NORDUGRID

NORDUGRID-TECH-20

5/2/2009

LIBARCCLIENT

A Client Library for ARC

Mattias Ellert[*]

Iván Márton[†]

Bjarte Mohn[‡]

Gábor Rőczei[§]

[*]mattias.ellert@fysast.uu.se

[†]martoni@niif.hu

[‡]bjarte.mohn@fysast.uu.se

[§]roczei@niif.hu

Contents

Chapter 1

Preface

This document describes from a technical viewpoint a plugin based client library for the new Web Service (WS) based Advanced Resource Connector [?] (ARC) middleware. The library consists of a set of C++ classes for

- handling proxy, user and host certificates,
- performing resource discovery and information retrieval,
- job submission and management and
- data handling.

All capabilities are enabled for three different grid flavours (Production ARC, ARC1 and gLite [?]) through a modular design using plugins specialized for each supported middleware. Future extensions to support additional middlewares involve plugin compilation only i.e. no recompilation of main libraries or clients is necessary.

Using the library, a set of command line tools have been built which puts the library's functionality at the fingertips of users. While this documentation will illustrate how such command line tools can be built, the main documentation of the command line tools is given in the client user manual [?].

In the following we will give a functionality overview in Section 2 while all technical details will be given in Section 3. Section 4 will through examples show how command line interfaces can be built upon the library.

Chapter 2

Functionality Overview

The new libarcclient makes extensive use of plugins for command handling. These plugins are handled by a set of higher level classes which thus are the ones to offer the plugin functionality to external calls. In this section an overview of the library's main functionality is given which also introduces the most important higher level classes.

2.1 Resource Discovery and Information Retrieval

With the increasing number of grid clusters around the world, a reliable and fast resource discovery and information retrieval capability is of crucial importance for a user interface. The new libarcclient resource discovery and information retrieval component consists of three classes; the `TargetGenerator`, the `TargetRetriever` and the `ExecutionTarget`. Of these the `TargetRetriever` is a base class for further grid middleware specific specialization (plugin).

Figure 2.1 depicts how the classes work together in a command chain to discover all resources registered with a certain information server. Below a description of each step is given:

1. The `TargetGenerator` takes three arguments as input. The first argument is a reference to a `UserConfig` object containing a representation of the contents of the user's configuration file. The second and third arguments contain lists of strings. The first list contains individually selected and rejected computing services, while the second list contains individually selected and rejected index servers. Rejected servers and services are identified by that its name is prefixed by a minus sign in the lists. The name of the servers and services should be given either in the form of an alias defined in the `UserConfig` object or as the name of its grid flavour followed by a colon and the URL of its information contact endpoint.

2. These lists are parsed through alias resolution before being used to initialize the complete list of selected and rejected `URL`s pointing to computing services and index servers.

3. For each selected index server and computing service a `TargetRetriever` plugin for the server's or service's grid flavour is loaded using the ARC loader. The `TargetRetriever` is initialized with its `URL` and the information about whether it represents a computing service or an index server.

4. An external call is received calling for targets to be prepared. The call for targets is processed by each `TargetRetriever` in parallel.

5. A `TargetRetriever` representing an index server first tries to register at the index server store kept by the `TargetGenerator`. If allowed to register, the index server is queried and the query result processed. The `TargetGenerator` will not allow registrations from index servers present in its list of rejected index servers or from servers that have already registered once. Index servers often register at more than one index server, thus different `TargetRetriever`s may discover the same server.

6. If while processing the query result the `TargetRetriever` finds an other registered index server or a registered computing service it creates a new `TargetRetriever` for the found server or service and forwards the call for targets to the new `TargetRetriever`.

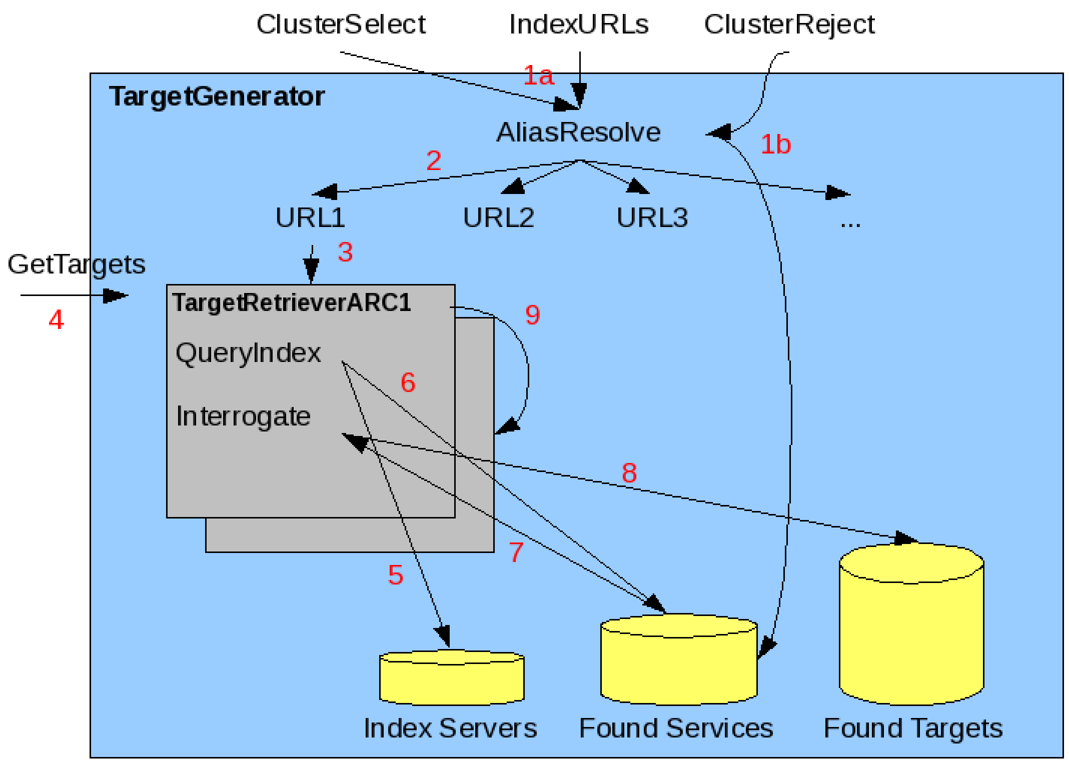

Figure 2.1: Diagram depicting the resource discovery and information retrieval process

