows the initial situation, where the manages can soft register with each other. In (b) the route between them have disappeared, but the job can still be located by both. Manager 2 accidentally thinks manager 1 has crashed and begins to manage the job. In (c) the information is not available across the partition and results in the job being duplicated.

8.5.1 Election

If there are several managers, that can take over the management of the jobs belonging to the failed manager, a mechanism for determining which manager should be elected is needed. To determine how this should be done we look at standard algorithms for leader election in distributed system [17]. This is of cause dependent on the level of distribution. If the relationships between the Job Managers are one-to-one, then the method for eliminating duplicate jobs are pretty straight forward as the manager which took charge of the management just returns it to the owner on subsequent registrations. In this case the original owner determines if the job was duplicated and kills the duplicate that has run for the least amount of time.

We look at the “bully” algorithm, since it is fairly simple, it allows managers to crash during an election, and it can handle concurrent elections. The algorithm is described in detail in [17]. The algorithm requires that the processes knows how to communicate with all the other mangers. In addition the managers should have a total ordering and the managers know of this ordering. This ordering should be based on the Job Manager id.

An election algorithm should have two properties, which we will look at briefly [17]:

- **Safety** – All participating processes eventually agrees on electing the (non-crashed) process with the highest identifier.
- **Liveness** – All processes participate and eventually elects a process, or crash.

The algorithm is as follows, if a manager has determined that the master has failed it checks to see which manager has the highest identifier. The manager then sends an election messages to the managers with higher identifiers. It then waits for acknowledgments from the managers it has just contacted. If the manager receives no replies before a timeout it assumes that the managers with higher identifiers have failed. It then elects itself as the new master and sends a coordinator message to all managers with lower identifiers. If the manager already has the highest identifier, it sends a coordinator message to all the other Job Managers. This ensures the manager with the highest identifier gets elected and there are no ambiguity[4]. There is another problem though, the process cannot guarantee, the safety property, that only the manager with the highest identifier gets elected. If a new manager is started, it calls an election, if it has the highest identifier it becomes the master. The algorithm assumes the system is synchronous because it uses timeouts to detect manager failure, but it has a problem: If a manager is restarted with the same identifier as replacement for a crashed one before the end of the election, it may announce itself as the master, but since there are no ordering in the message delivery, the clients can have elected different masters. This problem is similar to the problem with network failure. Especially in the election process, where a manager is presumed to have failed if it does not reply to an election message.

However, the problems that may arise from failure may be doubly managed jobs or multiple identical jobs running on the grid. These problems will be handled by the master manger.

8.5.2 Master Manager

To counter the problem of job duplication we introduce the concept of a master manager. Every job has a master manager assigned to it. If the master manager fails, then job control is transferred to another as described above, but if the master subsequently registers again, job control is transferred back to the master chosen by election. This scheme ensures that there are only one manager[5] to make decisions regarding a job. The situation where the response to an election, did not reach a failover, causing a job to be managed by two managers, will remedy itself, when the master registers again. When this happens the failover must call an election to transfers control back to the master, which is then the sole manager of the job.

If the managers have been separated by a network partition causing a job to be duplicated, the duplicated jobs run concurrently until the network gets reconnected. This can actually be a good thing, since the client can be located on either side, and the duration of the network partition may be very long. Whenever the network partition disappears, control of the duplicate job is returned to the master – remember that duplicates can be identified since they have identical job tags. It is now the responsibility of the master manager to eliminate one of the jobs. The most straight forward heuristic is

[4] This is true in a synchronous system with known upper bounds for message round trip times.

[5] A part from the some failure conditions which can result in more than one manager. But we will deal with these later.

to cancel the job that have been running for the least amount of time, but other factors may be taken into account. Due to the complexity of this subject, the behavior of the Job Manager should be configurable.

Another problem is that a network partition may result in the submission of a lot of jobs. There is currently no way to prevent this, but heuristics to determine the type of failure may be a solution. However this is not something that time allows us to dig deeper into, as it is a project in itself[6].

The introduction of a master manager does not in itself solve all the problems associated with duplicate jobs. Imagine if the failover manager, managing a job on behalf of a master assumed to have failed. If this manager also fails, and another manager has to take over the management of the jobs.

The manages periodically polls the information system to see if there are new members of the group. Doing so would also make the failover discover if the original master reappears. If the master reappears the failover (and the master if it was restarted) calls an election, which should result in the master being elected. Since the information system is not changed, the failover has no way of determining if the contact information is old, or the master never failed. To account for this, the failover must periodically check if the old master still responds, and if so call an election, to ensure that there is agreement about which is the master.

Whenever a master is reelected, the failover should transfer the meta data back, and the master should check duplicates of jobs. If the master did not fail, it can safely discard the meta data, but if it was restarted the meta data should be considered valid. The process is illustrated on Figure 8.4. This design puts the responsibility on the master to handle cases of duplication. By putting it on one specific manager makes it easier to write handlers to specify what to do in different situations. For this scheme to work, it is important to ensure that the master cannot forget about jobs it have managed, this can be done by the session management[7].

As an alternative a list of managed jobs could be piggy backed on acknowledge messages sent to the failover managers. This requires a lot of unnecessary bandwidth since the list does not change as often as the manager registers.

The best solution is to make sure the manager does not “forget” about a job. This means that the master must save the settings before distributing the information about the job. This makes election of a new master the event that should trigger handling of duplicate jobs.

If a manager for some reason does not want to be the master of a job anymore, e.g., it is shutting down, it should be possible to transfer job control to another manager by pushing it onto one of its failover managers. To do so, the master calls an election, by sending an election message to the failover manager with the lowest id. The election runs as normal, but the original master does not respond to the election message, thereby ensuring that another manager gets elected.

8.5.3 Manager Groups

The idea of letting all Job Managers belonging to a user be member of one group is generally fine, but there are some scalability issues involved if the job data are to be

[6] Some experiments with benchmarking are being performed [73], and it may provide be a tool to develop better heuristics for determining which duplicate to kill.

[7] This is guaranteed even if the session is alway saved prior to submitting jobs. It is, however, reasonable to assume that if a Job Manager fails in a way that destroys the session, then it has also forgotten about its prior id, and will be considered a new manager by the others in the group.

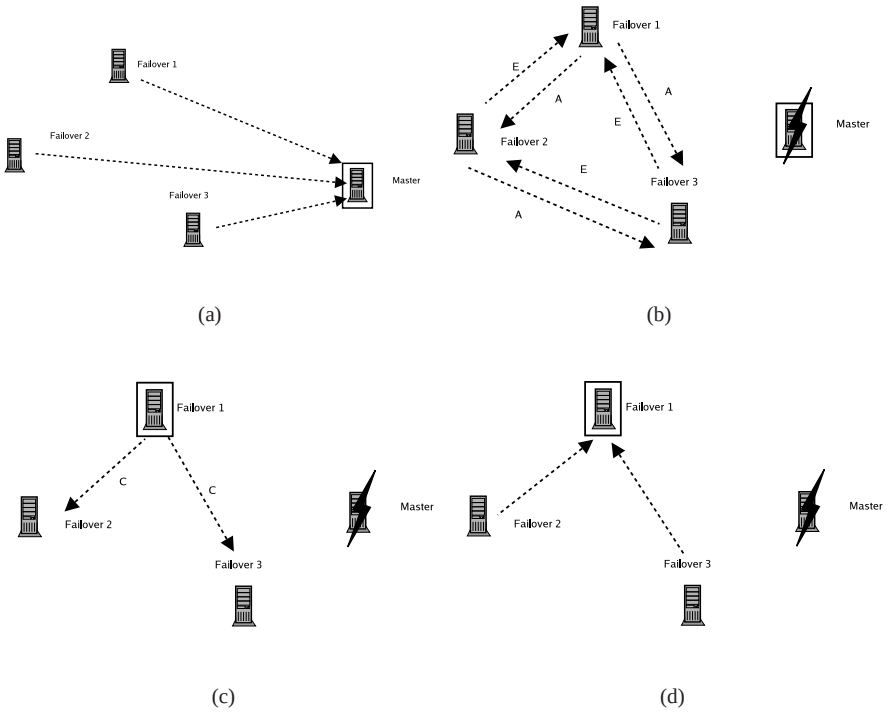


Figure 8.4: How failure is handled by the Job Managers in a group. The failover managers periodically checks if the master is responding (a). If it fails an election is called by the failover that detects it. The election is done by sending an election message to the managers with higher identifiers (b). If the manager receives an answer from a manager with a higher identifier it waits for a coordinator message (c). The manager with the highest identifier sends a coordination message to the managers in the group (d).

