Leiden Grid Infrastructure: a new approach

Mark Somers, Hugo Meiland
Leiden Institute of Chemistry, Leiden University, The Netherlands.
m.somers@chem.leidenuniv.nl

Abstract

In this paper a lightweight Grid middleware called the Leiden Grid Infrastructure (LGI) is presented. The LGI has been designed with a new scheduling approach that is based on a 'pull'-type algorithm, this in contrast to many other Grid middlewares that are based on 'push'-type scheduling algorithms. Another different approach that has been undertaken is that the LGI was designed, from the ground up, with a set of usage requirements in mind, to meet a demand that has arisen in the field of practical High Performance Computing (HPC) on an enterprise-scale deployment, and not from a computer scientific interest point of view as such.

1. Introduction

The Leiden Grid Infrastructure (LGI) project has its origin in the experiences obtained in the field of practical High Performance Computing (HPC), as is practiced at the Leiden Institute of Chemistry (LIC). At the LIC, Beowulf clusters are regularly being deployed now for some years, together with more usage of the supercomputer facilities at the national level, to meet the demand for computing power for science. With the increase of use of these systems, several problems are appearing for the scientists that need all that computational power; 1) the computer systems the scientists need to work with are becoming rather diverse, 2) the programs that are used are increasingly more complex to install on these different platforms, 3) different working environments with respect to batch managers are to be dealt with and 4) in general, most scientist, nowadays, need to focus more on producing scientific results for their projects, rather than having also the time for learning new computer science related techniques. In the end, one can conclude that, in the future, we will probably see relatively more HPC-users and less HPC-developers active in the field.

On the other hand, despite the above mentioned problems, experience is that the more users there are, the applications used by these users tend to be more general, and the more HPC systems will be regarded as consumables. This indicates that the emerging problems might very well dealt with by introducing a Grid infrastructure[1]. However, in doing so, certain prerequisites are to be met[1]. The Grid infrastructure needs to be based on a middleware that 1) is easy to use, maintain and setup by regular system administrators and should not introduce yet more complexity for users and administrators, 2) allows for the use of propriety applications for which certain licensing limits are to be met, 3) allows for deployment on a variety of resources that are not within the administrative domain and without the need for any extra privileges than regular users are expected to have, 4) allows for a diversity of application specific interfaces of which some are already developed and are natural to the task at hand and scientist performing the tasks and finally 5) allows for scalability and continuity of the infrastructure without any interference in the work flow of the users.

2. Why a different approach?

A literature study on the currently available Grid middlewares shows that most middlewares, including the well known ones like the Globus Toolkit[2] and UNICORE[3], are based on so-called “push”-type scheduling algorithms[4]. In these push-type algorithm based architectures, a shared service is available to which users submit their jobs. It is that shared “matchmaker” that, based on resource information it has or gathers, decides on what resource the submitted job is to be run. Usually, the job is then send to the resource by the matchmaker itself and this automatically implies that the resource has to offer services to the outside world too. Apart from the well known scheduling problems these push algorithms bring about (gathering the up-to-date resource states, how to properly schedule a job in the dynamical Grid environment, how to solve the NP-hard problem and how to keep all that scalable for large numbers of jobs or resources[4]), the implication of having to install a service on the resource is in clear contradiction with the third prerequisite we stated in the introduction.

On the other hand, when considering a “pull”-type algorithm, the third prerequisite can easily be met. Scalability to a high degree can also be achieved[4], as has been successfully demonstrated in two different projects; the Distributed Implementation with Remote Agent Control(DIRAC) project[5] and the Berkeley Open Infrastructure for Network Computing (BOINC) project[6]. Although, of the two, BOINC can be run on a resource under the third prerequisite, both architectures still rely on a form of matchmaking being performed on a shared service. Both also implement architectures that keep track of the capabilities of all the resources, despite the fact that they were developed with two different paradigms in mind. Within DIRAC this is achieved by using an instant messaging technique and in BOINC this is achieved through the client reporting basic hardware capabilities of the resource only once during setup.

Clearly, having an architecture with a single shared matchmaking service not only still represents a possible bottleneck, it does not solve the NP-hard problem of the matchmaking itself [4,5]. Moreover, it also requires some sort of abstraction to describe or encapsulate all the possible resource hardwares and job requirements that might be encountered in the matchmaking process itself. Traditional Grid middlewares have tried to solve this issue by introducing a general an extendable Job Submission Description Language (JSDL)[7], to somehow represent the hardware requirements one might need for a certain job, despite the fact that the actual determination of job requirements by itself is already known to be rather problematic in a Grid environment[4,8].