7. A `TargetGenerator` representing a computing service first tries to register at the service store kept by the `TargetGenerator`. If allowed to register, the computing server is queried and the query result processed. The `TargetGenerator` will not allow registrations from computing services present in its list of rejected computing services or from service that have already registered once. Computing services often register at more than one index server, thus different `TargetRetriever`s may discover the same service.

8. When processing the query result the `TargetRetriever` will create an `ExecutionTarget` for each queue found on the computing service and collect all possible information about them. It will then store the `ExecutionTarget` in the found targets store kept by the `TargetGenerator` for later usage (e.g. status printing or job submission).

2.2 Job Submission

Job submission starts with the resource discovery and target preparation as outlined in the Section 2.1. Not until a list of possible targets (which allows the user) is available is the job description read in order to enable bulk job submission of widely different jobs without having to reperform the resource discovery. In addition to the classes mentioned above the job submission makes use of the `Broker`, `JobDescription` and `Submitter` classes. The `Submitter` is base class for further grid middleware specific specialization (plugin) similarly to the `TargetRetriever`.

Figure 2.2 shows a job submission sequence and below a description of each step is given:

1. The `TargetGenerator` has prepared a list of `ExecutionTargets`. Depending on the `URLs` provided to the `TargetGenerator` the list of found `ExecutionTargets` may be empty or contain several targets. Targets may even represent more than one grid flavour. The list of found targets are given as input to the `Broker`.

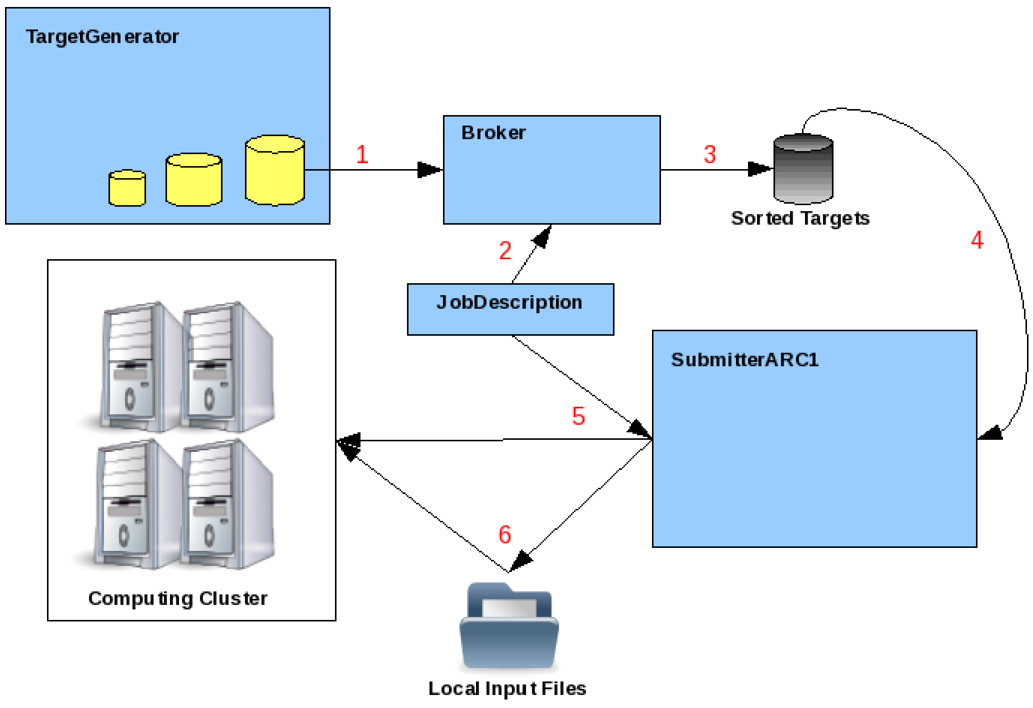

Figure 2.2: Diagram depicting the submission of a job to a computing service.

2. In order to rank the found services (`ExecutionTargets`) the `Broker` needs detailed knowledge about the job requirements, thus the `JobDescription` is passed as input to the brokering process.
3. The `Broker` outputs a ordered list of `ExecutionTargets` according to the provided `JobDescription`.
4. Each `ExecutionTarget` has a method to return a specialized `Submitter` which is capable of submitting jobs to the service it represents. The best suitable `ExecutionTarget` for the job is asked to return a `Submitter` for job submission.
5. The `Submitter` takes the `JobDescription` as input and uploads it to the computing service.
6. The `Submitter` identifies local input files from the `JobDescription` and uploads them to the computing service.

2.3 Job Management

Once a job is submitted job related information (job identification string, cluster etc) is stored in a local XML file which hosts similar information for all active jobs. This file may contain jobs running on completely different grid flavours, and thus job management should be handled using plugins similar to resource discovery and job submission. The job managing plugin is called the `JobController` and it is supported by the `JobSupervisor` and `Job` classes.