replicated on a lot of managers. This is not a problem if a user only starts a handful there should be no problem with the messages sent between the job managers.

Another purpose of groups is to provide a way of separating job managers from each others, making it possible for the user to start groups of managers with different capabilities, without disturbing each other.

8.5.4 Other Considerations

For this design to work, there is some requirements to the Job Manager contact information. This information includes the identifier of the Job Manager and the Job Manager group to which it belongs. The id is generated like a job tag, see Section 7.2. With a few differences; it is prefixed by three % signs, then the name of the Job Manager group, and lastly a hash to ensure different values for the identifier. The hash is used as the identifier on which the election is based. In order to be able to dictate the master, the hash is prefixed by a value. By using a high value as a prefix, it should be possible to generate the highest identifier in the group, thereby becoming the master.

```
jm-name%%%jm-group%prefix(hash)
```

If the master fails, the application should have some way to contact the failover manager. This means that information about failover managers must be propagated back to the application. This should be done the first time the client contacts the master manager. The contact information should be a list of possible managers to contact if the master fails. This list should be ordered according to the identifiers, so the application can query them in turn, to locate the one which is managing the job.

The communication protocol the managers use internally should also be based on the XML-RPC server, since it is already implemented in the Job Manager and there is no need to add to the complexity of the Job Manager.

8.6 Implementation

This section describes some of the implementation details of the Job Manager communication module. We start by examining the interface to other Job Managers.

- `alive()`
  Used to check if the Job Manager is alive. Returns 0[8].
- `update_status()`
  Called from the master to explicitly update the job data. If the job is not known by the callee, then the job is added to the list of jobs to manage if the master fails.
- `coordinator(identifier)`
  When this function is called the manager sets the current coordinator to this. If the identifier is lower than the managers own identifier, another election is called.
- `election(identifier)`
  When this function is called the manager sets `master = None` and calls `election` on managers in the group with higher identifiers. If all the calls to election times out, the managers are assumed to have failed. In this case the manager sets `master` to its own identifier, and calls `coordinator()` to managers with lower identifiers. When this is done `remove_duplicates()` are called.

The election is based on the timeouts in the XML_RPC, and this timeout is used to determine if other managers have failed.

Internally there are functions to perform the tasks of updating the information system and monitoring jobs.

- `initialize()`
  Calls the configuration and session modules and updates the internal settings.
- `submit_contact_job(target_list = None)`
  Called when the manager restarts. It creates a job description with name set to the identifier and submits it to the targets in `target_list`. If no targets are provided it submits a job to a random cluster.
- `remove_duplicates()`
  Called upon election this function queries all jobs belonging to the group and tests if there are jobs with identical job tags. If this is the case the one that has been running for the least amount of time.

[8] XML-RPC calls are not allowed to return the empty value.

- `start_election()`
  When this function is called. The manager starts an election by calling `election` on managers which has a higher identifier than itself.
- `locate_managers()`
  Queries the information system to for contact jobs. Once retrieved, the list is sorted to hold only managers from the same group as indicated by `jm_group`.
- `generate_id()`
  Generated an unique identifier, if master is set in the configuration it tries to generate a higher identifier than the managers running, by prefixing the hash with a large value.
- `job_info_update()`
  When this function is called, the Job Manager updates the job information if the job is known, if not, the the job is added.

The module has internal data structures used to keep track of the other managers.

- `managers` – Holds a list of manager objects ordered by identifiers.
- `jmgroup` – The group to which the manager belongs.
- `master` – Variable holding the identifier of the current master manager. If an election is in progress this variable is set to None.

8.7 Summary

This chapter displays the problems when working with asynchronously distributed systems. We have seen that, there are no way to prevent jobs from being duplicated or the loss of management due to byzantine failures. This is not a surprise since it is founded on well known results of research in distributed systems. However, the scheme we have proposed improve availability and if it is implemented correct makes it less probable that a failure will result in lost jobs, but may cause result in duplicated or multiple managed jobs. This is not as big a problem in the current NorduGrid, but in a grid with accounting, the risk of occasionally loosing a job, may be preferable to having the same job run multiple times. This scheme will be hard to enforce with the current algorithm since it does not guarantee that only one master is elected.

From this chapter it is clear that the information system should be extended to support the Job Manager. Furthermore if the Job Manager becomes a resource like other resources in NorduGrid it is possible to use the information system for failure detection, since a soft registration mechanism is already implemented here. In this case the Job Managers would check if its failover managers are still alive by sending a query to the information system. If the contact information is present, the Job Manager has checked in recently and can be considered to be working.

This does not have much impact on the implementation, except changing the registration mechanism to register with the information system, and implement an additional check for live managers against the information system. This can be combined with the discovery of new managers without much trouble.

Chapter 9

Future Work

This chapter presents ideas and suggestions for improving the Job Manager further. These ideas where either cut out due to time constraints, arrived later in the developement process or was not considered to be a part of the Job Manager when creating it, leaving it out. The ideas presented in this chapter is: Using a database for job information, the extensions to NorduGrid ARC to support the Job Manager, multi user Job Manager and finally meta jobs.

Currently the Job Manager serializes the job list to disk at regular intervals. This is done as a form of check pointing, making it possible for the Job Manager to recreate its former state after a crash or restart and recover gracefully. Unfortunately such a serialization can become invalid if the Job Manager crashes while performing it. Instead of using this technique to store job data, a database could be used. Most database systems has what is called ACID properties [65]. The A in ACID stands for atomicity, which means that something is either updated or not, no state exists between the two. Having this property when writing the session it would always be consistent, ensuring that the Job Manager would always return to a consistent state. This state would be relatively new since the job descriptions are kept in the database, and not just serialized at a certain interval. Using a fully fledged database system will make the Job Manager quite heavy, so the database system should be light weight. An example of such is the Zope Object Database [16], which is written in Python, meaning that no wrappers are necessary, making it a good choice.

As mentioned several times in the report, it would be nice to extend the xRSL language and the information system to have support for the Job Manager. For the xRSL language, this would be an extension that makes it possible to carry meta data with the job, such as previous clusters and submissions attempts. The information system should also make this information available; perhaps only through authenticated queries, since it is not all users that wants job meta data exposed. The information should also have support for tags, as a way to identify instances of the same job. Furthermore the information system should support the Job Manager as a resource, making it possible to query the information directly for Job Managers, instead of the current method, which submits “contact jobs”, for the other Job Managers to query after. This solutions is hack. For any of these changes to happen, the Job Manager would first have to prove it worth, becoming an integrated part of the NorduGrid ARC, proving its worth. Even if this would happen it is unlikely that such support would come, unless a part of the NorduGrid ARC would be redesigned.

