Leiden Grid Infrastructure

Dr. M.F. Somers

Introduction:

"The Leiden Grid Infrastructure (LGI) is a framework that allows scientists to perform computational tasks and store data with a common interface without needing to know the details of the underlying systems. Moreover the infrastructure allows for a work and load balancing of the underlying systems; The calculation will run on a machine, being a resource within the grid, that can actually do the calculation. Data will be stored again on a resource within the grid in whatever format the resource can handle wherever there is place. Furthermore, with the LGI a scalable infrastructure is built that can grow whenever the demand increases for more resources."

The above stated general goal of the LGI can be implemented according to the infrastructural design, made clear in this document. The design is based on well-known and proven technologies.

Rationale:

Researchers and students of today cannot be expected to understand all the technicalities involved in using a computer or storage device efficiently; each system behaves differently in it's details and because the technologies involved in the computational field are growing in such a rapid rate, keeping up with that knowledge, together with keeping up with the scientific knowledge, is hard to do.

Past experiences on supercomputers and clusters has shown that on average such a system is made redundant within 6 years. Often, when continuing present work on newly made available systems, a lot of porting is involved and a detailed knowledge of the technicalities of the new hardware is needed. Keeping current software running in such a dynamic surrounding can be a rather time consuming job. Just think about the new technologies like having your programs run in parallel, either using OpenMP or MPI, or perhaps even both is the system allows for this... Also think about which compiler and with what setting to use for your new hardware...

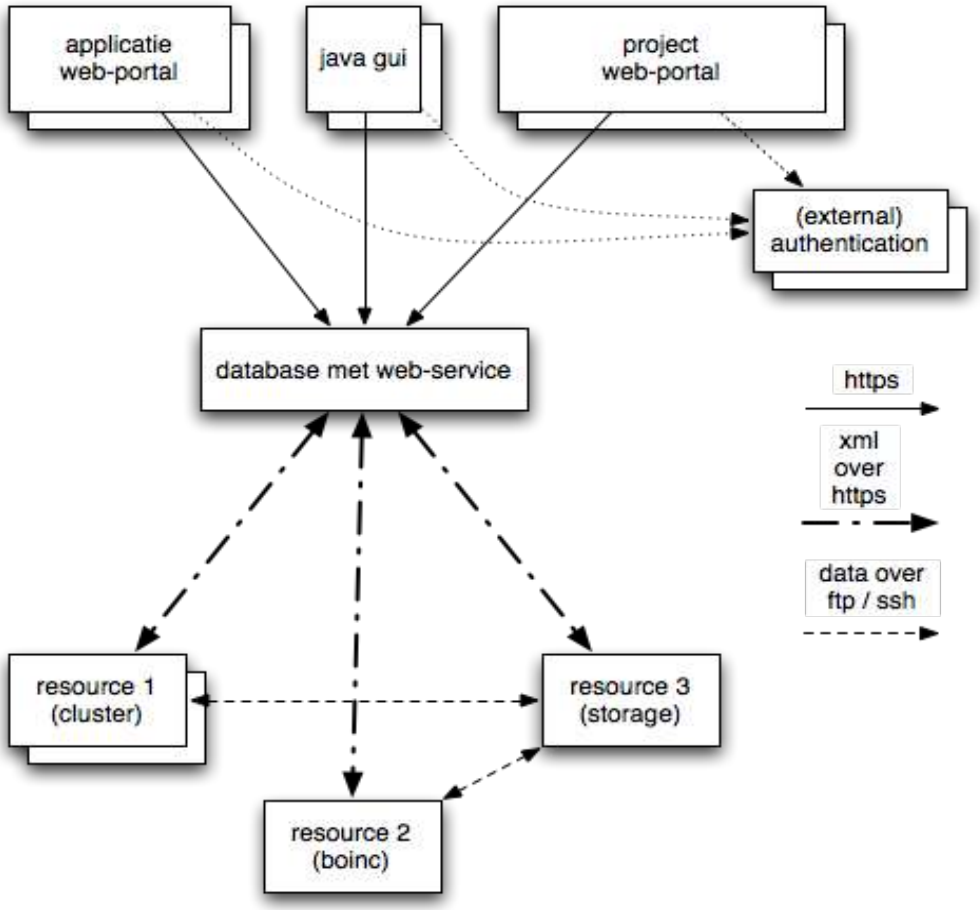
A general scientist should not be bothered with all of this, especially if it is known that most of the software run on computers is usually compiled and set up by others, and all this is often restricted to only a few general applications. The same rationale applies to you own desktop system. You are probably using Windows, or perhaps Linux. What software was installed and why do you use it? Count the number of applications you use on a daily basis and see for how many of them you actually compiled the application from sources yourself!

Nowadays, computers can be used far more efficiently, both from a user, and from a

management perspective, if the computers are part of a low-cost, perhaps application specific, but a very scalable grid infrastructure based on proven technologies.

General design:

The general LGI design is given by the following figure;



For each ***project***, any number of ***applications*** can be set up on any number of ***resources***. The ***application interface***, a common interface for an application within a project, communicates with any number of ***project web-servers***. One project web-server is designated to be the ***master project web-server***.

The project web-servers are nothing more than machines running a web-service to a general MySQL database with a known structure. The structure of the MySQL database will be detailed upon in this document: i.e. each project web-server contains an active list of all the resources within the LGI for that project and all the needed user management information. The project, within this framework, is nothing more than a specific MySQL database on each of the project web-servers. A single web-server machine can therefore automatically handle multiple projects independently.

The communication between an application interface and the project web-server is based on exchange of XML data over https using server- and client-certificates, all signed by the Leiden Grid Infrastructure Central Authority (LGI-CA). The LGI-CA is trusted by all clients running application interfaces, all project web-servers and all resources within the LGI.

The application interfaces queue calculation- or storage-jobs into any of the project's web-server databases. This is done by requesting the project's master web-server to get a list of project web-server databases. This allows for a round-robin load balancing over these project web-servers, if desired. An application interface can also monitor the completion of these jobs and depending on the details of the project, queue as much work as is allowed according to the users credentials. The user management for the project application can be implemented in the application interface itself but can also be set up on each of the project web-server database. The project web-servers only accept application interface clients with a valid LGI-CA singed certificate.

Application interfaces can be implemented in any programming language available as long as they communicate over https, using client- and server-certificates, all signed by the LGI-CA and adhere to the web-services offered by the project web-servers.

Resources, on the other end in this design, are nothing more than computers capable of doing calculations or storing data. A resource is identified with an unique ***resource name***. In the project's master web-server an active resource table exists which couples the unique resource name to an user account on the resource (by using i.e. a 'somers@teras.sara.nl' type URL) and to it's LGI-CA signed certificate, which in turn contains the public part of the RSA key-pair of the resource, as usual. The unique name of the resource is also put into the client-certificate as the common name. More details on the use of certificates is given below.

A resource essentially runs a ***resource-daemon*** in user-space and regularly polls the master and slave web-servers, perhaps even for different projects, to see if it is capable of performing any of the queued tasks for an application that was previously set up on that specific resource. The communication again makes use of XML over https, again with server- and client-certificates, all signed by the trusted LGI-CA. This allows for the project web-servers to identify requests through the verified client-certificates it receives.

The resource is regarded to be out of the administrating domain of the LGI; the owner and administrator of the resource can decide for which project's it will do what sort of work and for which users or groups of users for each of those project's applications. The resource-daemon runs usually as a non-privileged user on the resource and has only a limited access to the resources available on the resource. This construct allows for, for example, regular user accounts on super-computers to be set up as a resource within the LGI for that specific project, user or group of users. More details on the resource daemon implementations are given in the following sections.

Within this design, storage and file transfers are dealt with similar to computational

tasks; storing a file is regarded to be a running '***file storage***' job in the project's web-server. It can be queued (only by a resource) for a specific resource, identified through the unique resource name within the project, or for any storage resource that is capable of storing the data in one way or the other.