Figure 2.3 shows how the three different classes work together and below a step by step description is given:

1. The `JobSupervisor` takes four arguments as input. The first argument is a reference to a `UserConfig` object containing a representation of the contents of the user's configuration file. The second is a list of strings containing job identifiers and job names, the third is a list of strings of clusters to select or reject (in the same format as described for the `TargetGenerator` above). The last argument is the name of the file containing the local information about active jobs, hereafter called the joblist file.

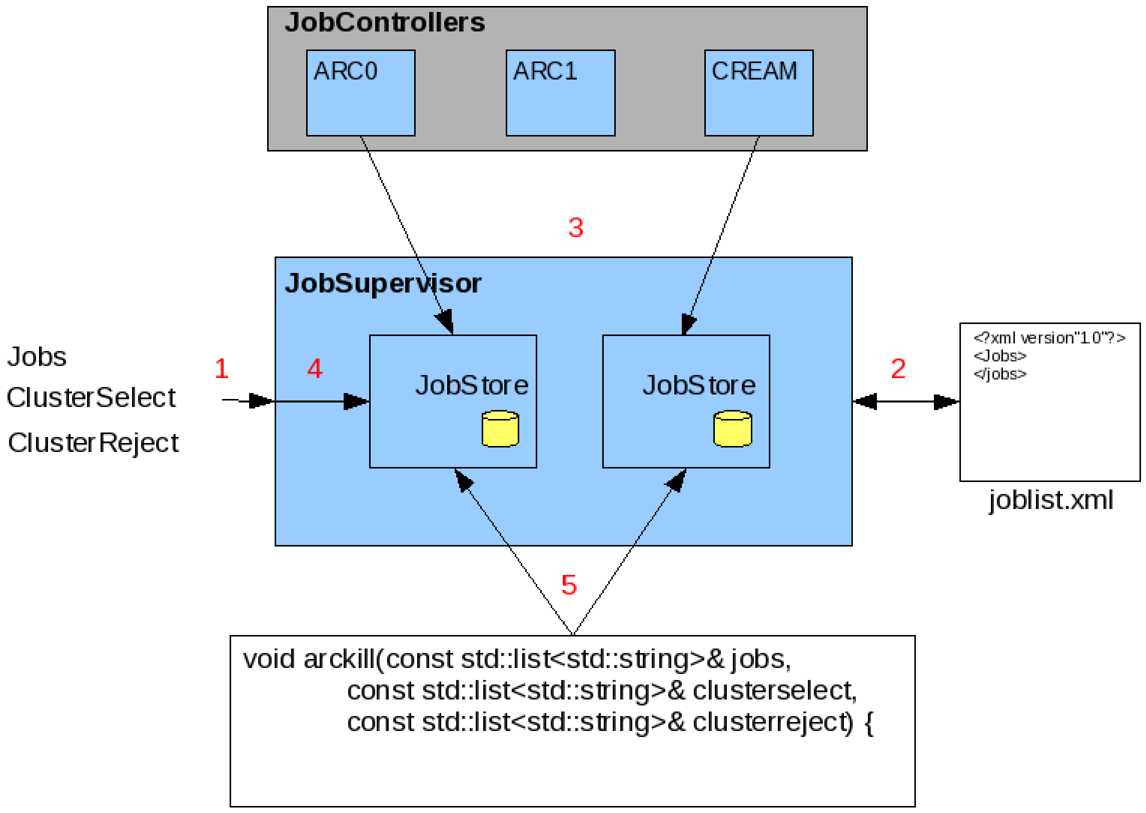
JobControllers
ARC0
ARC1
CREAM
3
JobSupervisor
Jobs
ClusterSelect
ClusterReject
1
4
JobStore
JobStore
2
<?xml version"1.0"?>
<Jobs>
</jobs>
joblist.xml
5
void arckill(const std::list<std::string>& jobs,
const std::list<std::string>& clusterselect,
const std::list<std::string>& clusterreject) {

Figure 2.3: Diagram depicting how job controlling plugins, `JobControllers`, are loaded and initialized.

2. A job identifier does not uniquely define which grid flavour runs a certain job. Thus this information is stored in the joblist file upon submission by the `Submitter` and the joblist file is extensively used by the `JobSupervisor` to identify the `JobController` flavours which are to be loaded. The information in the joblist file is also used to look up the job identifier for jobs specified using job names. Alias resolution for the selected and rejected clusters are performed using the information in the `UserConfig` object.

3. Suitable `JobControllers` are loaded

4. The list of job identifiers and selected and rejected clusters are passed to each `JobController` which uses the information to fill its internal `JobStore`.

5. Residing within the `JobSupervisor` the `JobControllers` are now accessible for external calls (i.e. job handling).

Chapter 3

Implementation

In this section an overview of all important classes in the new libarcclient is given. The overview is subdivided in two parts where first all the generic classes are presented, Section 3.1, before the grid flavour specializations are presented in Section 3.2. Classes are described both in words and by code.

3.1 Generic Classes

3.1.1 ACC

The Arc Client Component (ACC) class is a base class needed for the Loader in order to create loadable classes. It stores information about which flavour it supports and security related information needed by all ACC specializations.

```
class ACC {
protected:
  ACC(Config *cfg, const std::string& flavour);
public:
  virtual ~ACC();
  const std::string& Flavour();
protected:
  std::string flavour;
  std::string proxyPath;
  std::string certificatePath;
  std::string keyPath;
  std::string caCertificatesDir;
};
```

3.1.2 TargetGenerator

The TargetGenerator class is the main class for resource discovery and information retrieval. It loads TargetRetriever plugins in accordance with the input URLs using the ARC Loader [?] (e.g. if an URL pointing to an ARC1 resource is given the TargetRetrieverARC1 is loaded).