The mere use of a matchmaking service seems to contradict the fact that the decision of whether or not a given system can perform a certain task, can ultimately only be made by that system itself. Clearly, that decision can be made only if all the information that defines the task is available to that system. It seems to be very natural to the matchmaking problem at hand, to let a possible target resource decide for itself if a task can be performed or not, because an architecture based on that principle would not only automatically introduce a fail-over mechanism and allow for far more scalability, it would also alleviate the necessity of having resource or hardware related job requirement elements in the JSDL altogether. Even more, the philosophy of the principle is very much in correspondence with the fact that for most of the tasks encountered in the production environment that was described in the introduction, a simple application name and either a small description of the input, or the input text to the application itself, fully defines the complete task at hand. With this basic principle in mind, the general scheduling scheme would now be to have a list of jobs from which resources can pick out only those jobs they can handle because they have been setup to handle them. Or to put it in other words, in this scheme the science is brought to the computers and not the computers to the scientists.

Though, it is emphasized that the presented work has it's origin in meeting a demand in the field of HPC, the approach that was adopted in the process might still represent a new field open for research in Grid related computer science in general. Therefore, in the rest of this document, our implementation of a lightweight Grid middleware, the LGI, will be described, that has tried to encapsulate the considerations of this new approach while adhering to the set prerequisites of the introduction.

3. The Leiden Grid Infrastructure

The basic design of the LGI can best be understood considering Figure 1. In general, within an LGI project, one or more project web servers are defined to which users submit jobs for applications through various interfaces. These interfaces are often very dependent on the specific application. The resources on the other hand, periodically request work, only for those applications they can handle, from the project servers, through daemons that run on these resources. The daemon runs as an unprivileged user on the resource. Once a task has been accepted by a resource, the corresponding daemon will pass it through to the batch system running on that resource. Once the task is finished, the daemon sends back the results directly, or a reference to the results, to the project web server. Application interfaces can easily monitor the state of a job and once that is finished, it can present the results to the user, either through the reference, or directly. Within the LGI, all communication to project web servers is done through a basic Remote Procedure Call (RPC) Application Programming Interface (API) using Extendable Markup Language (XML)[9] over HyperText Transfer Protocol Secure (HTTPS)[10], and using x509 client- and server-certificates[11].

Although this work flow is basic, it suffices for many of the tasks found in the HPC field. However, the LGI also offers possibilities for more specialized work flows, which again are very application dependent. In the following sections, the different components of the architecture will be presented and it will be made clear how this basic design implements a very flexible framework. More details on the LGI setup can be found elsewhere[12].

3.1. Application interfaces

Several types of application interfaces can be used within the LGI. All interfaces communicate to the project web server by using RPC procedures over HTTPS using x509 client-certificates. The client-certificate corresponds to the identity submitting the job. This identity could be the actual person itself that is using the interface; i.e. by using a web browser, or any other Graphical User Interface (GUI), with a PKCS12[13] formated certificate and private key loaded into the browser or GUI. The identity could also very well be a general LGI identity that submits work on behalf of a group of users who have authenticated themselves through the (web-based) GUI by login in. The x509 certificates used within a project are all signed by the Certificate Authority (CA) of that project. With this setup, a single sign-on can be easily implemented for the user.

In the current setup of the LGI, two types of general interfaces are implemented by default: 1) a command line interface for POSIX[14] compliant systems that mimics a general batch manager interface and 2) a basic web based interface on the project web server itself. Both these interfaces use the client-certificate and private key of the actual user directly and implement a single sign-on mechanism.

3.2. Project web servers

The project server has been implemented using Apache[15], MySQL[16] and PHP[17]. Apache was configured for HTTPS using using x509 client- and server-certificates all signed by the project's CA. Three types of RPC procedure classes have been implemented in PHP that make use of a MySQL database. The MySQL database name is equal to the project name. This allows for several independent databases to be used for independent projects on the same web server hardware. The three different RPC procedure classes are for 1) interfaces communicating with the project web server, 2) resources communicating with the project web server and 3) for communication between different project web servers.

In the following sub-sections some details will be given on the database structure and these three different RPC procedure classes.

3.2.1. MySQL database: The database on the project web server consists of nine different tables[12]. The most important tables are the '*job_queue*', the '*running_locks*' and the '*running_sessions*' tables. These will be briefly discussed here.