Database structure

As is clear from the previous two sections, the project web-server, and with it the database, plays a central role in the LGI. A general and sound design of it's structure is therefore of out-most importance. In this part of the document some details on that matter are presented.

The project's web-server contains a MySQL database for each project it might serve for. The name of the database is taken to be equal to the project name. In the project database the following tables are defined;

1) The table '***job_queue***';

field	***data type***
job_id	integer
state	string
application	string
owners	string
read_access	string
target_resources	string
input	blob
output	blob
job_specifics	string
lock_state	integer
state_time_stamp	integer

This table shows the state for each job in the project's web-sever. It contains the 'job_id' assigned to the job, it's 'state', for what 'application' it is, for which 'target_resources', which groups or users are the 'owners' of the job and what groups or users have 'read_access'. Only users and groups listed in the 'owners' field can modify the job. If the 'target_resources' contains “any”, any capable resource can pick up the job. The 'owners' and 'groups' fields are comma separated values and can contain the “any” keyword.

The 'job_specifics' may contain extra details on the specific job, perhaps even in XML format. It might contain extra scheduling information, specific to the application and the target_resources. The resource can decide to accept the job based on information present in that string. The 'input' and 'output' fields are general input and output fields. The interpretation of these fields depend on the application in question. They need not be used.

The 'state_time_stamp' is the time this job went into the state it currently is. The 'lock_state' field is used to signal a lock on this specific job. If the number is non-zero positive, an entry in the 'running_locks' table should be present. Details on the locking mechanisms are given below.

2) The table '***running_locks***';

field	***data type***
lock_id	integer
job_id	integer
resource_id	integer
lock_time	integer
session_id	integer

This table will contain all the current running locks on the jobs of the 'job_queue' table for each session of resources. If a job in the 'job_queue' table is locked by a resource in a session, the 'lock_state' is a positive non-zero number in the 'job_queue' entry and an entry in this table is present. The 'job_id' indicates for which job in 'job_queue' this lock is, the 'resource_id' indicates, through the 'active_resources' table below, which resource has locked the job at what time, which is specified by 'lock_time' and the 'session_id' indicates what session of the resource has locked the job. Details on the locking mechanisms are given below.

3) The table '***active_resources***';

field	***data type***
resource_id	integer
resource_name	string
client_certificate	blob
last_call_time	integer
project_server	integer
url	string

This table contains the currently signed-up resouces within the LGI for this specific project. The 'last_call_time' keeps track of the last time this resource has accessed this web-server. The 'project_server' defines what sort of project server the resource is; 0 = no project server, 1 = master project server , 2 = slave project server. The 'url' field specifies the location, on the web, and under what user account this resource, on the resource itself, is part of the grid. The 'client_certificate' contains a copy of the resource's certificate.

4) The table '***running_sessions***';

field	***data type***
session_id	integer
resource_id	integer
session_time_stamp	integer

This table stores all the running sessions of all the resources. The 'session_id' uniquely identifies the session for the resource with id 'resource_id', The 'sessions_time_stamp' is a time value that is reset at each web request for which the resource needs a full session. If a session has timed-out for more than 30 minutes, the locks of that session are removed and the session itself is quited by the project web-server. any service to the resource, for that specific session, is the denied.

5) The table '***users_allowed***';

field	***data type***
user_name	string
application	string
job_limit	integer

This table contains users who are granted access to the project web-server. If a specific user is not in this list, the record with the user_name equal to “any” can be used for the specific application, however, in that case the groups setting are first checked. The application entry can also be set to “any” to allow access to all applications within the project. If no such “any” 'user_name' is present in the list, the requesting user should be denied any service.

If the 'job_limit' value is negative, it specifies the maximum number of 'running' and 'queued' jobs this user is allowed to have for the application on this project-server. If the number is positive it specifies the total number of jobs this user is allowed to have on this project-server, whatever the state. If the number is zero, there is no limit.

User settings always overrule all group settings, unless the “any” user record was used

and a specific group record is present in the 'groups_allowed' table.

6) The table '***groups_allowed***';

field	***data type***
group_name	string
application	string
job_limit	integer

This table contains the groups who are granted access to the project web-server. If a specific group is not in this list, the group should be denied any service for the specific application. This can be overruled by allowing for an “any” group. For application the “any” name can also be specified. However, in that case, the user tables are first consulted to check if there was no “any” user defined.

If the 'job_limit' value is negative, it specifies the maximum number of 'running' jobs this group is allowed to have for the application. If the number is positive it specifies the total number of jobs this group is allowed to have. If the number is zero, there is no limit.

Group settings are always overruled by user settings.

7) The table '***users_denied***';

field	***data type***
user_name	string
application	string

This table specifies which user is denied service for an application. This table overrules the 'users_allowed' and 'groups_allowed' tables. If the application is set to “any”, all service will be denied. The “any” 'user_name' is also allowed in this table.

8) The table '***groups_denied***';

field	***data type***
group_name	string
application	string

This table specifies which group is denied any service for an application. This table overrules the 'user_allowed' and 'groups_allowed' tables. If the application is set to “any”, all service will be denied. The “any” group can also be present in the table.

9) The table '***updates***';

field	***data type***
version	integer
servers	string
update_query	blob

This table contains all the updates this project server knows about. Each update is identified by a non-zero positive version number. If the update was intended for the specific server, listed in the comma separated 'servers' field, the 'update_query' has been applied to this servers project database. Only sequential updates are performed. Updates not intended for this server are still logged into this table.

Resource interfacing with the project's web-server

In this part the API between the project's web-server and a 'resource' is detailed upon.

The web-interface to a project web-server will be implemented in php and by using XML data transfer over https, using client- server-certificates. Using these certificates, the resource can be identified through the 'common name' in the certificate. All requests are made by passing data to the php scripts through the POST method. All responses from the project's web-server will be wrapped into "***<LGI> <response> </response> </LGI>***" XML tags.

If an error condition has occurred during the request, a "***<error> <number> 1 </number> <message> Back-off </message> </error>***" type of response is generated. The non-zero positive numbers, enclosed in the "***<number> </number>***" tags, constitutes an error and has with it a corresponding message within the "***<message> </message>***" tags.

If a server is overloaded it should respond with the above 'back-off' error message. The resource should then try to look for the "***<timeout> N </timeout>***" tags, within the "<error> </error>" message and wait for at least N seconds before trying again. It could also decide to try and request a different project's web-server in the mean-time.

If a resource, not present in the web-servers 'active_resources' table does a request, an error is returned and service is denied. Also if the certificate of a resource does not match the one stored in the 'active_resources' table, service will be denied.

The following requests can be made to the web-server. They are listed under the 'resources' directory on the project server's main URL. For example "https://fwnc7003.leidenuniv.nl/LGI/resources/resource_signup_resource.php".

1) The request '***resource_signup_resource***';

This request is made by the resource prior to any use of the project's web-services. It allows the project's web-server to setup a new session. The session number, which uniquely identifies this session, will be returned in the response and future requests should include that session id in their request when a full session is needed for that request. This request only needs the unique identifier of the resource, which is taken from the certificate's common name field, and the '***project***' name, which is POSTed.

The response, when an error occurred, on this request is an "<LGI> <response> </response> </LGI>" encapsulated error, as described above.

For a valid response, extra information can be placed by the web-server if needed, also encapsulated within the "<LGI> <response> </response> </LGI>" tags. Currently only the list of project's slave web-servers are exported by using the "***<project_server number=x> url </project_server>***" tags in which x is the logical number of the slave server. The total number of servers is also exported in the "***<number_of_slave_servers> N </number_of_slave_servers>***" tags.