To perform a resource discovery, first construct a TargetGenerator object using the TargetGenerator constructor,

```
TargetGenerator(const UserConfig& usercfg,
                const std::list<std::string>& clusters,
                const std::list<std::string>& indexurls);
```

where `clusters` and `indexurls` are lists of strings where each string is either the name of a grid flavour followed by a colon and the URL of an information contact endpoint for a computing service or index server, respectively, of that grid flavour or an alias defined in the `UserConfig` object passed as the first argument, e.g.:

```
ARC0:ldap://grid.tsl.uu.se:2135/Mds-Vo-name=Sweden,o=grid
ARC0:ldap://grid.tsl.uu.se:2135/nordugrid-cluster-name=grid.tsl.uu.se,Mds-Vo-name=local,o=grid
```

where the former is the URL to an index server while the latter is an URL to a computing service. If a string in `clusters` or `indexurls` is prefixed by a minus sign, the corresponding computing service or index server is rejected and excluded during resource discovery.

To prepare a list of `ExecutionTargets` use the `TargetGenerator` object and invoke its method

```
GetTargets(int targetType, int detailLevel);
```

The `TargetGenerator` will pass this request to the loaded `TargetRetrievers` running them as individual threads for improved performance. The `TargetGenerator` keeps records of index servers and computing services found by the `TargetRetrievers` in order to avoid multiple identical queries. Accepted `ExecutionTargets` are stored in the FoundTargets array kept by the `TargetGenerator`. See also Section 2.1 for a schematic drawing of the resource discovery process.

Information about the found `ExecutionTargets` can be printed by

```
PrintTargetInfo(bool longlist) const;
```

3.1.3 TargetRetriever

The `TargetRetriever` is base class for `TargetRetriever` grid middleware specializations and inherits from the `ACC` base class in order to be loadable by the ARC `Loader`. It is designed to work in conjunction with the `TargetGenerator` and contains the pure virtual method

```
virtual void GetTargets(TargetGenerator& mom, int targetType,
                        int detailLevel) = 0;
```

which is to be implemented by the specialized class. While it is not mandatory it is recommended that the specialized class divides this method into two components: QueryIndex and InterrogateTarget. The former handles index server queries, and the latter the computing service queries and `ExecutionTarget` preparation.

If an index server query yields a URL to a different index server than the one queried, then the `TargetRetriever` should call itself recursively creating a new `TargetRetriever` for the discovered index server.

3.1.4 ExecutionTarget

The `ExecutionTarget` is the class representation of a computing resource (queue) capable of executing a grid job. It serves as input to the `Broker` which is foreseen to be able to select between different `ExecutionTargets` from different grid flavours without a priori knowing their difference. The `ExecutionTarget` class mimics the Glue2 information model (with a flattened structure), and thus a mapping between attributes from other information systems into the Glue2 format is needed. Appendix A shows the current mapping for the production ARC, ARC1 and gLite middlewares.

All attributes of the `ExecutionTarget` can be printed by the method

```
void Print(bool longlist) const;
```

Following a broker decision jobs are to be submitted. Since all information about the selected computing service resides within the selected `ExecutionTarget`, the `ExecutionTarget` is capable of returning a `Submitter` capable of submitting a job to the service it represents.

```
Submitter *GetSubmitter(const UserConfig& ucfg) const;
```

The `UserConfig` object is used to configure the security related information needed by the `Submitter`.

3.1.5 Broker

The Broker inherits from the `ACC` base class in order to be loadable by the ARC `Loader`. It is the base class of the specialized Brokers. If you are using the arcsub command then you can choose a broker type with "-b" parameter. There is a solution for add parameters to these descendants Brokers. Here is an example:

```
arcsub -b RandomBroker:a,b,c -c ARC1:https://knowarc1.grid.niif.hu:60000/arex job.jsdl
```

The Broker's constructor set the PreFilteringDone and the TargetSortingDone values to false because we need to call at first the PreFilterTargets and second the GetBestTarget which trigger the specialised Broker sorting method.

The base Broker operations are the followings:

1. The first step is the PreFilterTargets, this implements the matchmaking: the JobDescription and the TargetGenerator's ExecutionTargets will be compared inside this method. JobDescription content the job's JobInnerRepresentation, and the ExecutionTarget content the candidate cluster's queue information. There are main elements in the JobInnerRepresentation which are used for matchmaking, these are documented in this Appendix: ... (every matchmaking elements and their comparison logic are there) The macthmaking working on this way: it takes every JobInnerrepresentation elements which are relevant and in an ordered way check them. If a matchmaking element is not presented in the JobInnerRepresentation then it will be ignored, but if it is there than the ExecutionTarget should content its pair. For example SessionLifeTime is in the JobInnerRepresentaion and the WorkingAreaLifeTime is in the ExecutionTarget (these are matchmaking pair). If the WorkingAreaLifeTime is not defined then the Broker cannot check the times therefore it will ignore this cluster, but if the WorkingAreaLifeTime exist and the comparison between this elements are true then this check will be passed other way the cluster will be ignored. We are using this rule for every elements checking. There is only one exception and this is the ProcessingStartTime because if the DownTime is not defined in the ExecutionTarget than we can trust by it that the job can anytime start and it will be not killed with shutdown. This rule checking runs for every ExecutionTargets and if every matchmaking checks are ok for a target then this ExecutionTarget will be added to the PossibleTargets vector.
2. If the matchmaking finished then the PreFilteringDone variable will be true and the TargetSortingDone will be false.
3. The next step is the GetBestTarget method calling, this will give an ExecutionTarget which is good for our job. If the Possibletarget vector empty then it will give nothing. The first time the GetBestTarget will trigger the specialid Broker type sorting with this method calling: SortTargets. This SortTargets method is virtual method, therefore every specialised Broker should implement it.
4. If the SortTargets finished then the TargetSortingDone will be true and the GetBestTarget will return the first element of the PossibleTargetVector and the current iterator will be incremented with one.