It was mentioned in Section 4.3.4 that it was not possible for the Job Manager to

support multi user due to constraints in Globus and the NorduGrid ARC. The limitations in NorduGrid ARC are mostly due to the constraints inherited from Globus. If the NorduGrid ARC was to be leveraged from the Globus dependency, it should be possible to make the Job Manager work for multiple users. However this would remove the ability for the users to make their own handler, since would jeopardize the stability of the Job Manager. However it could be imagined that both single user and multi user Job Managers could co exits, providing the best of both worlds.

Finally the idea of meta jobs will be presented. The Job Manager already makes it possible to submit jobs and monitor jobs. If any of these jobs has dependencies on each other, e.g., some jobs need the output files of other jobs as input files, the user or application will have to coordinate this. Meta jobs makes it possible to describe dependencies between jobs. To make meta jobs possible a new language which makes it possible describing dependencies, and splitting a job into parts automatically would have to be created. Devising such a language is a major task, and it is not clear for us how useful this feature is. However Job Managers and meta jobs are a perfect match since the a Job Manager can continuously monitor jobs and react to them; relieving the user or application to do this themselves. If this idea is to be realized it should first be investigated whether or not there is a need for this, since creating such a system is not simple task.

Besides the specific ideas presented, it is possible that new requirements may arise if the Job Manager is used. We hope that the Job Manager can take place in a production system, where it is used to monitor jobs. If this is the case new requirements will certainly arise, but we hope that the handler concept will make it easy to extend the way jobs will be handled.

Chapter 10

Conclusion

In this project we have constructed an advanced flexible job management system. To create this system we drew from the experience gained when creating NG Proxy. The main experience from NG Proxy was that it was possible to automate tasks by providing a daemon. However NG Proxy was very inflexible, and was not geared to fulfill the wishes of feature request. Ranging from migration of jobs to automatic data management. Another problem with NG Proxy was its interface, which was extremely inflexible. Furthermore NG Proxy was a single point of failure, a feature unwanted in production systems.

With these issues in mind we designed the Job Manager, to be more flexible than NG Proxy. Furthermore we wanted to create a way for application to use the grid, since the existing command line interface is not suited toward this. We decided that the Job Manager should feature a protocol as its interface. To cope with feature request we designed the Job Manager to be extensible. Therefore a plug-in structure was created, making it possible for users to extend the functionality themselves. Finally the possibility of having a failover Job Manager was introduced; avoiding the single point of failure existing in NG Proxy.

Part of a production system is to automate trivial tasks, but also to monitor the grid and react to changes. By making the Job Manager aware of the state of the grid by collecting information it is possible to make the Job Manager react autonomously.

10.1 Achieving the Goals

One of the main purposes of the Job Manager was to provide a simple, yet powerful interface to the NorduGrid ARC. The reason for this was to make it more simple for applications to interact with the grid. We wanted to make something complex simple, to attract users to the grid.

Making the functions available to applications could be done in two ways, either through an API or through a protocol to a separate application. We opted for the second since it makes it possible to move general grid logic away from the application. By having the Job Manager deliver an API through a standard protocol, XML-RPC, applications can use the Job Manager as an interface to grid, reducing the amount of grid code in application.

Also, the Job Manager can keep running when the application is shut down and in addition different applications can be used to connect to the same Job Manager, e.g.,

an application, a web portal, or a lightweight client for a portable device.

All these features makes the Job Manager a platform for application development and delivers a simpler view of the grid to the application, making it simpler to develop applications for the grid.

The Job Manager provides flexibility through handlers, which enables the user to redefine what actions should be done to a job when a state changes. This makes it possible to specialize the Job Manager to different types of jobs and applications, e.g., requiring only resubmissions in special cases. This gives new possibilities to users, since they can tailor the Job Manager to fit any special needs.

In a highly distributed system as grid it is important that there are no single points of failure that can bring the system down. We have extended this philosophy to also include the Job Manager. In order to make it resilient to failures it is possible start several managers and having them work as failover managers.

This introduces some complexity into the design of the Job Manager but it has been necessary to provide resistance to failure. We chose primary-backup solution because of its simplicity, and because we believe that it is enough to support the needs of most users. There are scalability issues, but due to the expected usage we do not foresee this to be a problem, i.e., most users will only have one or two managers running,

To make the distribution work we have had to make some “hacks” regarding discovery of job managers. This have made the solution more complicated than if the necessary information had been in the information system. It is necessary to make the Job Manager integrate better into the NorduGrid ARC if it is to be used for production. However we feel that the current solution is adequate to gain some first real life experience with the Job Manager.

There are still some problems distributing certain information like the job description, and meta data. These data are distributed directly between the Job Manager, but this may not be preferable in the long run and another way to support this information may be a good idea.

10.2 Extending the NorduGrid ARC

During the design of the Job Manager it became clear that the NorduGrid ARC had to be extended in order to make the Job Manager integrate seamlessly into the toolkit. However instead of rewriting NorduGrid ARC components, we focused on making the Job Manager work. This has led to ad hoc solutions in some areas where we decided not to change NorduGrid ARC components. This was done because we felt that the Job Manager should prove its worth, before making changes to the NorduGrid ARC. If so, the developers would have an incentive for integrating it into the toolkit.

There was parts of the NorduGrid ARC that it was necessary to modify in order for to create the Job Manager. This was mainly minor changes, e.g., adding empty constructors to classes, in order to be able to create wrappers properly. It should not pose a problem to integrate these changes into the NorduGrid ARC.

During the development of the Job Manager, several shortcomings where found in the NorduGrid ARC with respect to the desired functionality of the Job Manager. One of them is that there are no way of determining if two running jobs are different instances of the same job, which the Job Manager need. In order to provide the needed functionality this had to be possible. Therefore we introduced the concept of job tags to accommodate this. Tags expand the job concept in NorduGrid, by making it possible to track a job through several executions from the same job description, since tags

are persistent between submissions. This stands in contrast to the single instance job concept in the current NorduGrid architecture. Tags also make the communication between the Job Manager and the application more consistent, since one identifier can be used even though the job gets moved or resubmitted.

Another limitation was that it is not possible to introduce new types of resources to the grid. This became a problem since the Job Manager should be regarded as a resource, and need to be discoverable. We came up with a solution of submitting contact jobs. This is not the right solution, but it is enough for testing the Job Manager.

If the Job Manager is to be used in a production system, the changes proposed should be incorporated into the NorduGrid ARC, but this decision is up to the users and the developers. For the time being the Job Manager will work as it is supposed to though.

Other changes to the NorduGrid ARC, is that it should be considered to phase out the Globus dependency, even though some components rely heavily on it. The information system might as well be based on standard OpenLDAP and the authentication could be achieved through other means. Removing the Globus dependency would be the first step toward supporting multiple users at a single Job Manager.

10.3 Caching

The Job Manager delivers a simple caching functionality, allowing it to reuse former queries. The need arose since no caching is done in NorduGrid. Since the Job Manager can potentially monitor and manipulate large sets of jobs, this might have lead to Job Manager using large amount of time querying the grid. Therefore we decided to cache information into the Job Manager.

Caching is by some regarded as a problem, but in a large distributed system caching is almost necessary for it to be scalable. The Job Manager does not use the valid-from and valid-from fields in the information system, since these are hard obtain in the present system. This, however, is not a problem the Job Manager only uses the information for a short while, before throwing it away. Also this fits along with what we believe is the most common use pattern for information: A lot is needed over short amounts of time, for then not be needed for a large amount of time.