The '*job_queue*' table represent the jobs users have submitted into the LGI. It contains fields for specifying the '*application*', the '*job_state*', the '*owners*' of the job, who has '*read_access*' to the job, what the '*target_resources*' are, the application specific '*input*' and '*output*' and the '*lock_state*' of the job. The '*lock_state*' is used to signal a lock on the job and is very important to avoid race-conditions among resources inspecting jobs for execution. If this integer field is non-zero, a lock was set on the job and details of that lock can be found in the '*running_locks*' table. The '*running_locks*' table keeps track of which resource locked what job for what running session. Sessions of resources are kept track of in the '*running_sessions*' table. Further details on the precise structure of the database can be found elsewhere[12].

3.2.2. Interface RPC procedures: On the project web server five RPC procedures have been defined for interfaces specifically: '*interface_submit_job.php*', '*interface_job_state.php*', '*interface_delete_job.php*', '*interface_project_server_list.php*' and '*interface_project_resource_list.php*'. These five procedures can be called by using the HTTP POST [18] method to the project web server. The PHP implemented procedures check if a valid CA signed certificate was used and that the identity POSTed corresponds to the identity present in the used x509 certificate. Only if the correct fields for a request have been POSTed, the PHP code performs the request, using the MySQL database on the back-end, and will generate the response XML. Details on the precise XML format used in these procedures can be found elsewhere[12]. The basic command line interface that is implemented by default into the LGI uses the interface specific RPC procedures and can be used as an example for other interfaces.

3.2.3. Resource RPC procedures: For resources communicating with the project web server, a set of ten resource specific RPC procedures has been defined: '*resource_signup_resource.php*', '*resource_signoff_resource.php*', '*resource_request_work.php*', '*resource_request_job_details.php*', '*resource_lock_job.php*', '*resource_unlock_job.php*', '*resource_update_job.php*', '*resource_submit_job.php*', '*resource_job_state.php*' and '*resource_request_resource_data.php*'. Again, with these procedures, the resource daemon needs to POST the correct fields before the request is served and again checks on whether the certificate was correctly signed by the project's CA and whether the resource was enlisted in the project's '*active_resources*' table, are performed.

Within the LGI a resource is only allowed to update a job if the job was correctly locked by that resource. The job can only be locked by the resource if the resource was one of the '*target_resources*' of the job and the job was not locked by any other resource. Before any locks can be set on jobs or any requests for work can be made by a resource, the resource needs to setup a session with the project web server by signing up to that server. The session is kept track of and is used to remove locks set on jobs by resources that seem to be inactive too long by the project web server. When a resource has successfully signed up to a project web server, it can request for work for a specific application. It does that by POSTing, among other fields, the '*application*' field to the '*resource_request_work.php*' RPC procedure. This procedure selects a set of unlocked jobs in the '*queued*' state for that specific application which also have the requesting resource in the '*target_resources*' field (usually this field states 'any' and so the job can be offered to any resource). The jobs send back in the XML response are automatically locked by the server for that resource. The resource should now inspect the details of these jobs and decide to accept any one of them or not. If it does accept a job, it should change the job's state into '*running*' and unlock it. If it decides to reject the job, it should just unlock it. More details on the specific XML formats and fields POSTed in these RPC procedures can be found elsewhere[12].

3.2.3. Project server RPC procedures: To achieve complete scalability in the architecture, extra project servers can be added to the project. For two or more project servers, each with their own MySQL database for the project, to be able to communicate, a server specific set with two RPC procedures has been defined[12]. With these two procedures, and the '*updates'* table in the database, changes to any project database can be easily synchronized among all the servers of a project[12].

3.3. Resource daemon

Within the LGI, each resource is identified through yet another x509 client-certificate that was signed by the project's CA. Each resource also has a daemon running and it is this daemon that communicates to all of the project web servers. Moreover, the daemon is allowed to participate in several projects. The daemon itself has been implemented using standard C++[19] using the Standard Template Library (STL)[20] and the libCurl[21] library. The current implementation of the daemon is found to be portable enough for any POSIX compliant resource. The workings of the daemon can best be understood by considering the XML based example configuration in Figure 2.

From Figure 2 it is clear that for each project the daemon participates in, any number of applications can be setup. Moreover, user and group limits on the number of jobs can be set at the resource, the project and the application level if desired. For each application the daemon can handle, simple shell scripts are used to implement the application. The script '*check_system_limits_scripts*' is used to determine, even before requesting any new jobs, whether or not a system wide limit has been reached. The '*job_check_limits_script*' is used to determine whether or not a specific job can actually be run on the system. Before this script is executed, the daemon will create a 'job-directory' in the directory specified with the '*run_directory*' tag of the configuration. In this job-directory information regarding the job is placed by the daemon in hash-protected files. By using the job-directories, that cache the actual job information, the daemon has become resilient to system crashes on the resource.