5. If the submision failed to this cluster then we can call again the GetBestTarget and we will get an other ExecutionTarget till it will not consume. If the targets run out from the PossibleTargetVector then the GetBestTarget's bool parameter will be true, its name is EndOfList"

```
class Broker : public ACC {
public:
    ExecutionTarget& GetBestTarget(bool &EndOfList);
    void PreFilterTargets(Arc::TargetGenerator& targen, Arc::JobDescription jd);
protected:
    Broker(Config *cfg);
    virtual ~Broker();
    virtual void SortTargets() = 0;
    std::vector<Arc::ExecutionTarget> PossibleTargets;
    bool TargetSortingDone;
    Arc::JobInnerRepresentation jir;
    static Logger logger;
private:
    std::vector<Arc::ExecutionTarget>::iterator current;
    bool PreFilteringDone;
};
```

The Broker class content two useful data elements: logger and JobInnerRepresentation. These can be helpful when somebody create a specialised Broker and he would like to use the logger and need some information about the job.

3.1.6 JobDescription

When addressing interoperability it is of paramount importance to transparently address grid job descriptions written in different job description languages by translating them automatically. In the libarcclient library this functionality is implemented in the `JobDescription` class. The inner representation (`JobInnerRepresentation` class) is like to the GIN-JSDL and all elements are correspond to variables. This variables are simple variables or complex data structures, and private on the `JobDescription` class.

This is a generic class that takes a job description (string) as input

```
bool setSource(const std::string source);
```

in any supported formats (currently XRSL [?], JDL [?], JSDL [?]), converts and stores it in a GIN-JSDL-like internal job description format. The identification of the source's description format is internally handled by a `JobDescriptionOrderer` class which identifies the format by pattern matching. If the source string is empty or the `JobDescriptionOrderer` class can not be recognize to any formats, then it is not store nothing and return with false, else convert and store it.

Different operations, like getting the job description in other formats or getting job-related information, can be performed using the description-independent functions of this class, e.g.:

```
bool getProduct(std::string& product, std::string format = "POSIXJSDL")
```

The `JobDescription` class has three back-end classes, sometimes referred to as back-end or translator modules, corresponding to the three supported job description languages. The `JobDescription` class chooses the appropriate back-end module according to the pattern match performed by the `JobDescriptionOrderer`, and uses the back-end module for parsing and generating the job descriptions. If the inner representation is empty or the source job descrition is not valid, then the output is empty and return with false. When the source format is equal with the output's format and it is not XRSL, than set the product string with the source string. The XRSL output is always generation from the inner representation.

Parsing is currently possible from any of the languages. JSDL, XRSL and JDL output generation is implemented yet and even for these there are some limitations.

The conversion mainly means a syntactical transformation and the `JobDescription` class can of course do nothing in those cases where not enough data to assemble a given attribute is available or when information can be lost because the received attribute has no equivalent in the output language.

For the JSDL output generation the `JobDescription` class is using the core JSDL's capabilities amended with the JSDL-POSIX and JSDL-ARC extensions. The loss of information is minimal in case of generating such an output.

For the JDL generation the latest version of JDL specification was used (v0.9) [?]. Deprecated and backward compatibility attributes are not implemented.

3.1.7 Submitter

`Submitter` is base class for grid specific specializations (plugin). It submits job(s) to the service it represents and uploads (by the job needed) local input files.

```
virtual bool Submit(JobDescription& jobdesc, XMLNode& info) = 0;
```

The `Submit` method fills the `XMLNode info` with all needed information about the job for later job management. The `Submitter` is returned by the `ExecutionTarget` selected for job execution and thus the `ExecutionTarget` populates (through its `XMLNode` config element) the `Submitter` with information about submission endpoint (`URL`) and job description languages understood by the target.

3.1.8 JobSupervisor

The `JobSupervisor` is responsible for loading the appropriate `JobControllers` for managing jobs running on a certain grid flavour. Job manipulation can be performed either on individual jobs or on groups of jobs (e.g. all jobs running on certain cluster or all jobs with job state "FINISHED"), and in order to translate the information given by the user into a set of loadable `JobControllers` the `JobSupervisor` makes extensive use of the local joblist file housing information about all active jobs. Thus the `JobSupervisor` constructor becomes

```
JobSupervisor(const UserConfig& usercfg,
              const std::list<std::string>& jobs,
              const std::list<std::string>& clusters,
              const std::string& joblist);
```

where `jobs` is a list of job identifiers and job names, `clusters` is a list of selected and rejected computing services in the same format as described above for the `TargetGenerator`, `joblist` is a string containing the name of the joblist file.

Although being loaded by the `JobSupervisor` the `JobController` objects truly resides within the ARC `Loader` which is a member of the `JobSupervisor` class. In order to get handles on the `JobControllers` the inline method

```
const std::list<Arc::JobController*>& GetJobControllers();
```

returns a list of pointers to the loaded `JobControllers`.

3.1.9 JobController

The JobController is a base class for grid specific specializations, but also the implementer of all public functionality offered by the JobControllers. In other words all virtual functions of the JobController are private. The initialization of a (specialized) JobController object takes two steps. First the JobController specialization for the required grid flavour must be loaded by the ARC Loader, which sees to that the JobController receives information about its flavour (grid) and the local joblist file containing information about all active jobs (flavour independent). Next step is the filling of the JobController's job pool JobStore which is the pool of jobs that the JobController can manage.

```
void FillJobStore(const std::list<URL>& jobids,
                  const std::list<URL>& clusterselect,
                  const std::list<URL>& cluterreject);
```

Here jobids, clusterselect and clusterreject have been resolved for job names and aliases by the JobSupervisor, and no further resolution is needed. The following rules are observed when filling the JobStore:

1. If the jobids list has entries, fill JobStore with the by user requested jobs.
2. If the clusterselect list has entries, fill JobStore with the jobs running on the selected clusters.
3. If clusters are rejected and JobStore has entries, remove from JobStore all jobs running on rejected clusters.
4. If jobs and clusterselect are both empty lists, fill JobStore with all jobs except those running on possible rejected clusters.