The the interface to NorduGrid Information System and the cache system in the Job Manager, may be to simple. The problem is that the module has no way of knowing what information the caller needs. This makes the modules fetch extensive amounts of information, which may not be necessary.

10.4 Language Choice

As described we decided to implement the Job Manager in Python. We have been very satisfied with this choice as development in this language is very rapid. Python supplies a great module collection, relieving the developer of implementing basic functionality and focusing on getting the application to work. It is our belief that such high level languages offers more compared to low level languages. If speed is a concern, one can often implement a part of the program in a low level program, create bindings and use a high level language for the rest.

Some of functionality in NorduGrid ARC has been reimplemented in Python. This has mostly been the monolithic functions, which where necessary to reimplement to

get the desired functionality. This was refactored into more modularized code, making it possible to extend certain parts of the NorduGrid codebase. Some of the code developed for the Job Manager can be used to replace some of the existing NorduGrid code since it implements the same functionality, but in a more modularized way, e.g., pluggable schedulers.

10.5 In Conclusion

The purpose of the Job Manager has been twofold. Firstly the Job Manager addresses the need for an automated production system, delivering needed functionality to the users. We have addressed part of the need for a production system and the Job Manager delivers a framework, or a platform, for a production system. Secondly it addresses another issue concerning the difficulties of developing applications by making a cleaner interface working over a standard protocol. We have made the interfacing with the NorduGrid ARC more accessible and made it possible to develop applications on a multitude of platforms.

By providing a simple interface to the grid, the Job Manager delivers a host of new possibilities for the usage of the grid. If this is enough to “lure” users and applications to the grid remains to be seen, but hopefully it will attract more applications and users, as grid technology offers new possibilities in several areas of computing.

Appendix A

Proposal for a new User Interface in the NorduGrid Toolkit

A.1 Introduction

This paper presents a proposal for a new design for a user interface and job manager for the NorduGrid Toolkit. It presents the goals and requirements of the new user interface, describes what capabilities it should provide, and what advantages it gives compared to the existing user interface. We start by describing the previous user interface in the NorduGrid Toolkit. Its features and short comings are described, explaining what it does and why a new architecture is required.

Hereafter the new design is presented. First an overall description is given, where after each logic part of the design is explained. An important part of the design is that it can (and must) co-exist with the existing NorduGrid command line tools. This criteria was important during the design, and the design can be considered both a new design or an extension to the existing. Finally a road map for the implementation is presented.

For those not familiar with the job manager[1]. The job manager is a daemon, running on the client side, monitoring jobs and acting on changes on their state. In its current functionality it is able to resubmit jobs if the fail.

A.2 The Existing User Interface

The existing user interface in the NorduGrid Toolkit is a group of command line tools. A brief overview of these tools are given in appendix B. These tools are invoked by the user, to submit jobs, query the status of existing jobs etc. It is not possible to automatize actions like fetching output files after job completion. This is due to the commands only acts when explicitly told to do so by the user. The NorduGrid Job Manager has delivered proof-of-concept that an automatized reactions on events are possible, making the client side in NorduGrid more aware of the state of the grid and acting on changes in the job status.

[1] Formerly NG Proxy

Also no sane API for submitting jobs and querying information about them are available for applications, making it hard for applications to reuse parts of the existing user interface. This makes development of applications which wishes to actively integrate with NorduGrid somewhat hard.

All these issues and examples of improvements makes it desirable to develop a new user interface that facilitates a more feature rich interaction between the user (application) and the grid.

A.3 Goals and Requirements

The section describes the motivation for creating a new user interface. These reasons can be expressed in the term of goals for a new user interface and are summarized below.

- **Flexible job control** - The user interface should support a more fine grained control over the different aspects of job control (e.g., scheduling, submission, resubmission, and data management)
- **Application interface** - It should be possible for applications to access and use the functionality provided by the user interface and thereby interact more closely with the grid.

These goals give rise to several requirements which the new user interface must meet in order to achieve the goals just described. These requirements are summarized below:

- **Modular User Interface** - To make it possible to extend, or add parts of the underlying functionality, the current user interface must be made more modular, providing easier access and more flexibility from a developer point of view. This modularity will, e.g., allow different schedulers to co-exist.
- **Providing an API/Protocol to applications** Currently the primary way of submitting jobs is to use the command line tool `ngsub`. This makes it hard for applications to use the user interface. Giving application an API or a protocol which they can use to communicate will make it easier for applications to use the NorduGrid Toolkit.
- **Back wards compatibility** - In order for a new user interface to be used, the existing command line user interfaces, i.e., ngsub and friends must continue to work as they did before, while not imposing any new requirements on their users.
- **Job Manager** - Integrating the Job Manager into the user interface, giving the user a way to supervise jobs and fetching output files automatically etc.
- **Handlers** - The user interface should support the possibility of extending or replacing existing functionality in the job manager by creating handlers for different aspects of job control and using these as modules supplying the user interface with new or extended functionality.

To meet these requirements we must create a new design. This design will be discussed in the next section, along with the implications it may have on the existing NorduGrid implementation.

A.4 New User Interface

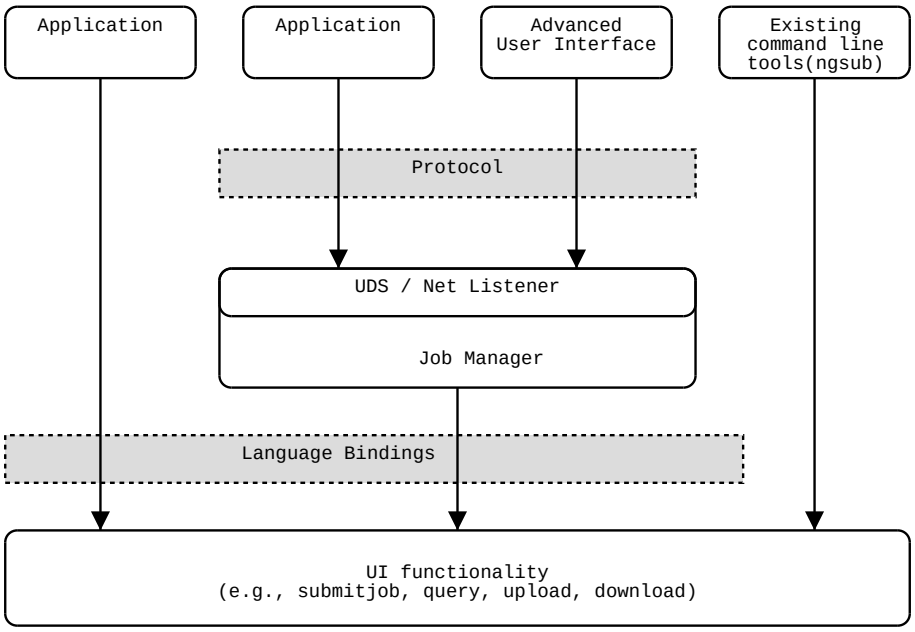


Figure A.1: *Overview of the design.*

On figure A.1 an overview over the new design is presented. The lower layer corresponds to the current functionality in the existing user interface, made modular. This makes it possible to extend or add parts of the user interface more easily, while also making a more fine grained API. Such a change will give applications and the job manager more flexibility, since it can use more fine grained functions, and not monolithic functions spanning several hundred lines of code.

Above the UI functionality sits the existing command line tools. These will use the functionality provided by the lower layer. To access the underlying functionality language bindings will be created, making the functionality accessible for applications using a high level language. These bindings should be accessible from several languages, meaning that generated bindings should be preferred.