The '*job_check_limits_script*' script can make use of the cached information and decide if the job can be completed. If so, the script should return a zero exit status to the daemon. Once a job can be dealt with, the '*input*' field of the job is imported into the job-directory and the '*job_prologue_script*' will be executed. If the prologue script also returns a zero exit status, the job will be set into the '*running*' state on the project web server and the '*job_run_script*' will be forked into the background.

The daemon will poll the status of the jobs it monitors through the '*job_check_running_script*' and '*job_check_finished_script*' scripts. If the job has finished, the '*output*' field is POSTed back to the project server where the job resides in the database and the state will be set into the '*finished*' state only if the '*job_epilogue_script*' has been called successfully.

The status of the job in the database is also monitored at regular intervals by the daemon. As soon as the daemon detects a change in status of the job in the database on the project web server, the current state from the database is again imported into the job-directory.

By using these basic shell scripts, applications can easily be implemented to run on any POSIX compliant resource. It also allows for an easy setup to any batch managers that might be present on the resource. Moreover, it allows the daemon to run as a regular user on the resource and any system or user limits imposed on the users by the resource administrators, are automatically also imposed onto the jobs that trickle in through LGI. More details can be found elsewhere[12].

3.4. Job bulk input and output

Currently, within LGI, the database can accommodate any input or output, but not too big (BLOBs have been used). However, by installing the x509 client-certificates of the interfaces and resources as OpenSSH[22] RSA/DSA public keys onto a separate file up- and down-load server, the '*input*' and '*output*' fields of the job can be used to refer to the job specific up- and down-load areas on the shared server. This allows for an easy setup of applications on resources; the corresponding '*job_run_script*' script can just 'scp' in the input into the job-directory, and when finished, 'scp' the output back, using the issued x509 client-certificates to authenticate. Within LGI this poses no real security issues because there is already a chain of trust between the interface and resources through these x509 client-certificates used in conjunction with the LGI project web server, setup by the project administrator.

The more preferred method of bulk job IO is currently being implemented in LGI. In this method two special applications are defined within the LGI; the '*file-storage*' and the '*file-transfer*' applications. With these special applications, the storage of data will now be associated with a 'running' '*file-storage*' job on a resource. To retrieve or store the data, a special '*file-transfer*' job is issued. Interfaces will also be able to make use of such '*file-transfer*' jobs to retrieve data from the LGI[12].

4. Summary

In this paper a lightweight Grid middleware, the Leiden Grid Infrastructure, has been presented. The LGI was designed specifically to solve issues encountered in the field of practical HPC with certain prerequisites in mind and at the same implements perhaps a new approach to Grid scheduling. Within LGI the scheduling is implemented in a scalable and distributable fashion in which the matchmaking is now being preformed on the resources. The resources are now assumed to be suitably setup for special often used applications and it was argued that with this philosophy the need for a JSDL could be relieved.

6. References

[1] M. Parashar and C. A. Lee, “Grid computing: introduction and overview”, http://www.caip.rutgers.edu/TASSL/Papers/proc-ieee-intro-04.pdf

[2] I. Foster, “Globus toolkit version 4: software for service-oriented systems”, Journal of computational sciences and technology, Vol. 21, No. 4 pp. 513-520, 2006.

[3] D. Erwin, M. Rambadt, A. Streit and Ph. Wieder, “Production-quality grid environments with UNICORE”, Proceedings of the Workshop on Grid Applications: From Early Adopters to Mainstream Users in Conjunction with Global Grid Forum 14, pp. 10 - 17, Chicago US, 2006.

[4] F. Dong and S. G. Akl, “Scheduling algorithms for grid computing: state of the art and open problems”, Technical report no. 2006-504, Queen's University, Canada, 2006.

[5] I. Stokes-Rees, A. Tsaregorodtsev and V. Garonne, “DIRAC Lightweight information and monitoring services using XML-RPC and instant messaging”, CHEP 2004 proceedings, 2004.

[6] D. P. Anderson, “BOINC: a system for public-resource computing and storage”, GRID 2004: 4-10, http://boinc.berkeley.edu/grid_paper_04.pdf

[7] A. Anjomshoaa et. al., “Job Submission Description Language (JSDL) specification version 1.0”, http://www.ogf.org/documents/GFD.56.pdf

[8] F. Berman, “High-performance schedulers”, chapter in “The grid: blueprint for a future computing infrastructure”, Morgan Kaufmann Publishers, 1998.