The steps above completes the initialization of the JobController which is now ready for handling jobs. The public functions of the JobController offer to get (download), clean, cancel, etc one or more jobs and uses the private specializations for issuing the command. Here exemplified by the Stat command:

```
bool JobController::Stat(const std::list<std::string>& status,
                         const bool longlist,
                         const int timeout) {

  GetJobInformation();

  for (std::list<Job>::iterator it = jobstore.begin();
       it != jobstore.end(); it++) {
    if (it->State.empty()) {
      logger.msg(WARNING, "Job state information not found: %s",
                 it->JobID.str());
      Time now;
      if (now - it->LocalSubmissionTime < 90)
        logger.msg(WARNING, "This job was very recently "
                   "submitted and might not yet"
                   "have reached the information-system");
      continue;
    }
    it->Print(longlist);
  }
  return true;
}
```

The Stat command prints the job information to screen and in order to do so the JobController has to query local information server for the latest status. Due to different protocols used for different grid flavours (e.g. ldap for production ARC), the GetJobInformation has to be grid flavour specific and is only declared as a private virtual method within the JobController base class. For details about the flavour specific implementations see Section 3.2.

3.1.10 Job

The Job is a generic job class for storing all job related information. Attributes are derived from the Glue2 information model and thus a mapping is needed for non Glue2 compliant grid middlewares. Appendix B shows the present mapping schema.

All attributes of the Job can be printed by the method

```
void Print(bool longlist) const;
```

3.1.11 UserConfig

The UserConfig class handles the client setup i.e. proxy, certificate and key location, user and system configuration and local joblist location. Upon initialization (constructor) the UserConfig locates the user files

```
$HOME/.arc/client.xml
$HOME/.arc/jobs.xml
```

and if either of them is non-existing a default (empty) one is created.

The UserConfig has three main public methods

```
const std::string& JobListFile() const;
const XMLNode& ConfTree() const;
bool ApplySecurity(XMLNode& ccfg) const;
```

where JobListFile() returns the string pointing to the jobs.xml file while ConfTree() returns a configuration XMLNode object which is the merge between the user and system configurations. In order to do this the method has to locate the system configuration and resolve possible conflicts with the user configuration. This proceeds through the following chain of actions:

1. Try reading system configuration from <ARC Install Location>/etc/arcclient.xml
2. If the previous step failed try reading system configuration from /etc/arcclient.xml
3. Merge system and user configuration by adding all system configuration not already listed in the user configuration to the latter.

The ApplySecurity(XMLNode& ccfg) adds security tags to the ccfg XMLNode as follows:

- If the $X509_USER_PROXY environment variable is set, use its value to define a ProxyPath tag in cfg. If the file does not exist or can't be read exit with an error.
- Otherwise, if the $X509_USER_CERT environment variable is set, use its value and the value of the $X509_USER_KEY environment variable to define CertificatePath and KeyPath tags in cfg. If $X509_USER_KEY is not set or either file does not exist or can't be read exit with an error.

- Otherwise, if the merged user configuration tree (see `ConfTree` contains a `ProxyPath` tag, copy it to `cfg`. If the file does not exist or can't be read exit with an error.
- Otherwise, if the merged user configuration tree contains a `CertificatePath` tag, copy it to `cfg` along with the accompanying `KeyPath` tag. If the `KeyPath` tag is undefined or either file does not exist or can't be read exit with an error.
- Otherwise, if the file `/tmp/x509up_u + userid` exists add a `ProxyPath` tag in `cfg` pointing to it.
- Otherwise, add `CertificatePath` and `KeyPath` tags to `cfg` pointing to `$HOME/.globus/usercert.pem` and `$HOME/.globus/userkey.pem`, respectively. If either file does not exist exit with error.
- If the `$X509_CERT_DIR` environment variable is set, use its value to define a `CACertificatesDir` tag in `cfg`. If the directory does not exist, exit with an error.
- Otherwise, if the user ID is not zero and the directory `$HOME/.globus/certificates` exists add a `CACertificatesDir` tag in `cfg` pointing to it.
- Otherwise, add a `CACertificatesDir` tag in `cfg` pointing to `/etc/grid-security/certificates`. If the directory does not exist, exit with an error.

3.1.12 ACCConfig

Locating Arc Client Components (plugins) is handled by the `ACCConfig` class. It inherits from `BaseConfig` and implements only one method

```
virtual XMLNode MakeConfig(XMLNode cfg) const;
```

The `MakeConfig` method searches plugin paths for all libraries named `libacc*`. Matching libraries are added as plugins to the configuration object `cfg`.

3.1.13 ClientInterface

The `ClientInterface` class is a utility base class used for configuring a client side Message Chain Component (MCC) chain and loading it into memory. It has several specializations of increasing complexity of the MCC chains.

ClientTCP

The `ClientTCP` class is a specialization of the `ClientInterface` which sets up a client MCC chain for TCP communication, and optionally with a TLS layer on top.

ClientHTTP

The `ClientHTTP` class inherits from the `ClientTCP` class and adds an HTTP MCC to the chain.

ClientSOAP

The `ClientSOAP` class inherits from the `ClientHTTP` class and adds a SOAP MCC to the chain. Specializations of the `TargetRetriever`, `Submitter` and `JobController` classes that communicate with SOAP based services make use of this class.

3.2 Specialized Classes (Grid Flavour and Broker plugins)

3.2.1 ARC0 Plugins

The ARC0 plugins enables support for the interfaces used by computing elements running ARC version 0.x.