A special application will be the job manager, which will be a continuation of the existing job manager.The job manager will be probably be rewritten in a high level language like python, using the language bindings just described. The job manager should have support for various handlers, that is, reacting on certain events, like job completion or failure, making it possible to collect output files or resubmit jobs automatically. We aim to make an interface for implementing such handlers, so it will be possible to write new handlers (especially since people have a lot of ides for use of the job manager) and choose between the existing ones.

The job manager will listen for commands, either on a Unix domain socket and on a net listener, enabling applications to use the functionality of the job manager. One of the major problems with the current job manager is that there exist no real protocol between it and its applications. It is clean that an existing protocol should be used. The protocol should support the needed functionelity and be simple to use and program

against. Requirements for the protocol should be examined as well as a way of doing authentication.

The first application to use the job manager should be a new command line user interface, which will have the functionality of current user interface, but will also be able to use the functionality provided the job manager. Other applications building on top of the job manager could be a web interface / portal. One could also imagine a WRSF job submission service to run on top of the job manager.

A.4.1 Issues with the design

So far, the design is very general and a lot of details has not been set in stone. A list of some of the most critical issues are listed.

- **Interfaces** A lot of interfaces for, e.g., handlers and schedulers has not yet been defined. This interfaces are important if it is to become possible to plug-in new schedulers and handlers. Must be flexible and relatively easy to code against. Feedback on this is very welcome.
- **Protocol** We have not decided on a protocol yet, but it will probably come down to XML-RPC or web services. Other options are CORBA and Jabber. CORBA seems like overkill though. The protocol must support authentication with X.509 certificates. Jabber does not appear to support this. Discovery of capabilities should also be supported (XML-RPC and web services support this.
- **Remote use of Job Manager** A lot of issues arise when using the job manager remotely. E.g. should upload of files go through the job manager or happen directly from the submitter to the cluster.

A.5 Road map

The new user interface build on the existing user interface in the toolkit. The user interface currently contains almost 12.000 lines of code; throwing this out, and starting all over is a daunting task. Instead we aim to change the user interface into something that meets the requirements, using an evolutionary approach refactoring the components over time while keeping the old tools and maintaining a certain level of back wards compatibility.

1. Design (your reading it). Determine the structure of the new user interface, how it interacts with other components. Locating critical points in the design.
2. Post the document to the nordugrid-discuss mailing list. Hopefully get some constructive feedback, second opinions and flames :-).
3. Modularize the functionality of existing the user interface, thereby creating (and determining) an API for the user interface to export.
4. Create bindings for the user interface, making it possible to use the functionality from other languages.
5. Determine the protocol for the job manager to use.
6. Implement job manager with lots of new features and options. New job manager will be written in python (most of the new functionality will).

Appendix B

The NorduGrid Command Line Interface

The command line interface in NorduGrid.

- **ngacl** Get and set access rules for remote files on gridftp servers with gacl support.
- **ngcat** Shows output of job
- **ngclean** Cleans up after a job by removing files on the cluster.
- **ngcopy** Copy from URL to URL.
- **ngget** Downloads output files of a job
- **ngkill** Kills a job on a computing element.
- **ngls** Lists contents and attributes of objects on a rc.
- **ngremove** Remove file at URL (replica catalog).
- **ngrenew** Renews proxy certificates of jobs
- **ngresub** Resubmits a job.
- **ngstat** Obtains status of jobs
- **ngsub** Submits jobs
- **ngsync** Synchronizes local job list with global list.

Appendix C

Generating SWIG Wrappers

This appendix describe the creation of Python language bindings to the underlying NorduGrid ARC code. This appendix describes how we have used the tool Swig to create bindings to this.

Basically there are two ways to create binding to another language; either writing them by hand, or using a tool to automatically generate them. Generally the creation of bindings require a lot of code, but it is rather simple and can therefor easily be automatically generated. Since we have no special needs that require that we write the bindings our selves, we have chosen to generate the necessary code.

There are several tools which can be used for automatic creation of bindings; one of the most popular are the Simplified Wrapper and Interface Generator (Swig)[1]. It is a popular tool for creating C/C++ language bindings to Python [2] which is one of the reasons for choosing it to create bindings to the NorduGrid and Globus code.

C.1 Using Swig

When using Swig there are two ways of generating bindings to functions classes and their respective methods, written in C++ file. Either by supplying it with the header file letting it generate bindings to everything specified in the header file. This approach may not, as in our case, always be preferable. Either because we do not want bindings to everything, or because there are some statements in the header file which Swig cannot handle. Swig have no problem with most of C with some few exceptions concerning the more esoteric features of the language. It does not behave so well when it comes to the features of C++. These features include namespaces, functions passed as arguments.

The other way to create bindings is to create an interface file, written in a language resembling C, and include the functions, classes, methods, and typedefs to which bindings should be created. The functions that should be wrapped are defined in the same way as a prototype i a normal C header file, and the interface file are feed to Swig to create bindings. This is also the method used for wrapping classes, in order for objects to be passed as parameters between the languages. Swig generates a C file to compile and a Python file to include in programs that makes use of the bindings. An example of and interface file is shown below.

[1] http://www.swig.org

[2] Swig can in fact be used to create binding to several languages including; C#, Guile, Java, MzScheme, OCAML, Python, Perl, Ruby, Tcl/Tk, and Tcl

```
%module ngbindings

%{

#include "DateTime.h"
#include "MdsQuery.h"
#include "CertInfo.h"
#include "LdapQuery.h"
#include <globus_common.h>
#include <globus_rsl.h>
#include "MdsQuery.h"
#include "Target.h"
#include "Xrsl.h"
#include "time.h"
#include "Preferences.h"

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include "Environment.h"

#include "wrappers.h"

%}

%include "std_string.i"
%include "std_vector.i"
%include "std_map.i"

%include "wrappers.h"

%template(string_vector) std::vector<std::string>;
%template(giis_vector) std::vector< Giis >;
%template(job_vector) std::vector< Job >;
%template(cluster_vector) std::vector< Cluster >;
%template(queue_vector) std::vector < Queue >;
%template(target_vector) std::vector < Target >;
```

In the interface file, there are two types of includes. The `#include` directive in the beginning of the interface file is copied directly to the generated wrapper file and works in the same way as a normal include. This ensures that it is possible to compile the generated wrapper file. The `%include` directive is used to include other Swig interface files. In our case we use the interface files for the Standard Template Library, and the file wrapper.h, which contains the classes that needs to be wrapped.

There are things to pay attention to when it comes to using Swig. When it comes to pointers, Swig will wrap most pointers in C correctly. A pointer is encoded as an address and type information and it is important to note that this representation cannot be dereferenced in the target language. This may sound odd, but it enables the target language to pass pointers to other wrapped functions.

Another thing to note is that Swig does not handle composite C++ types natively. To handle these types, Swig works with a concept called typemaps, which is code that are wrapped around the C++ code making it possible to create bindings to the types and use them in the target language. Swig 1.3 delivers typemaps to most of the Standard Template Library (STL)[3]. This somewhat solves the problem of dealing with strings in C (char pointers) as we can use the C++ STL equivalent instead.

When using container classes they too must be defined in order to access the elements contained in such a class. This is done by defining templates in the interface

[3]The Standard Template Library, is a library of standard templates and types.

file and defining the types for the functions in the interface file. To create bindings to STL types containing other STL types e.g., a vector of strings, a template should be defined in the interface file, in order to get access to the elements of the type (vector) from within Python. If no template is created the type cannot be accessed from Python, but it can still be passed, though Python in the same way as a pointer, as a parameter to other methods written in C++. An example of this could be a binding to a method returning a vector of objects, this vector may not be accessible in Python in a meaningful way, but can be passed on to another method written in C++ with the desired result.