[9] http://www.xml.org

[10] http://www.openssl.org

[11] http://www.itu.int/rec/T-REC-X.509-200508-I

[12] http://fwnc7003.leidenuniv.nl/LGI/docs

[13] http://www.rsa.com/rsalabs/node.asp?id=2138

[14] http://standards.ieee.org/regauth/posix

[15] http://www.apache.org

[16] http://www.mysql.com

[17] http://www.php.net

[18] ftp://ftp.isi.edu/in-notes/rfc2617.txt

[19] B. Stroustrup, “The C++ programming language”, Addison-Wesley Pub Co; 3rd edition, 2000.

[20] http://www.sgi.com/tech/stl

[21] http://curl.haxx.se/libcurl

[22] http://www.openssh.com

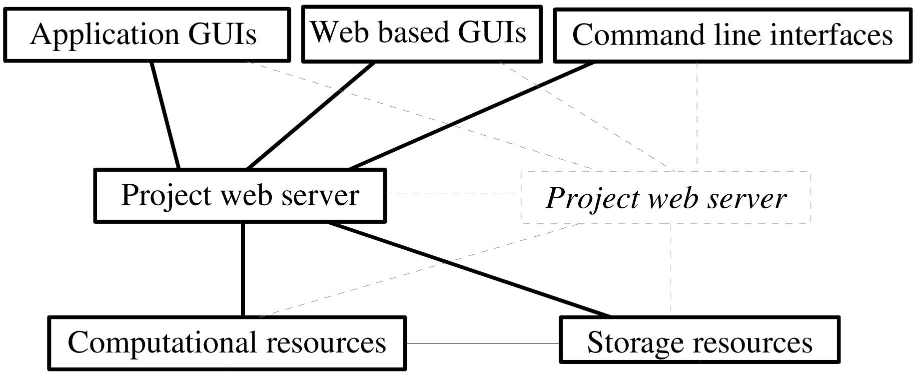

Figure 1: the basic design of the LGI. Dashed boxes and lines represent additions to the basic architecture.

```
<LGI>
        <ca_certificate_file> /home/mark/temp/LGI/certificates/LGI+CA.crt </ca_certificate_file>
        <resource>
                <resource_certificate_file> /home/mark/temp/LGI/certificates/fwnc7003.crt
</resource_certificate_file>
                <resource_key_file> /home/mark/temp/LGI/certificates/fwnc7003.key </resource_key_file>
                <run_directory> /home/mark/temp/LGI/daemon/runhere </run_directory>
                <owner_allow> <any> 10 </any> </owner_allow>
                <owner_deny> nobody </owner_deny>
                <job_limit> 20 </job_limit>
                <number_of_projects> 1 </number_of_projects>
                <project number='1'>
                        <project_name> LGI </project_name>
                        <project_master_server> https://fwnc7003.leidenuniv.nl/LGI </project_master_server>
                        <owner_allow> <any> 5 </any> </owner_allow>
                        <owner_deny> nobody </owner_deny>
                        <job_limit> 10 </job_limit>
                        <number_of_applications> 1 </number_of_applications>
                        <application number='1'>
                                <application_name> hello_world </application_name>
                                <owner_allow> <any> 2 </any> </owner_allow>
                                <owner_deny> nobody </owner_deny>
                                <job_limit> 4 </job_limit>
                                <max_output_size> 4096 </max_output_size>
                                <check_system_limits_script> ./hello_world_scripts/check_system_limits_script
</check_system_limits_script>
                                <job_check_limits_script> ./hello_world_scripts/job_check_limits_script
</job_check_limits_script>
                                <job_check_running_script> ./hello_world_scripts/job_check_running_script
</job_check_running_script>
                                <job_check_finished_script> ./hello_world_scripts/job_check_finished_script
</job_check_finished_script>
                                <job_prologue_script> ./hello_world_scripts/job_prologue_script
</job_prologue_script>
                                <job_run_script> ./hello_world_scripts/job_run_script </job_run_script>
                                <job_epilogue_script> ./hello_world_scripts/job_epilogue_script
</job_epilogue_script>
                                <job_abort_script> ./hello_world_scripts/job_abort_script </job_abort_script>
                        </application>
                </project>
        </resource>
</LGI>
```

Figure 2: An example configuration of the LGI resource daemon.