The ARC 0.x local information system uses the Globus Toolkit® [?] GRIS with a purpose made ARC schema. The information index server used is the Globus Toolkit® GIIS. Both these servers are using the LDAP [?] protocol. The specialization of the `TargetRetriever` class for ARC0 is implemented using the ARC LDAP Data Management Component (DMC) (see [?] for technical details).

Jobs running on an ARC 0.x computing element are handled by the ARC grid-manager [?]. Job submission and job control are done using the gridftp [?] protocol. The specializations of the `Submitter` and `JobController` classes use the globus ftp control library.

Stage-in and stage-out of input and output files are also done using the gridftp [?] protocol. This means that proper functionality of the ARC0 plugins requires the gridftp DMC.

3.2.2 ARC1 Plugins

The computing element in ARC version 1.x is the A-Rex [?] service running in a HED [?] container.

A-Rex implements the BES [?] standard interface. Since this is a SOAP [?] based interface the specializations of the `TargetRetriever`, `Submitter` and `JobController` classes make use of a chain of ARC Message Chain Components (MCC [?]) ending with the SOAP client MCC.

The A-Rex service uses the https protocol put and get methods for stage-in and stage-out of input and output files. Therefore, the ARC1 plugins requires the http DMC.

3.2.3 gLite Plugins

The gLite computing element offers several interfaces, one of them being the Web Service based computing element interface known as the CREAM CE [?]. The current revision of this interface (CREAM version 2) has been chosen for implementation within libarcclient for the following reasons:

- CREAM2 has a Web Service interface that fits the Web Service based ARC.
- CREAM2 enables direct access to the gLite computing element without having to go via the gLite workload management system.
- CREAM2 contains numerous improvements when compared to the earlier CREAM versions.
- CREAM2 supports direct job status queries.
- CREAM2 offers a convenient way of handling input and output files through accessing the input and output sandbox via GridFTP.

gLite resources are registered in top level and site BDIIs. The CREAM specialization of the `TargetRetriever` therefore makes use of the LDAP DMC similarly to the ARC0 plugins.

CREAM has its own SOAP based interface. The CREAM specializations of the `Submitter` and `JobController` classes therefore use an MCC chain ending with the SOAP client MCC the same way the ARC1 plugin does.

Stage-in and stage-out of input and output files are done using the gridftp protocol. The gridftp DMC is therefore required.

3.2.4 Broker plugins

RandomBroker

The sorting method is random.

TODO: finish

FastestCPUBroker

The sorting method based on the fastest cpu, we are using for this the specint2006 benchmark solution.
TODO: finish

FastestQueueBroker

The sorting method based on the shortest queue, where the waiting jobs number is low.
TODO: finish

DataBroker

The sorting method based on the A-REX file cache. I will describe how does it working.
TODO: finish

PythonBroker

TODO: finish

Chapter 4

Building Command Line Interfaces

Given all components listed above it is possible to write versatile command line interfaces (cli) for grid job submission and management. libarcclient offers the following native commands:

1. `arcstat` - for computing resource or grid job status queries.
2. `arcsub` - for grid job submission
3. `arcget` - for downloading output of finished, cancelled or failed grid jobs.
4. `arckill` - for terminating grid jobs.
5. `arcclean` - for removing a grid job's session directory including all contents
6. `arccat` - for performing the `cat` command to a running grid job's std out or std error file.

Each of the commands above are encoded within one C++ file with the following structure, here exemplified with the arcget command:

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

#include <iostream>
#include <list>
#include <string>

#include <arc/ArcLocation.h>
#include <arc/IString.h>
#include <arc/Logger.h>
#include <arc/OptionParser.h>
#include <arc/client/JobController.h>
#include <arc/client/JobSupervisor.h>
#include <arc/client/UserConfig.h>

int main(int argc, char **argv) {

  setlocale(LC_ALL, "");

  Arc::Logger logger(Arc::Logger::getRootLogger(), "arcget");
  Arc::LogStream logcerr(std::cerr);
  Arc::Logger::getRootLogger().addDestination(logcerr);
  Arc::Logger::getRootLogger().setThreshold(Arc::WARNING);
```

```
Arc::ArcLocation::Init(argv[0]);

Arc::OptionParser options(istring("[job ...]"),
                          istring("The arcget command is used for "
                                  "retrieving the results from a job."),
                          istring("Argument to -c has the format "
                                  "Flavour:URL e.g.\n"
                                  "ARC0:ldap://grid.tsl.uu.se:2135/"
                                  "nordugrid-cluster-name=grid.tsl.uu.se,"
                                  "Mds-Vo-name=local,o=grid"));

bool all = false;
options.AddOption('a', "all",
                  istring("all jobs"),
                  all);

// Removed most of the option definition from this write-up to
// save space. See the real source file for the complete list.
// ...

bool version = false;
options.AddOption('v', "version", istring("print version information"),
                  version);

std::list<std::string> jobs = options.Parse(argc, argv);

if (!debug.empty())
  Arc::Logger::getRootLogger().setThreshold(Arc::string_to_level(debug));

Arc::UserConfig usercfg(conffile);
if (!usercfg) {
  logger.msg(Arc::ERROR, "Failed configuration initialization");
  return 1;
}

if (debug.empty() && usercfg.ConfTree()["Debug"]) {
  debug = (std::string)usercfg.ConfTree()["Debug"];
  Arc::Logger::getRootLogger().setThreshold(Arc::string_to_level(debug));
}

if (version) {
  std::cout << Arc::IString("%s version %s", "arcget", VERSION)
            << std::endl;
  return 0;
}

if (jobs.empty() && joblist.empty() && !all) {
  logger.msg(Arc::ERROR, "No jobs given");
  return 1;
}

if (joblist.empty())
  joblist = usercfg.JobListFile();