An example response is given below:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <resource> testresource </resource>
                <project> testproject </project>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
                <number_of_slave_servers> 2 </number_of_slave_servers>
                <project_sever number=1> https://fwnc7003.leidenuniv.nl/testslaveserver
</project_server>
                <project_sever number=2>
https://fwnc7003.leidenuniv.nl/testslaveserver2 </project_server>
                <session_id> 101 </session_id>
        </response>
</LGI>
```

2) The request '***resource_signoff_resource***';

This request is made by the resource to signal the end use of the project's web-services for the session identified through the posted '***session_id***' field. It allows the project's web-server to unlock any stale locks this resource might have on jobs in the project database and to kill the session it had with the resource. This request needs the unique identifier of the resource, which is taken from the certificate's common name field, the 'project' name, and the 'session_id' which should all be POSTed. The response generated to this request is exactly the same as described in the above 'resource_signup_resource' request.

3) The request '***resource_request_work***';

This request is made by a resource that seeks for work to be done. If an unregistered resource does the request, the request will be ignored and an error response will be send back, as was previously described. A valid session of the resource is required to be able to serve this request.

The common name field of the resource certificate is used to identify resource doing the request. The 'session_id', '***application***' and the '***project***' fields are set by a POST. The POSTing of the '***start***' and '***limit***' fields are optional in this request. If these fields are not POSTed, a default 'limit' of 10 will be used and if not POSTed, the 'start' is set to 0.

The web-server will query the 'job_queue' table, of the 'project' table in it's database for unlocked jobs with an offset of 'start' in the 'queued' state for the requesting resource. It will generate locks for these rows on the fly as is described below and will respond with the XML data enclosed in the "<LGI> <response> </response> </LGI>" tags.

The response will have a "***<number_of_jobs> N </number_of_jobs>***" section that gives the amount of jobs returned in the response. It will also have details on the requesting resource, project, the application of the request and the server's involved.

For each job in the response the "***<job number=x> </job>***" tags are used with x set equal to the logical number of the job in the response. Within these "<job> </job>" tags, the 'job_id', 'owners', the 'read_access', 'state_time_stamp' , 'the_job_specifics', and 'target_resources' are also specified. An example of a valid response is:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <start> 0 </start>
                <limit> 10 </limit>
                <resource> testresource </resource>
                <project> testproject </project>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
                <session_id> 102 </session_id>
                <application> testapp </application>
                <number_of_jobs> 3 </number_of_jobs>

                <job number=1>
                        <job_id> 140 </job_id>
                        <target_resources> testresource </target_resources>
                        <owners> mark </owners>
                        <read_access> theor, sara, cyttron </read_access>
                        <state_time_stamp> 1259926661 </state_time_stamp>
                        <job_specifics>
                                <nprocs> 2 </nprocs>
                                <disk> 5G </disk>
                                <mem> 64G </mem>
                        </job_specifics>
                 </job>

                <job number=2>
                        <job_id> 142 </job_id>
```

```
                        <target_resources> testresource </target_resources>
                        <owners> freek, cyttron </owners>
                        <read_access> cyttron </read_access>
                        <state_time_stamp> 1259927661 </state_time_stamp>
                        <job_specifics>
                                <nprocs> 1 </nprocs>
                                <mem> 64M </mem>
                        </job_specifics>
                </job>

                 <job number=3>
                        <job_id> 147 </job_id>
                        <target_resources> any </target_resources>
                        <owners> sjoerd, cyttron </owners>
                        <read_access> any </read_access>
                        <state_time_stamp> 1259936661 </state_time_stamp>
                        <job_specifics> </job_specifics>
                </job>
        </response>
</LGI>
```

The project web-server automatically sorts the jobs for the response according to the 'state_time_stamp' field of the job and implements a basic fifo scheduling algorithm.

The resource-daemons should now inspect the individual jobs in the response and decide to accept the job or reject it. When the resource decides to reject it, it should unlock this job so other resources can have a look at it. If it does decide to run the job, it should change the state of the job to 'running' and request the details of the job as explained below.

4) The request '***resource_request_job_details***';

This request is made to the project's web-server to retrieve the full information for a job identified through it's 'job_id'. The resource is again identified through the certificate and only if the resource is known and has a running lock on the job, this request is served. The 'job_id' is again POSTed together with the 'session_id' because running session with the resource is needed for this request to be able to serve. The response is very similar to the above described response, the extra information of the 'input', 'output', 'application' and the 'state' are now also returned. The 'input' and 'output' blobs are encoded into a hexadecimal (BinHex) notation so they can be transferred in the XML format over https. An example response is:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <resource> testresource </resource>
                <project> testproject </project>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
                <session_id> 1 </session_id>
                <job>
                        <job_id> 147 </job_id>
                        <application> testapp </application>
                        <state> running </state>
                        <target_resources> any </target_resources>
                        <owners> sjoerd, cyttorn </owners>
                        <read_access> any </read_access>
                        <state_time_stamp> 1259936661 </state_time_stamp>
```

```
                        <job_specifics> </job_specifics>
                        <input> CDEF9021569C8787E </input>
                        <output> </output>
                </job>
        </response
</LGI>
```

5) The request '***resource_lock_job***';

This request is made to the project's web-server to lock a job in the 'job_queue' table. The resource is identified through the certificate, the 'job_id' together with the 'session_id' is POSTed and the lock will only be set if the job has not been locked before by any other resource and if the job's 'target_resources' contains "any" or the requesting resource name. If the resource is unknown, this service is again denied and an error response is returned. A successful respond will be in the "***<lock> </lock>***" tags wrapped XML, with details of the lock:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <resource> testresource </resource>
                <project> testproject </project>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
                <session_id> 103 </session_id>
                <job_id> 104 </job_id>
                <lock>
                        <lock_id> 10020 </lock_id>
                        <job_id> 104 </job_id>
                        <lock_time> 127657651276 </lock_time>
                        <session_id> 103 </session_id>
                        <resource_id> 3 </resource_id>
                </lock>
        </response>
</LGI>
```

If the lock cannot be set, an error response is sent back, as described previously. The resource should now wait and try again later.

6) The request '***resource_unlock_job***';

This request is made to the project's web-server to unlock a previously set lock on a job. Again the resource is identified through the certificate and both the 'job_id' and the 'session_id' should be POSTed. The lock will only be removed if the resource is known and had originally locked the job, or a job was locked for the resource through the 'resource_request_work' request. The response of a successful call again is embedded into the "<lock> </lock>" tags:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <resource> testresource </resource>
                <project> testproject </project>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
```

```
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
                <session_id> 103 </session_id>
                <job_id> 104 </job_id>
                <lock>
                        <lock_id> 10020 </lock_id>
                        <job_id> 104 </job_id>
                        <unlock_time> 127657651276 </unlock_time>
                        <session_id> 103 </session_id>
                        <resource_id> 3 </resource_id>
                </lock>
        </response>
</LGI>
```

7) The request '***resource_update_job***';

This request is made to the server to alter some of the fields of a job. The job should be locked by the resource before this request is allowed. The resource should have a running session too. The 'job_id' and the 'session_id' should be POSTed together with fields that can be altered; 'state', 'target_resources', 'input', 'output' and 'job_specifics'. If any of these fields need changing, the new data should be POSTed. The server automatically adjusts the 'state_time_stamp' field if needed. The response of the server is the final data the job record will contain, as described by the 'resource_job_details' requests response. The 'input' and 'output' data POSTed should be first BinHexed and the server should un-BinHex them.

8) The request '***resource_submit_job***';

This request is made by a resource to insert a new job into the project's web-server database. Again a running session is needed for this request. The fields needed for this request are 'session_id', 'state', 'application', 'owners' and 'target_resources'. The other fields, 'read_access', 'job_specifics', 'input' and 'output', can also be POSTed if needed.

The server will check if any of the POSTed 'owners' are allowed to submit a job for the specific 'application'. If not, an error response is returned. The 'owners' credentials are checked against the 'users_denied', 'groups_denied', 'users_allowed' and 'groups_allowed' tables. Only the subset of the 'owners' that is allowed to submit a job will become part of the jobs 'owners'.

If the credentials are in order, the job is inserted into the database and automatically will be locked by the project's web-server for the resource, identified through the client certificate. The 'job_id' will be determined by and the 'state_time_stamp' will also be initialized by the server.

If the 'read_access' data was not POSTed, it will be set equal to the 'owners' by the server.

The response is again equal to the response obtained from the server in the 'resource_job_details' request. The resource can check this data, if needed, change fields by issuing an 'resource_update_job' request and finally do the

'resource_unlock_job' request.

9) The request '***resource_job_state***';

This request is made by the resource to get the state of a job, identified through the 'job_id' being POSTed. This request is only served if the known resource, identified through the certificate, is present in the 'target_resources' list of the job, or if this field contains the “any” resource. The response by the server is very similar to the one obtained by issuing the 'resource_job_details' request. The difference is that the 'input' and 'output' fields are now not reported and that the job need not be locked prior to the request. Neither does the resource need to have a running session for this request. An example of a valid response is thus:

```
<LGI>
	<CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
	<response>
		<resource> testresource </resource>
		<project> testproject </project>
		<project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
		<this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
		<job>
			<job_id> 147 </job_id>
			<application> testapp </application>
			<state> running </state>
			<target_resources> any </target_resources>
			<owners> sjoerd </owners>
			<read_access> any </read_access>
			<state_time_stamp> 1259936661 </state_time_stamp>
			<job_specifics> </job_specifics>
		</job>
	</response>
</LGI>
```

This request can be issued to monitor a change in state of a job by the resource. This can be used to detect an '***aborting***' state issued by an application interface. This request can also be used in the file transfer protocol, as will described below.

10) The request '***resource_request_resource_data***';

This request is made by the resource to get the certificate details of another resource. The compulsory field '***resource_name***' should be POSTed to be able to serve this request. The resource in again identified through it's certificate and is allowed to POST the 'project' field too. The certificate is reported in BinHexed format and no session is needed to serve this request. The response to this request looks like:

```
<LGI>
	<CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
	<response>
		<resource> testresource </resource>
		<project> testproject </project>
		<project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
		<this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
```

```
                <requested_resource_data>
                        <client_certificate> 0F1206AC5D ... 1B4D6E </client_certificate>
                        <resource_url> mark@ffwnc7003.leidenuniv.nl <resource_url>
                        <last_call_time> 1287876287 </last_call_time>
                </requested_resource_data>
        </response>
</LGI>
```

Job locking mechanism

Because in the LGI several resources can request work form the project's web-server, and decide for them selves, based on the 'job_specifics' field to perhaps accept the work for the application, a locking mechanism is needed. If a resource decides to inspect the job, another resource should not in the mean-time decide to run the job or else a race-condition is met.

The locking mechanisms in the LGI will be implemented per job and a resource is only allowed to work on a job entry of a project database if the resource has successfully locked that specific job prior to use. Moreover, the project's web server will refuse to serve 'resource_request_work' requests to the resource if a lock exists in the 'running_locks' table for that resource. If somehow a resource fails to unlock the job after a while, the project's web-servers will unlock the jobs that where locked and kill the session it had and the resource should setup a new session with the project's web-server by signing up again.

When the 'resource_request_work' is made, the project's server follows the following steps to ensure a correct and safe locking mechanism;

The requesting resource name is taken from the client-certificate it presents. It will check if the resource is registered and had a valid session running. If the resource was not registered or has no valid session, an error response is returned. Otherwise, a query is run on the 'running_locks' table for the project to see if there are any lock currently running for that resource in the current session. If so, again an error is returned.

If no locks were present on any jobs for this session and resource, the 'job_queue' table is queried for the specific resource for unlocked jobs (lock_state<=0), for the requested application, being in the 'queued' state. A limit is set to this request and results are sorted according to the 'state_time_stamp' field. Now for each job returned in this query in turn, the project's web-server should increase the 'lock_state' value. It should then query again for the same job's 'lock_state' and if it is equal to 1, the resource has successfully set the lock. If the value turns out to be 2 or more, a query to decrease the 'lock_state' of this job is issued. The job is then ignored and not put into the response. If the 'lock_state' returned 1 however, a new entry in the 'running_locks' table is inserted for this job, resource and session. It will set the current time as the 'lock_time' in the 'running_locks' table and the job is put into the response set.

When a request is made by a resource to unlock a job, the resource is first identified

and checked if it had a lock on the job in this session. If it did, the 'lock_state' of the job in the 'job_queue' is decreased first, before removing the corresponding entry in the 'running_locks' table.

Application interfacing with the project's web-server

In this section the communication between an 'application interface' and the project's web-servers is detailed upon. In general the URL used to communicate between a resource and the project's web-server can be different than the URL used by the 'application interfaces' that also communicate with the project's web-server.

Reasons are that for resources there is no need to check the user and group access control lists of the database (the 'user_allowed', 'user_denied', 'group_allowed', 'group_denied' tables), a resource is identified through it's certificate, usually resources interact differently and less frequent with the project's web-server than users do. Moreover, by logically separating the web-servers for 'resources' and 'application interfaces', extra scaling possibilities are introduced.

In general, all communications between the 'application interface' and the project's web-server need user credentials in the request. Also all communication is done in XML over https, using client certificates, signed by the LGI-CA, but now with common-names fields within the certificates being equal to user credentials. See the later section on certificates for details on this.

The user credentials need to be checked for each request and only jobs, for which the user has access to, through the 'owners' and 'read_access' fields, can be changed and inspected respectively.

Again all responses by the server are to be wrapped into the "*<LGI> <response> </response> </LGI>*" tags. If an error condition is met, the response will contain the usual "<error> </error>" tags embedded XML messages.

The following requests can be made by the application interface to the project web-server. They are listed under the 'interfaces' directory on the project server's main URL. For example "https://fwnc7003.leidenuniv.nl/LGI/interfaces/interface_submit_job.php".

1) The '***interface_submit_job***' request;

This request is done by an 'application interface ' to submit a job. The user is identified through the certificate's common-name field and by the POSTed 'user' and 'groups' fields. The POSTed 'user' field should equal the certificates common-name or otherwise service is denied. Further certificate checks that might be done are described below.

The other compulsory fields that also need to be POSTed in this request are the 'application' and the 'target_resources' fields. The 'input', the 'job_specifics', the 'owners' and 'read_access' fields are not compulsory but can be POSTed too.

If both the 'owners' and 'read_access' fields were not POSTed, the 'owners' field will be set to the 'users' field and the 'read_access' field will be set to the combined 'groups' and 'user' fields which both should be POSTed. If only the 'owners' field was POSTed, it is made sure that the POSTed 'user' is part of the 'owners' and the 'read_access' field is set to the new 'owners' field. If both the 'owners' and the 'read_access' are posted, it is made sure that the POSTed 'user' is part of the 'owners' and 'read_access' fields. If only the 'read_access' field was POSTed, the 'owners' field is set to the POSTed 'user' field and it is made sure that the 'read_access' field contains the POSTed 'user' field.

The 'target_resources' is checked to contain valid resources, known to the project, or the “any” resource. The 'input' field, when needed, is again POSTed in BinHex format, which is again un-BinHexed by the server.

A new job will created by the server, based on the above described values of POSTed fields. The default state of the job will be the 'queued' state. Clearly the other fields will be set by the server and a response to the 'application interface' will contain the POSTed details:

```
<LGI>
	<CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
	<response>
		<project> testproject </project>
		<project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
		<this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
		<user> mark </user>
		<groups> teras, cyttron </groups>
		<job>
			<job_id> 147 </job_id>
			<application> testapp </application>
			<state> queued </state>
			<target_resources> any </target_resources>
			<owners> sjoerd, cyttron </owners>
			<read_access> any, sjoerd, cyttron </read_access>
			<state_time_stamp> 1259936661 </state_time_stamp>
			<job_specifics> </job_specifics>
			<input> CDEF9021569C8787E </input>
		</job>
	</response>
</LGI>
```

2) The '***interface_job_state***' request;

This request is made by the 'application interface' to obtain information on the queue contained in the project's web-server database. Again the 'user' and 'groups' field should be POSTed and the common-name of the client certificate should match the POSTed 'user' field. This request has two ways of generating an response, depending on whether the 'job_id' field is POSTed or not.

In case the 'job_id' is not POSTed, the response will be a list of current jobs in the database, based on a selection by the possible POSTed 'application' and 'state' fields. Only jobs in which the POSTed 'user' or any of the 'groups' is part of the 'owners' or the 'read_access' field will be returned. In the response, the number of listed jobs will be wrapped into the "*<number_of_jobs> </number_of_jobs>*" tags. A valid response looks like:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <project> testproject </project>
                <user> mark </user>
                <groups> teras, cyttron </groups>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
                <number_of_jobs> 3 </number_of_jobs>

                <job number=1>
                        <job_id> 140 </job_id>
                        <state> queued </state>
                        <application> testapp </application>
                        <state_time_stamp> 1259926661 </state_time_stamp>
                 </job>

                <job number=2>
                        <job_id> 142 </job_id>
                        <state> running </state>
                        <application> testapp </application>
                        <state_time_stamp> 1259927661 </state_time_stamp>
                </job>

                <job number=3>
                        <job_id> 147 </job_id>
                        <state> finished </state>
                        <application> testapp </application>
                        <state_time_stamp> 1259936661 </state_time_stamp>
               </job>
        </response>
</LGI>
```

In this mode, it does not matter if the jobs reported have running locks or not.

If the request was made with a specific 'job_id' field POSTed, the response will be more elaborate for the specific job only:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <project> testproject </project>
                <user> mark </user>
                <groups> teras, cyttron </groups>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
                <number_of_jobs> 1 </number_of_jobs>
                <job_id> 140 <job_id>

                <job number=1>
                        <job_id> 140 </job_id>
                        <state> queued </state>
                        <application> testapp </application>
                        <target_resources> testresource </target_resources>
                        <owners> mark </owners>
                        <read_access> theor, sara, cyttron </read_access>
```

```
                        <state_time_stamp> 1259926661 </state_time_stamp>
                        <job_specifics>
                                <nprocs> 1 </nprocs>
                                <mem> 64M </mem>
                        </job_specifics>
                        <input> DCEF18982 </input>
                </job>
        </response>
</LGI>
```

If the possible POSTed 'job_id' is not present in the database, or the 'user', or any of his 'groups' is not part of the 'owners' or 'read_access' fields of the job, service is denied and an error response is returned. Also if the specific job has a lock running, an error response is sent back after the time-out period of 30 seconds has passed. During this time-out period the server waits for the lock to be cleared before a response is generated and sent back.

3) The ***'interface_delete_job***' request;

This request is made by an 'application interface' to remove a job from the project's web-server database. Again the 'user' and 'groups' fields should be POSTed and the 'user' field should match the certificate. Now also POSTed is the 'job_id' of the job that is to be killed or deleted. If the 'user', or the 'groups' are not part of the 'owners' field of the job with the POSTed 'job_id', service is denied and an error response is sent back.

If the user credentials match and the job with the POSTed 'job_id' is in the 'finished', 'aborted' or 'queued' state, the job can be safely removed from the database if no locks are present on the job. If the job was locked an error response is sent back after a certain time-out. During the time-out of 30 seconds, the server waits for the lock to be freed first.

If the job was in any other state than 'queued' or 'finished', the job is set into the 'aborting' state and not removed from the database, only if no lock is active on the job. The 'aborting' state allows the resource that is currently dealing with the job, to detect the abort signal. The resource will then set the job into the 'finished' or 'aborted' state when possible. Whether the resource can abort a job immediately is highly dependent on the application and the resource it is running on. Again, if the job is still locked after the preset time-out period of 30 seconds, an error response is sent back.

If the job was successfully removed from the database, or has been set into the 'aborting' state, the response of the servers is similar to the response obtained by the 'user_job_state' request described above:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <user> mark </user>
                <groups> teras cyttron </groups>
                <project> testproject </project>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
```

```
</this_project_server>
                <job>
                        <job_id> 140 </job_id>
                        <state> aborting </state>
                        <application> testapp </application>
                        <target_resources> testresource </target_resources>
                        <owners> mark </owners>
                        <read_access> theor, sara, cyttron </read_access>
                        <state_time_stamp> 1259926661 </state_time_stamp>
                        <job_specifics>
                                <nprocs> 1 </nprocs>
                                <mem> 64M </mem>
                        </job_specifics>
                        <input> DCEF18982 </input>
                 </job>
        </response>
</LGI>
```

This allows the 'application interface' to record or log jobs if desired.

4) The '***interface_project_server_list***' request;

This request is made to get a list of project web-servers for a specific project. The 'user' and 'groups' field are POSTed and compared to the clients-certificates common-name. If they do not match an error response is returned and service is denied. If the credentials are met, a valid response will contain a list of project servers:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <user> mark </user>
                <groups> teras cyttron </groups>
                <project> testproject </project>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/testmasterserver
</project_master_server>
                <this_project_sever> https://fwnc7003.leidenuniv.nl/testslaveserver
</this_project_server>
                <number_of_slave_servers> 2 </number_of_slave_servers>
                <project_sever number=1> https://fwnc7003.leidenuniv.nl/testslaveserver
</project_server>
                <project_sever number=2>
https://fwnc7003.leidenuniv.nl/testslaveserver2 </project_server>
        </response>
</LGI>
```

The 'application interface' can now choose what project web-server to communicate to to submit or query about jobs.

Certificates

Within the LGI heavy use is made of x509 server- and client-certificates. This part of the document details on the 'CommonName' field of these certificates and how they are used within the LGI.

For certificates belonging to 'resources' the 'CommonName' contains possibly two fields separated by a ';'. The first field specified the 'resource' unique name which is

also present in the 'active_resources' table. The second field is a comma separated field of all the projects this resource has access to. These fields within the certificate always take precedence to any other access rules implemented in the project server or fields posted to the project server. Resources always supply a valid client-certificate to the project servers.

For certificates belonging to users or 'application interfaces', the 'CommonName' possibly contains three fields separated by a ';'. The first field is the name of the user that accesses the project server through the application interfaces API. The possible present second field is a comma separated field with all the groups the user belongs to. The possible third field is also a comma separated field with all the projects the user is allowed to interface to. Application interface's always supply a valid user's client-certificate to the project servers.

Project servers communication API

In this part of the document the API that deals with the communication between two or more project servers within a single project is detailed upon. This API has been developed to allow for project servers to exchange 'updates' to keep all servers within a project synchronized.

Each project server within a project is identified as being a resource that has the 'project_server' field set to non-zero in the 'active_resources' table of all the project servers. A project server is therefore also identified through a 'resource' client-certificate signed by the LGI-CA.

Updates are queries that can be executed on the project server's database. Each update is identified by an integer 'version' number and only sequential updates are allowed. Each update will be logged into the 'updates' table.

Again all responses by the server are to be wrapped into the "*<LGI> <response> </response> </LGI>*" tags. If an error condition is met, the response will contain the usual "*<error> </error>*" tags embedded XML messages.

The following requests can be made by any other project server within the project to the project web-server. They are listed under the 'servers' directory on the project server's main URL. For example "https://fwnc7003.leidenuniv.nl/LGI/servers/server_get_update.php".

1) The '***server_get_update***' request;

This request allows for any project server to inquire this server for the presence of possible updates. The requesting server is identified through it's certificate common name. Only if the requesting server is listed as a project server, this request can be

met. An error response is returned otherwise.

The compulsory '***version***' and 'project' fields are POSTed and identify the requesting server's latest update. If the server has any updates with a higher version number in the 'updates' table, it will report the first sequentially applicable update the requesting server should apply or log into it's own 'updates' table. A valid response looks like:

```
<LGI>
	<CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
	<response>
		<project> testproject </project>
		<project_master_sever> https://fwnc7003.leidenuniv.nl/LGI
</project_master_server>
		<this_project_server> https://fwnc7003.leidenuniv.nl/LGI
</this_project_server>
		<requesting_project_server> https://fwnc7003.leidenuniv.nl/LGI2 </
requesting_project_server>
		<requesting_project_server_version> 0
</requesting_project_server_version>
		<update>
			<updates> 2 </updates>
			<update_version> 1 </update_version>
			<target_servers> any </target_servers>
			<update_query> 0002020010002AB3 </update_query>
		</update>
	</response>
</LGI>
```

If there are no relevant updates, the above response will contain "*<updates> 0 </updates>*" and the latest version number encountered in the database is reported within the "*<update_version> </update_version>*" tags.

2) The '***server_run_update***' request;

This request allows for any other project server within the project to 'push' an update to this server. For this to work the '**version**', '***servers***' and '***update***' fields together with the 'project' fields should be POSTed. If the server has any updates higher than the POSTed version, it will ignore this update being pushed and return it's highest version number within the "*<update_version> </update_version>*" tags. If this server should sequentially apply or log this update, it will do so. If this server has to apply other updates first, an update cycle is started. If the 'version', 'servers' and 'update' fields were not POSTed, also an update cycle is started.

A valid response, when an update is being pushed, looks like:

```
<LGI>
	<CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
	<response>
		<project> testproject </project>
		<project_master_sever> https://fwnc7003.leidenuniv.nl/LGI
</project_master_server>
		<this_project_server> https://fwnc7003.leidenuniv.nl/LGI
</this_project_server>
		<project_server> https://fwnc7003.leidenuniv.nl/LGI2 </
requesting_project_server>
		<update_version> 1 </update_version>
		<target_servers> any </target_servers>
		<update_query> 0102030405011 </update_query>
```

```
                <update>
                        <update_version> 1 </update_version>
                </update>
        </response>
</LGI>
```

During an update cycle, each of the marked servers is requested to report back any possible update. For this a POST to the 'server_get_update' API routine is performed on each server. If all updates have been applied and all project servers have been polled, the update cycle is finished. A valid response after such an update cycle looks like:

```
<LGI>
        <CA_certificate> http://www.LGI.org/LGI-CA.crt </CA_certificate>
        <response>
                <project> testproject </project>
                <project_master_sever> https://fwnc7003.leidenuniv.nl/LGI
</project_master_server>
                <this_project_server> https://fwnc7003.leidenuniv.nl/LGI
</this_project_server>
                <project_server> https://fwnc7003.leidenuniv.nl/LGI2 </
requesting_project_server>
                <update>
                        <update_version> 1 </update_version>
                </update>
        </response>
</LGI>
```

Resource Daemon

In this section a more detailed description of the 'resource daemon' is given, which by default runs as a non privileged user on the resource. The resource daemon is configured by a configuration file. This file has also an XML format and the workings of the resource daemon is best described using the following example configuration:

```
<LGI>
        <ca_certificate_file> ../certificates/LGI+CA.crt </ca_certificate_file>
        <resource>
                <resource_name> mark@laptop </resource_name>
                <resource_url> mark@laptop </resource_url>
                <resource_certificate_file> ../certificates/laptop.crt
</resource_certificate_file>
                <resource_key_file> ../certificates/laptop.key </resource_key_file>
                 <run_directory> ./runhere </run_directory>

                <owner_allow> <any> 10 </any> </owner_allow>
                <owner_deny> nobody </owner_deny>

                <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>

                        <number_of_applications> 1 </number_of_applications>

                        <application number=1>
                                <application_name> hello_world </application_name>
```

```
                                <owner_allow> <any> 1 </any> </owner_allow>
                                <owner_deny> nobody </owner_deny>

                                <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>
```

The resource daemon reads the the above illustrated configuration from an input file. Before running any further, the daemon verifies all the tags of the configuration. Files referred to should be readable and exist, directories specified should be readable and also exist, all the above illustrated tags should be present and so on.

After the daemon has verified the configuration, the configuration is stored in memory and used as long as the daemon is active. Changes to the configuration file can be made active by restarting the daemon. To stop the daemon, when it is running in the background, a simple 'kill pid' command can be used to let the signal handler of the daemon to gracefully shut down the daemon. Any open sessions to any server are then guaranteed to be finished before the daemon is stopped.

The daemon will also generate a log. The logging level of the daemon can be supplied at the command line when the daemon is started. At the same time, the configuration file is supplied together with the option whether or not to 'daemonize' and run in the background. The daemon can run as any user on the system and it is preferred to run the daemon as an unprivileged user and not as root. This also allows one to setup a daemon on a resource for which one has no root privileges. Below are listed the options that the daemon supports:

```
./LGI_daemon.x [options] configfile

options:

-d          daemonize and run in background.
-q          log only critical messages.
-n          log normal messages. this is the default.
-v          also log debug messages.
-vv         also log verbose debug messages.
-l file     use specified logfile. default is to log to standard output.
```

In general, when the daemon has verified the configuration and is ready to run, the '***run_directory***' directory is scanned through for cached jobs. This run directory contains a tree of projects and applications and within each branch, the jobs for that

project and that application is cached on disk. Each job will have a unique ***job directory*** in which the complete state of the job is stored.
The state of a job in its corresponding job directory is described by several files, all beginning with a 'LGI_' prefix. Each such file contains exactly the same information that is also stored in the projects web-server's database for that job or contains an exact copy of the scripts that should be used for that application, as was specified in the configuration file. Moreover, all the 'LGI_*' files, except for the LGI_output file, each has a corresponding '.hash' file. In that file the hash fingerprint in hexadecimal format is stored. If any of the files or scripts are corrupted, edited or changed, the hash will make sure that this is detected. Scripts, for which the hash does not match, will not be run. Job directories that contain a file for which the hash does not match, will not be monitored.

At the start of the daemon the run directory is scanned through for cached jobs. Any job directory encountered and verified to be valid through the hashes, will be included into the set of jobs that are to be monitored by the daemon. When the daemon runs, every two minutes the state of the jobs that are to be monitored, are checked by contacting the correct job project server. If the job was issued on a slave server, only that slave server is contacted by using the 'resource_job_state' function. If, during this 'update cycle', the state in the database was changed, the job details are requested and the job directory on disk is synchronized to the result of the post.

After having updated all jobs on disk, each job will be inspected through the '***job_check_XXXX***' scripts. These scripts were provided in the configuration file, but are now also stored, with a hash, in the job directory too. This allow the successful completion of jobs even after the configuration file has changed the scripts for any of the applications.

In general the '***job_check_running_script***' script is started to check if the job is still running. This script should return an exit code of zero to signal that the job is still running. If the job was not found to be running, the '***job_check_finished_script***' is started to check if the job was finished. It also should return an exit code of zero if the job was finished. If the job was found to be running, the 'state' of the job is checked. If the 'state' of the job was changed to 'aborting', the '***job_abort_script***' script is run. If then the job_abort_script returns an exit code of zero, the job is considered to be aborted and the job details are again posted to the project web-server through the 'resource_update_job' function. After the successful post, the job is removed from the list of jobs to be monitored and the corresponding job directory is deleted. The job will end up in the 'aborted' state in the database on the project's web-server.

If the job was not found to be finished or running, the run sequence of the job will be started; first the '***job_prologue_script***' is started and if that script returns an exit code of zero, the '***job_run_script***' is started on the background.

If the job was found to be finished, because the 'job_check_finished_script' returned an exit code of zero, the '***job_epilogue_script***' script is started. If the 'job_epilogue_script' script returns an exit code of zero, the job details will be posted

to the server. Again after a successfull post, the job directory is removed, the job will not be monitored again and the job will end up in the 'finished' state on the project's web-server's database.

The use of these 'job_xxxx_scripts' allows the LGI administrator to fully configure the way the application should be started. Moreover, these scripts can access any of the job details through the 'LGI_xxxxx' files also stored in the job directory. Because these scripts are copied and hashed into the job directory, a change in the configuration, or in the job scripts, will not affect any jobs prior to these changes.

Apart from doing the above described 'job update' cycle every two minutes, the daemon also performs 'request work' cycles every ten minutes or so. During this cycle all the projects and applications that are specified in the configuration file are inspected. The '***check_system_limits_script***' script is used to see if there is a system wide limit reached to running a job of this application. The script returns zero if no limit is present and work can be requested by posting to the 'resource_request_work' function on the project's web-server.

All jobs offered by any of the project's web-servers are inspected and checked against the limits imposed by the '***owner_allow***' and '***owner_deny***' tags specified in the configuration file. If any of the jobs 'owners' was specified in any of the 'owners_deny' tags, the job is not accepted. If a limit was specified for any of the 'owners' in the 'owner_allow', by wrapping the maximum number of jobs allowed for that owner in tags of the owner: “<owner_allow> <mark> 2 </mark> <any> 1 </any> </owner_allow>”, the number of jobs currently being monitored is checked. If the set limit would be exceeded, the job is ignored again. Only if none of the owners would exceed any limits, the job will be checked further by running the '***job_check_limits_script***' script. If this script returns an exit code of zero, no limit is present for the job and the job will be accepted. Only now a job directory will be created for it, the job details will be stored there and the job will be monitored and updated through the 'job update' cycle of the daemon.

For projects with multiple project (slave) servers, a list of servers is compiled first by the daemon during the 'request for work' cycle. Each of these servers is then requested to give work as was described above. This allows for a work balancing. The list of project (slave) servers is obtained by first signing up at the server specified by the 'project_master_server' tags in the configuration file.

The actual daemon code is implemented in C++ and makes use of the Standard Template Library (STL) and the cURL library. The source code can be found in the 'daemon/src' directory. There a 'Makefile' is present and by issuing the 'make' command, the code will be compiled. One can also run 'make clean' to clean up any makes performed.

In the 'daemon/src' directory the following files can be found:

```
binhex.cpp:    contains the algorithms to convert binary to hexadecimal strings and
back.
```

```
csv.cpp:       contains the comma separated value parser for strings.

hash.cpp:      contains the hash algorithm for strings.

xml.cpp:       contains the xml parser algorithms for strings.

logger.cpp:    contains the advanced logging facilities.

resource_server_api.cpp:     contains the 'project server' resource api functionality
in a class.

daemon_configclass.cpp:      contains the class to read in and parse the
configuration file of the daemon.

daemon_jobclass.cpp:         contains the job directory handler class for jobs.

daemon_mainclass.cpp:        contains the daemon main scheduling algorithms in a
class.

daemon_main.cpp:             is the main daemon code that uses the above mentioned
classes.
```

Furthermore, an example configuration has been setup too. Check the 'LGI.cfg' configuration file and the 'hello_world_scripts' scripts for that example.

File transfers and storage protocol

A simple basic storage can be implemented through a directory based repository on the machines file-system using the following transfer protocol based on 'scp', using private-public session key pairs. This protocol will now be explained in more detail, as it automatically servers as an example for other file transfer protocol based implementations;

The resource, called the ***initiator***, submitting a '***file transfer***' job into the project's web-server, because it wants to send or receive a file, starts off by generating a public-private session key pair for the file transfer. It will use the 'ssh-keygen' command for this and then temporarily store the generated public session key into it's 'authorized_keys' file. After that, it encrypts the generated private session key with the known public key of the resource to which it wants to send the file too or receive the file from. This resource is called the ***target***. The initiator can retrieve that information from the project's web-servers. It will then submit the 'file transfer' job into any of the project's web-servers. It does this with the encrypted private session key stored into the request.

If the file initially resides on the initiator and needs to be transferred to the target (the file is being '***send***' in the request), the transfer request will already contain the size of the file and the complete URL of the file to be send (i.e. 'somers@teras.sara.nl:/home/somers/LGI/Repository/Project/Owner/file_to_send_and_store'). If the file transfer is a 'read' request, only the 'file name' or unique handle of the file, together with the ***owners*** requesting the data, are part of the request. In this framework the owner of a file can be a specific user of a project, or a group of users of

a project.

After the 'file transfer' job has been queued by the initiator, it will start to poll that same project server's database to see if this 'file transfer' job has changed state into the '***transferred***' state or the '***request rejected***' state.

The target resource, also polling the web-server, detects the '***queued***' 'file transfer' job in the project's web-server. It alone can decrypt the session key stored in the request and will do so to be able to start the scp command to retrieve from or send the file to the initiator whom, at this point, is still waiting for the file transfer to finish.

If the transfer job was a 'send' request from the initiator, the file will now be stored permanently on the target resource and thus the size is first checked, together with the credentials of the owner of the file. The resource can decide not to store and accept this file because of possible 'owner' limits set on the resource for a specific project. If it is a 'read' request, the target will also check the credentials and see if the request can be met.

If the target resource decides to accept the transfer, the scp command is issued and afterwards the md5 of the file is computed. After the transfer, whether it was successful or not, the target resource communicates to the project's web-server and flags the file transfer job as '***transferred***' and reports the md5 of the file into the same request. It will now wait until the the transfer is accepted and acknowledged by the initiator.

If the target, however, decided to reject this 'file transfer' job, it changes the state of the job to 'request rejected' and starts again polling the web-servers for new 'file transfer' jobs.

The initiator now detects that the file transfer is over.

If the transfer request was rejected by the target, the initiator stops with the protocol.

If the transfer was however accepted and started, it can now check the md5 of the file received or sent by the target resource to see if the file was transferred successfully. It can re-initiate the transfer again by resetting the state of the transfer job to '***queued***' again if the transfer was unsuccessful.

If the transfer was a successful 'receive' request, the data was transferred from the target to the initiator. The initiator now changes the state of the 'file transfer' request to '***transfer accepted***' after inserting a '***running***' 'file storage' job into the web-server if the data now resides permanently on the initiator. This inserted 'file storage' job takes over the owner of the file and the full URL to retrieve the file later on is also stored, together with the md5 of the file. If the transfer was a 'copy' and the data only resides temporarily on the initiator, no such 'file storage' job is inserted.

If the transfer was a successful 'send', the initiator just changes the state of the request

to 'transfer accepted'.

It will now start polling the request again until it changes state to '***finished***'.

The target resource, in the mean time, detects the change of state of the request from 'transferred' to 'transfer accepted' or 'queued' if the transfer had failed.

If the transfer was not successful, the target resource removes the stale data, in case this was a 'send' request issued by the initiator, and starts again polling the web-servers for new 'file transfer' jobs.

If the transfer was successful and it was a 'send' request by the initiator, a new 'file storage' job is inserted into the project's web-server to keep track of the file. This file storage job takes over the owner of the file and the storage job is set to the 'running' state indicating that the file now permanently resides on this specific target resource. In the storage job, again, the full URL to retrieve the file later on is also stored, together with the md5 of the file.

If, however, it was a 'receive' request issued by the initiator, the target needs to check if the request was a 'copy' or a 'move'. In case it was a 'move' it should check if it has a running 'file storage' job on the file. If so, it should now be put into the 'finished' state and the file should be removed from the repository on the target.

Now the target changes the state of the request to 'finished' and starts polling for new 'file transfer' jobs again.

The initiator detects that the request has changed state into 'finished'.

If the transfer was a 'send' request, the initiator can now remove the file if the request was a 'move'. It also checks if there was a 'running' 'file storage' job on this file and if so, changes that state to 'finished'. If the 'send' was a copy, no such things are done.

This protocol works if both the initiator and the target are know a priori. If, however, the initiator does not care where exactly the file will be stored, the initial file transfer request job will not contain a specific target resource and the session key cannot be encrypted with the public key of the target resource. In this case the resource accepting the transfer reports to the project's web-server by setting the target identifier of the job to itself and the state of the 'queued' job into the '***accepted***' state. The initiator detects this change of state and now knows to which target resource it should address the 'file transfer' request. It will restart the 'file transfer' job, but now with the correctly encrypted session key in the request and by setting the state again to '***queued***'.

This protocol automatically generates a full back-log of all file transfers between the resources and a state-log of all the currently stored files per project and owner. Moreover, all the needed synchronizations and error detections are also automatically built into this secure protocol.

File transfer program

...
...
...

Example libcURL program

This program shows how to do a post request to a web-server using client-server certificates in C using libcURL:

```
#include <stdio.h>
#include <curl/curl.h>

const char *pClientCertFile = "CLIENT.crt";
const char *pClientPrivateKeyFile = "CLIENT.key";
const char *pCACertFile = "CA.crt";
const char *pRequestedData = "field=SSL_CLIENT_VERIFY";

int main(void)
{
  CURL *curl;
  CURLcode result;

  curl = curl_easy_init();
  if( curl ) {

    curl_easy_setopt( curl, CURLOPT_URL,
"https://fwnc7003.leidenuniv.nl/Dump/test_cert.php" );

    // for use with client certificates ...
    curl_easy_setopt( curl, CURLOPT_SSLCERT, pClientCertFile );
    curl_easy_setopt( curl, CURLOPT_SSLKEY, pClientPrivateKeyFile );

    // verify server certidicate ...
    curl_easy_setopt( curl, CURLOPT_CAINFO, pCACertFile );
    curl_easy_setopt( curl, CURLOPT_SSL_VERIFYPEER, 1 );
    curl_easy_setopt( curl, CURLOPT_SSL_VERIFYHOST, 2 );

    // post the request for the common name
    curl_easy_setopt( curl, CURLOPT_POSTFIELDS, pRequestedData );

    result = curl_easy_perform( curl );

    curl_easy_cleanup( curl );

  }
  return 0;
}
```

The corresponding php script on the web-server was:

```
<?php
 if( $_POST['field'] )
 {
  $field=$_POST[ 'field' ];
  $data=$GLOBALS[ 'HTTP_SERVER_VARS' ][ $field ];

  echo "A post was done for field '".$field."'\n";
  echo "Result: '".$data."'\n";
```

```
 }
 else
  var_dump($GLOBALS);
?>
```

and the output was:

```
A post was done for field 'SSL_CLIENT_VERIFY'
Result: 'SUCCESS'
```

The basic XML parser in C

The below code is a basic example of how to implement the needed XML parsing capabilities within the resource-daemons:

```
#include <string>
#include <iostream>

using namespace std;

/* ------------------------------------------------------------ */

string Parse_XML( string XML, string Tag, string &Attributes )
{
 int XMLLength = XML.length();
 int TagLength = Tag.length();
 int Index, BeginTagStart, BeginTagEnd,
     AttributeStart, AttributeEnd,
     EndTagStart, EndTagEnd, FoundTagLength;
 string Empty( "" );
 string FoundTag, FoundAttr;

 if( XMLLength <= 0 ) return( Empty );
 if( TagLength <= 0 ) return( Empty );

 for( Index = 0; Index < XMLLength; ++Index )
  if( XML[ Index ] == '<' )
  {

   BeginTagEnd = BeginTagStart = Index + 1;

   while( BeginTagEnd < XMLLength  && XML[ BeginTagEnd ] != ' ' && XML
[ BeginTagEnd ] != '>' ) ++BeginTagEnd;

   if( BeginTagEnd >= XMLLength ) return( Empty );

   FoundTag = XML.substr( BeginTagStart, BeginTagEnd - BeginTagStart );
   FoundTagLength = BeginTagEnd - BeginTagStart;

   if( XML[ BeginTagEnd ] == ' ' )
   {
    AttributeStart = BeginTagEnd + 1;

    while( AttributeStart < XMLLength  && XML[ AttributeStart ] == ' ' )
++AttributeStart;

    if( AttributeStart >= XMLLength ) return( Empty );

    AttributeEnd = AttributeStart + 1;

    while( AttributeEnd < XMLLength  && XML[ AttributeEnd ] != '>' ) ++AttributeEnd;

    if( AttributeEnd >= XMLLength ) return( Empty );

    FoundAttr = XML.substr( AttributeStart, AttributeEnd - AttributeStart );
   }
   else
```

```
    AttributeStart = AttributeEnd = BeginTagEnd;

   EndTagStart = XML.find( "</" + FoundTag + ">", AttributeEnd + 1 ) + 2;

   if( EndTagStart >= XMLLength ) return( Empty );

   EndTagEnd = EndTagStart + FoundTagLength;

   if( EndTagEnd >= XMLLength ) return( Empty );

   if( FoundTag == Tag )
   {
    Attributes = XML.substr( AttributeStart, AttributeEnd - AttributeStart );
    return( XML.substr( AttributeEnd + 1, EndTagStart - AttributeEnd - 3 ) );
   }

   Index = EndTagEnd;
  }

 return( Empty );
}

/* ------------------------------------------------------------ */

int main( void )
{
 string Attrs;
 string tstxml1 = "<testtag>first value</testtag>";
 string tstxml2 = "<testtag><testtag2>second value</testtag2></testtag>";
 string tstxml3 = "<testtag>first value</testtag><testtag><testtag2>second
value</testtag2></testtag>";
 string tstxml4 = "<testtag name=1>first value</testtag>";
 string tstxml5 = "<testtag name=2><testtag2 name=3 looks=1>second
value</testtag2></testtag>";

 cout << "1 DATA: '" << Parse_XML( tstxml1, "testtag", Attrs ) <<  "' ATTR: '" <<
Attrs << "'" << endl;
 cout << "2 DATA: '" << Parse_XML( tstxml2, "testtag", Attrs ) <<  "' ATTR: '" <<
Attrs << "'" << endl;
 cout << "3 DATA: '" << Parse_XML( Parse_XML( tstxml2, "testtag", Attrs ), "testtag2",
Attrs ) <<  "' ATTR: '" << Attrs << "'" << endl;
 cout << "4 DATA: '" << Parse_XML( tstxml2, "testtag2", Attrs ) <<  "' ATTR: '" <<
Attrs << "'" << endl;
 cout << "5 DATA: '" << Parse_XML( tstxml3, "testtag", Attrs ) <<  "' ATTR: '" <<
Attrs << "'" << endl;
 cout << "6 DATA: '" << Parse_XML( tstxml3, "testtag2", Attrs ) <<  "' ATTR: '" <<
Attrs << "'" << endl;
 cout << "7 DATA: '" << Parse_XML( tstxml4, "testtag", Attrs ) <<  "' ATTR: '" <<
Attrs << "'" << endl;
 cout << "8 DATA: '" << Parse_XML( tstxml5, "testtag", Attrs ) <<  "' ATTR: '" <<
Attrs << "'" << endl;
 cout << "9 DATA: '" << Parse_XML( Parse_XML( tstxml5, "testtag", Attrs ), "testtag2",
Attrs ) <<  "' ATTR: '" << Attrs << "'" << endl;

 return( 0 );
}
```

The output of this program is:

```
1 DATA: 'first value' ATTR: ''
2 DATA: '<testtag2>second value</testtag2>' ATTR: ''
3 DATA: 'second value' ATTR: ''
4 DATA: '' ATTR: ''
5 DATA: 'first value' ATTR: ''
6 DATA: '' ATTR: ''
7 DATA: 'first value' ATTR: 'name=1'
8 DATA: '<testtag2 name=3 looks=1>second value</testtag2>' ATTR: 'name=2'
9 DATA: 'second value' ATTR: 'name=3 looks=1'
```