The use of typemaps and templates are important when working with the NorduGrid code since it is primarily written in C++ and makes use of STL types, pointers, and objects as parameters and return types of the functions. There are however some problems that Swig cannot handle satisfactory for our use and in order to counter this problem some wrapper functions are created.

C.2 Wrapper Functions

The wrapper functions are functions that does not add new functionality to the interface, but it merely "wraps" the functions into a method or function that can more easily be wrapped by Swig.

To counter these as well as problems with pointer to pointer references and problems with references as parameters to some of the methods in the NorduGrid code base, it has been necessary to write wrapper functions to some of the methods in order to make some of the bindings. The wrapper functions does not add functionality or features to the methods, but serves only as an other interface to the methods in question. Below is an example of a wrapper function. Notice the conversion between C++ and C strings.

```
/* Wrapper for ../grid-manager/ui_uploader.h */

string ui_upload(string resource,
                 string rsl,
                 string session_url,
                 string job_id,
                 int act,
                 vector<string> filenames,
                 int debug,
                 int timeout) {

        char* jobid;
        strcpy(jobid,job_id.c_str());

        int result = ui_uploader(resource.c_str(), rsl.c_str(),
                                 &jobid, session_url.c_str(),
                                 (rsl_action) act, filenames,
                                 debug, timeout);
        if (result == 0) {
                return string(jobid);
        }
        else {
                stringstream ss;
                ss << result;
                return ss.str();
        }
}
```

This example also demonstrates, that often it is necessary to access a value returned as pointer, or to pass a value as an argument. And we have to dereference or reference

the pointers to access the values when passing them between the languages.

One of the problems handled in this way is the passing of functions as a parameter as well as problems with C++ namespaces. But these problems aside it is generally fairly easy to use Swig once you have the hang of it.

Appendix D

NorduGrid Wrapper Interface

This appendix describes the NorduGrid ARC interface wrapped to Python. This interface serves a basis for the functionality of the Job Manager. Many of the functions and classes in NorduGrid ARC and C++ has been directly wrapped to Python. However it was necessary to write wrappers to some of the functions and methods, mostly due to the inability to pass pointers to C++ from Python. The interface reminds much of the one in NorduGrid ARC, and will therefore not be described extensively. This also means that methods are not described, since this would take up an extensive amount of space. The wrapped interface is listed below:

- **Shadow classes –** Shadow classes are C++ classes wrapped to Python, making it possible to instantiate objects from C++ classes from within Python. This makes it easy to operate with C++ classes in Python. The wrapped classes are:
  - `CertInfo` Represent a proxy certificate.
  - `Giis` Represents a GIIS server.
  - `Cluster` Contains information about a cluster. Also contains the queue objects of the cluster.
  - `Queue` Contains information about a queue. Also contains job objects for the queue.
  - `Job` Contains information about a job.
  - `Target` Represent a target, i.e., a cluster and queue.
  - `Xrsl` Represent a job description.
- **Functions –** The functions are the ones wrapped directly from the NorduGrid ARC without any separate wrapper functions between.
  - `ActivateGlobus` Acticates the Globus modules, making the NorduGrid ARC and Globus ready to use.
  - `DeactivateGlobus` Deactivate the Globus modules.
  - `TimeStamp` Returns a timestamp.
- **Wrapped Functions –** These functions are wrapped functions which call similar functions in NorduGrid ARC. These functions are created to avoid passing pointers from Python into C++. Furthermore there are some problems with C++ namespaces that must avoided as well.

  - `ngFindClusterInfo` Wrapper for FindClusterInfo.
  - `ngFindClusters` Wrapper for FindClusters
  - `ngGetGiises` Wrapper for GetGiises
  - `ui_download` Wrapper for ui_downloader.
  - `ui_upload` Wrapper for ui_uploader.
  - `ui_upload_cancel_job` Specific wrapper for ui_uploader when using it to cancel jobs.
  - `ui_upload_clean_job` Specific wrapper for ui_uploader when using it clean jobs.
  - `ui_upload_submit_job` Specific wrapper for ui_uploader when using it submit jobs

- **Templates –** These classes are C++ STL classes wrapped to Python, making it possible to create these for wrapped functions and methods that need these types as argument. The only difference between these and shadow classes are these classes are provided by the STL library (even though they may be containers for NorduGrid ARC classes).

  - `cluster_vector` A Vector of Clusters.
  - `giis_vector` A Vector of GIISes.
  - `job_vector` A Vector of Jobs.
  - `queue_vector` A Vector of Queues.
  - `string_vector` A Vector of Strings.
  - `target_vector` A Vector of Targets.

These wrappers provides an API which makes it possible to support the needed features for the Job Manager. All the interfaces describes are wrapped using SWIG [25], which use was explained in Appendix C

Appendix E

Application Protocol

This chapter defines the protocol through which the applications communicates with the Job Manager. The primary source for determining the protocol can be found by looking at the NorduGrid command line interface. Also, a lot of requirements emerged throughout the report, in order to control different aspects of the Job Manager. Since the application interface is the only way to communicate with the Job Manager (apart from sending signals to it) the interface must be rich enough to support every type of interaction with the manager.

The functions can be divided into three categories, similar to the three pillars of the Globus Toolkit version 2, resource management, information services, and data management [2]. All of the calls will raise an XML-RPC fault type if used inproperly, e.g., if using a tag that does not exist. For brevity it is not described when these are raised. However some contraints will be mentioned.

E.1 Resource Management

This section presents the functions that have to do with resource management. These are primarily functions that do with job control:

- `buildJob(executable, arguments = “”, input_files = None, output_files = None)`
  Builds an xRSL job description for the application. Since xRSL supports a multitude of options only the simplest attributes are supported. Returns a string containing xRSL job description.
- `submitJob(xrsl_string, input_files = None)`
  Submits a job, provided a proper xrsl description. If the job has any local input files, they must be passed along in an array. A job tag is returned.
- `cancelJob(tag)`
  Cancels a job given its tag. The job must be running to cancel it.
- `cleanJob(tag)`
  Cleans a job given its tag. The job must finished to clean it.
- `getJobStatus(tag)`
  Returns the status of a job given its tag.

- `getJobId(tag)`
  Given a tag, the jobid (if submitted) of the job instance is returned

The Job Manager can also be regarded as resource on the grid and should therefore be controllable. The functions for this is placed here.

- `listManagers()`
  Returns a list of structs/tuples each containing a hostname and a port number of the other Job Managers, that the Job Manager knows.
- `getFailOverJobManagers()`
  Returns a list of structs/types containing the hostnames and ports numbers of the fail over managmers.
- `getMaster()`
  Returns contact information about the master of the Job Manager.
- `isMaster()`
  Returns true if the Job Manager is master, false otherwise.
- `registerManager((hosname, port))`
  Provides the Job Manager with a contact string to another Job Manager.
- `handoverJobs((job_manager, port) = None`
  Hands over the jobs to another Job Manager. A Job Manager can be specified, if not the Job Manager chooses one of itself.
- `shutdown()`
  Makes the Job Manager shutdown. It does not handover jobs. This must be done with the `handoverJobs()` call first if this is desired.
- `restart()`
  Restarts the Job Manager. Session will be saved before restart.
- `renewProxy(proxy)`
  Sends a new proxy certificate to the Job Manager, overwriting its previous proxy certificate.
- `destroyProxy()`
  Makes the Job Manager destroy, i.e., delete its proxy certificate. This will cut if off from the rest of the grid, including the client, since it no longer can be veried that it runs on behalf of the user.

E.2 Information Services

Functions in information services are pertaining to getting information about the grid, but also getting information about the Job Manager. The information service functions are:

- `getJobList()`
  Returns a list tags representing the jobs in the Job Manager
- `clearCache()`
  Clears the information system cache in the Job Manager.

- `getGlobalJobList()`
  Get a list of all the jobs of the user running on the grid.

E.3 Data Management

The functions in data managament concerns handling data on the grid. The functions are mostly related to the movement of files, but also functionality like setting access control lists are supported.

- `downloadFile(url)`
  Downloads a file to the client, using the Job Manager as Proxy.
- `uploadFile(file, url)`
  Uploads a file to the grid, using the Job Manager as proxy.
- `copyFile(from_url, to_url, blocking = True)`
  Copies a file from one url to antoher. If blocking is set the call will first return when the copying is done; if set to false the call will return imediately, making the Job Manager act as copying service.
- `deleteFile(url)`
  Deletes a file on the grid.
- `getOutput(tag)`
  Returns standard output of a job, given its tag.
- `getAcl(url)`
  Returns an access control list from the given url.
- `setAcl(url, acl)`
  Sets an access control list for the url.

Appendix F

Analysis of deadlock when using Globus concurrently

During the development of the Job Manager, it was discovered that under certain circomstances the function `ui_uploader` in NorduGrid ARC would block, i.e., wait forever and not return. Since `ui_uploader` handles job submission, cancelation, cleaning and renewal of remote proxy certificates, the function must always work; the Job Manager is highly dependent on it. Investigating the problem deeper, lead to the discovery that `ui_uploader` would never block the first time, but usually the second time it was called, and sometimes the third time. Since the problem was not deterministic, we believed that the cause of the blocking was a race condition leading to a deadlock, making `ui_uploader` block.

A race condition is an unwanted behaviour in concurrent system where the actions of the threads of processes must happen in a specific order, but this order is not enforced by the synchronization mechanisms [20]. The result of a race condition us usually that data can be modified by two or more threads (or processes) at the same time. Race conditions can be notoriously hard to find and debug for since they often require very special circumstances to trigger. A deadlock is when one or more threads each have acquired one or more locks, and are waiting to acquire another lock, which is held by another thread (this thread may be itself). This causes the waiting thread to sleep forever, since they are waiting for each other [19]. Strictly speaking a deadlock is not a race condition, however when a deadlock cannot be provoked deterministically it has the same charistica of a race condition, and if the data that must be protected by the synchronization mechanisms is a set of locks, but these locks are not properly protected, they can be subject to a race condition, which can lead to a deadlock.

Our initial thoughts concerning the deadlock was that `ui_uploader` could be triggered to not cleaning up properly after it. However this was not consistent with the fact that `ngsub` was able to submit several jobs when calling `ui_upload` iteratively. Investigating the problem further is was discovered that the problem only existed when using the SOAP server delivered with pyGlobus for RPC calls. When disabling the RPC server and using the API of the Job Manager directly everything worked fine. This turned the attention towards pyGlobus instead. However the pyGlobus library is used by the Access Grid Project[55], to create their grid middleware. Also we had used functions in Globus through pyGlobus which worked fine. However when recieving an RPC call a certain program technique is used heavily: Callback. Callback is method in

which function as passed along to other functions, to be called later. Callbacks are common for event notification, authorization or connection handling in servers. The last is used in the SOAP server in pyGlobus, which builds upon a GSI enabled TCP server in Globus. When this server recives an incoming connection, a handler in pyGlobus is called, which again call a function in the Job Manager, which is calls `ui_uploader` in NorduGrid ARC, which again calls Globus. This is vhere the deadlock happens. On Figure F.1 an example of regular calls from the Job Manager to pyGlobus and NorduGrid ARC, and down to Globus is depicted. When calls happen this way, everything works fine. On Figure F.2 the scheme where the deadlock happens is depicted. Here a callback from Globus results in a new library call to Globus. It is our belief that this is what is causing the deadlock, i.e., the cause is an error in the Globus library.

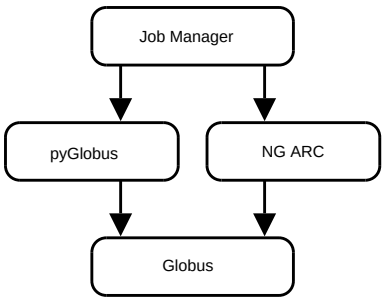


Figure F.1: Noramlly the Job Manager make calls to globus through pyGlobus and the NorduGrid ARC. The scheme works fine.

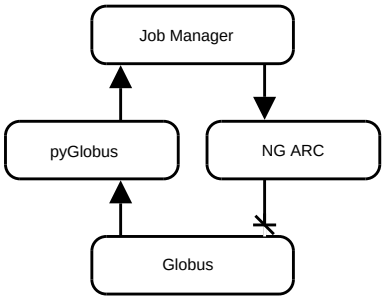


Figure F.2: Deadlock when a Globus callback exits the library and calling itself, resulting in a deadlock.

After having reached this conclusion we decided to leave the use of pyGlobus and use XML-RPC [75] over OpenSSL [61] using M2Crypto [66], even though this would require some additional work. The alternatives was to have insecure or non working RPC, or to fix the Globus framework, and we simply did not have the time to start debugging pyGlobus and Globus to find deadlocks. Finally leaving pyGlobus would also allow for much easier deployment for a Job Manager client, since no Globus installation would be required on the client side.

Bibliography

[1] David Abramson, Jon Giddy, and Lew Kotler. High performance parametric modeling with nimrod/G: Killer application for the global grid?, 2000.

[2] The Globus Alliance. Guide to globus toolkit 2.0 software. http://www.globus.org/gt2/install/download-guide.html.

[3] The Globus Alliance. The globus alliance: Ws-resource framework. http://www.globus.org/wsrf, April 2004.

[4] The Globus Alliance. The globus toolkit. http://www-unix.globus.org/toolkit/, May 2004.

[5] The Globus Alliance. Grid security infrastructure (gsi). http://www-unix.globus.org/security/, April 2004.

[6] The Globus Alliance. Ws-resource framework: Frequently asked questions. http://www.globus.org/wsrf/faq.asp, January 2004.

[7] Lawrence Besaw. Bsd socket reference. http://www.cs.iastate.edu/∼cs586/f03/notes/Socket_Reference.pdf, January 1991.

[8] Maciej Bogdanski, Michal Kosiedowski, Cezary Mazurek, and Malgorzata Wolniewicz. Progress – access environment to computational services performed by cluster of sun systems. http://progress.psnc.pl/English/cgw02.pdf, December 2002.

[9] Maciej Bogdanski, Michal Kosiedowski, Cezary Mazurek, and Malgorzata Wolniewicz. Grid service provider: How to improve flexibility of grid user interfaces? http://progress.psnc.pl/English/petersburg_progress.pdf, June 2003.

[10] The NorduGrid Collaboration. Nordugrid middleware, the advanced resource connector. http://www.nordugrid.org/middleware/.

[11] The NorduGrid Collaboration. Nordugrid general information. http://www.nordugrid.org/about.html, May 2004.

[12] World Wide Web Consortium. Soap version 1.2 part 1: Messaging framework. http://www.w3.org/TR/soap12-part1/, June 2003.

[13] World Wide Web Consortium. Web services architecture. http://www.w3.org/TR/ws-arch/, August 2003.

[14] World Wide Web Consortium. Soap specifications. http://www.w3.org/TR/soap/, May 2004.

[15] Oracle Corporation. Oracle grid computing. http://www.oracle.com/solutions/grid.

[16] Zope Corporation. Zope object database. http://zope.org/Products/ZODB3.2, June 2004.

[17] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems - Concept and Design*. Pearson Education Ltd., 3rd edition, 2003.

[18] Jon Crowcroft. *Open Distributed Systems*. UCL Press Limited, 1st edition, 1996.

[19] Inc. Cunningham & Cunningham. Dead lock. http://c2.com/cgi/wiki?DeadLock, May 2004.

[20] Inc. Cunningham & Cunningham. Race condition. http://c2.com/cgi/wiki?RaceCondition, May 2004.

[21] The European Datagrid. The datagrid project. http://eu-datagrid.web.cern.ch/eu-datagrid.

[22] Philip A. DesAutels. Sha1 version 1.0. http://www.w3.org/PICS/DSig/SHA1_1_0.html, October 1997.

[23] D. Eastlake and P. Jones. Rfc 3174 - us secure hash algorithm 1 (sha1). http://www.faqs.org/rfcs/rfc3174.html, September 2001.

[24] P. Eerola, T. Ekelof, M. Ellert, J. R. Hansen, S. Hellman, A. Konstantinov, B. Konya, T. Myklebust, J. L. Nielsen, F. Ould-Saada, O. Smirnova, and A. Waananen. Atlas data-challenge 1 on nordugrid. *CHEP'03*, 2003.

[25] Dave Beazley et al. Simplified wrapper and interface generator. http://www.swig.org/, April 2004.

[26] Karl Czajkowski et al. From open grid service infrastructure to ws-resource framework: Refactoring and evolution. http://www.ibm.com/developerworks/library/wsresource/ogsi_to_wsrf_1.0.pdf, March 2004.

[27] Internet Engineering Task Force. Internet x.509 public key infrastructure. http://www.ietf.org/rfc/rfc2459.txt, January 1999.

[28] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[29] Ian Foster. What is the grid? a three point checklist, July 2002.

[30] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2. Morgan Kaufmann, 1998.

[31] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2001.

[32] Python Software Foundation. Python programming language. http://python.org/, April 2004.

[33] The Python Software Foundation. logging - logging facility for python. http://docs.python.org/lib/module-logging.html, December 2003.

[34] The Python Software Foundation. What is python? http://python.org/doc/Summary.html, May 2004.

[35] G. Fox, M. Pierce, D. Gannon, and M. Thomas. Overview of grid computing environments, Febuary 2003.

[36] Alan O. Freier, Philip Karlton, and Paul C. Kocher. Ssl 3.0 specification. http://wp.netscape.com/eng/ssl3/, May 2004.

[37] Bernhard Herzog. Skencil: Homepage. http://sketch.sourceforge.net/, June 2004.

[38] T. Howes, S. Kille, and M. Wahl. Rfc 2251 - lightweight directory access protocol (v3). http://www.faqs.org/rfcs/rfc2251.html, December 1997.

[39] Keith R. Jackson. Python globus(pyglobus). http://www-itg.lbl.gov/gtg/projects/pyGlobus/.

[40] Henrik Thostrup Jensen and Jesper Ryge Leth. Automatic job resubmission in the nordugrid middleware. http://www.cs.auc.dk/∼htj/nordugrid/dat5_report.ps, January 2004.

[41] Brian W. Kerninghan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[42] Carl Kesselman. Gssapi_ssleay for globus security. http://www-fp.globus.org/presentations/retreat98/security/, Juli 1998.

[43] A. Konstantinov. The http and soap framework. http://www.nordugrid.org/documents/HTTP_SOAP.pdf, October 2003.

[44] A. Konstantinov. The nordugrid grid manager and gridftp server - description and administrators manual. http://www.nordugrid.org/documents/GM.pdf, July 2003.

[45] Alexander Konstantinov. The nordugrid smart storage element. http://grid.uio.no/cvs/cvsweb.cgi/∼checkout∼/nordugrid/doc/httpsd/SE.pdf, March 2004.

[46] Sun Microsystems. Grid computing solutions. http://wwws.sun.com/software/grid.

[47] Sun Microsystems. What is grid computing? http://www.sun.com/2003-1118/feature/grid.html.

[48] Oracle Technological Network. Oracle grid computing technologies. http://otn.oracle.com/products/oracle9i/grid_computing/index.html.

[49] Jakob Nielsen and Oxana Smirnova. Nordugrid / data challenges. http://www.nordugrid.org/slides/20031127-jakob.ppt, November 2003.

[50] Nordugrid. Nordic testbed for wide area computing and data handling (nordugrid), September 2001.

[51] Inc. Object Management Group. Welcome to the omg's corba website. http://www.corba.org/, May 2004.

[52] Farid Ould-Saada. Nordugrid sg meeting. http://www.nordugrid.org/slides/20031127-farid.sxi, November 2003.

[53] Legion Project. Legion a world wide virtual computer. http://legion.virginia.edu/.

[54] Legion Project. Legion: Frequently asked questions. http://legion.virginia.edu/FAQ.html.

[55] The Access Grid Project. Access grid. http://www.accessgrid.org/, May 2004.

[56] The Globus Project. About the globus toolkit. http://www-unix.globus.org/toolkit/about.html.

[57] The Globus Project. Globus collaborators. http://www.globus.org/about/collaborators.html.

[58] The Globus Project. Globus toolkit 3.0 fact sheet. http://www.globus.org/toolkit/gt3-factsheet.html.

[59] The Globus Project. Globus toolkit^{TM}2.4 overview. http://www.globus.org/gt2.4/overview.html.

[60] The OpenSSL Project. Openssl: Documents ssl(3). http://www.openssl.org/docs/ssl/ssl.html, May 2004.

[61] The OpenSSL Project. Openssl: The open source toolkit for ssl/tls. http://www.openssl.org/, May 2004.

[62] The pywebsvcs meta project. Python web services. http://pywebsvcs.sourceforge.net/.

[63] Kate Rhodes. Xml-rpc vs. soap. http://weblog.masukomi.org/writings/xml-rpc_vs_soap.htm, May 2004.

[64] Anthony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. http://research.microsoft.com/∼antr/PAST/pastry.pdf, November 2001.

[65] Abraham Silberschatz, Henry F Korth, and S Sudarshan. *Database System Concepts*. McGraw-Hill, 4th edition, 2002.

[66] Ng Pheng Siong. M2crypto - a python crypto and ssl toolkit. http://sandbox.rulemaker.net/ngps/m2/, May 2004.

[67] Martin Sjögren. pyopenssl - a python interface to the openssl library. http://pyopenssl.sourceforge.net/, May 2004.

[68] O. Smirnova. Extended resource specification language. http://www.nordugrid.org/documents/xrsl.pdf, October 2003.

[69] Bjarne Stroustrup. *The C++ Programming Language (Special Edition)*. Addison Wesley, 3rd edition, 1997.

[70] Inc. Sun Microsystems. Java remote method invokation (java rmi). http://java.sun.com/products/jdk/rmi/, May 2004.

[71] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall Inc., 3rd edition, 1996.

[72] The GIMP Team. Gimp: The gnu image manipulation program. http://www.gimp.org, June 2004.

[73] Johan Tordsson. Resource brokering for grid environments. http://www.cs.umu.se/ tordsson/thesis/, May 2004.

[74] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open grid services infrastructure (ogsi) version 1.0, 2003.

[75] Inc UserLand Software. Xml-rpc specification. http://www.xmlrpc.com/spec, May 2004.

[76] Jon Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL*. O'Reilly and Associates, Inc., 1st edition, 2002.