Arc::JobSupervisor jobmaster(usercfg, jobs, clusters, joblist);

std::list<Arc::JobController*> jobcont = jobmaster.GetJobControllers();
```

```
  if (jobcont.empty()) {
    logger.msg(Arc::ERROR, "No job controllers loaded");
    return 1;
  }

  int retval = 0;
  for (std::list<Arc::JobController*>::iterator it = jobcont.begin();
       it != jobcont.end(); it++)
    if (!(*it)->Get(status, downloaddir, keep, timeout))
      retval = 1;

  return retval;
}
```

Appendix A

ExecutionTarget

A.1 Domain and Location attributes

std::string DomainName

ARC0: nordugrid-cluster-name
CREAM: GlueSiteName

std::string Owner

ARC0: nordugrid-cluster-owner

std::string Address

std::string Place

CREAM: GlueSiteLocation

std::string PostCode

ARC0: nordugrid-cluster-location

float Latitude

CREAM: GlueSiteLatitude

float Longitude

CREAM: GlueSiteLongitude

A.2 ComputingService and ComputingEndpoint attributes

std::string CEID

std::string CEName

std::string Capability

std::string Type

std::string QualityLevel

URL url

ARC0: nordugrid-cluster-contactstring
CREAM: GlueCEInfoContactString

std::string Technology

std::string Interface

ARC0: “GridFTP”

std::string InterfaceExtension

std::string SupportedProfile

std::string Implementor

ARC0: “NorduGrid”

std::string ImplementationName

ARC0: “ARC0”
CREAM: GlueCEImplementationName

std::string ImplementationVersion

ARC0: nordugrid-cluster-middleware
CREAM: GlueCEImplementaionVersion

std::string HealthState

ARC0: nordugrid-queue-status

std::string ServingState

CREAM: GlueCEStateStatus (GlueVOView || GlueCE)

std::string IssuerCA

ARC0: nordugrid-cluster-issuerca (nordugrid-cluster-issuerca-hash)

std::list¡std::string¿ TrustedCA

ARC0: nordugrid-cluster-trustedca

Time DowntimeStarts

Time DowntimeEnds

std::string Staging

ARC0: nordugrid-cluster-nodeaccess

std::string Jobdescription

A.3 ComputingService and ComputingShare load attributes

int TotalJobs

ARC0: nordugrid-cluster-totaljobs
CREAM: GlueCEStateTotalJobs (GlueVOView || GlueCE)

int RunningJobs

ARC0: nordugrid-queue-running
CREAM: GlueCEStateRunningJobs (GlueVOView || GlueCE)

int WaitingJobs

ARC0: nordugrid-queue-queued || nordugrid-cluster-queuedjobs
CREAM: GlueCEStateWaitingJobs (GlueVOView || GlueCE)

int StagingJobs

int SuspendedJobs

int PreLRMSWaitingJobs

ARC0: nordugrid-queue-prelrmsqueued

int LocalRunningJobs

ARC0: nordugrid-queue-running − nordugrid-queue-gridrunning

int LocalWaitingJobs

ARC0: nordugrid-queue-queued − nordugrid-queue-gridqueue

int FreeSlots

CREAM: GlueCEStateFreeJobSlots || GlueCEStateFreeCPUs (GlueVOView || GlueCE)

std::string FreeSlotsWithDuration

int UsedSlots

ARC0: nordugrid-queue-usedcpus

int RequestedSlots

A.4 ComputingShare attributes

std::string MappingQueue

ARC0: nordugrid-queue-name

Period MaxWallTime

ARC0: nordugrid-queue-maxwalltime
CREAM: GlueCEPolicyMaxWallClockTime (GlueVOView || GlueCE)

Period MaxTotalWallTime

Period MinWallTime

ARC0: nordugrid-queue-minwalltime

Period DefaultWallTime

ARC0: nordugrid-queue-defaultwalltime

Period MaxCPUTime

ARC0: nordugrid-queue-maxcputime
CREAM: GlueCEPolicyMaxCPUTime (GlueVOView || GlueCE)

Period MaxTotalCPUTime

Period MinCPUTime

ARC0: nordugrid-queue-defaultcputime

Period DefaultCPUTime

ARC0: nordugrid-queue-defaultcputime

int MaxTotalJobs

CREAM: GlueCEPolicyMaxTotalJobs (GlueVOView || GlueCE)

int MaxRunningJobs

ARC0: nordugrid-queue-maxrunning
CREAM: GlueCEPolicyMaxRunningJobs (GlueVOView || GlueCE)

int MaxWaitingJobs

ARC0: nordugrid-queue-maxqueable
CREAM: GlueCEPolicyMaxWaitingJobs (GlueVOView || GlueCE)

int NodeMemory

ARC0: nordugrid-queue-nodememory || nordugrid-cluster-nodememory
CREAM: GlueHostMainMemoryRAMSize

int MaxPreLRMSWaitingJobs

int MaxUserRunningJobs

ARC0: nordugrid-queue-maxuserrun
CREAM: GlueCEPolicyAssignedJobSlots (GlueVOView || GlueCE)

int MaxSlotsPerJob

CREAM: GlueCEPolicyMaxSlotsPerJob (GlueVOView || GlueCE)

int MaxStageInStreams

int MaxStageOutStreams

std::string SchedulingPolicy

ARC0: nordugrid-queue-schedulingpolicy

int MaxMemory

ARC0: nordugrid-queue-nodememory || nordugrid-cluster-nodememory
CREAM: GlueHostMainMemoryVirtualSize

int MaxDiskSpace

URL DefaultStorageService

ARC0: nordugrid-cluster-localse
CREAM: GlueCEInfoDefaultSE (GlueVOView || GlueCE)

bool Preemption

CREAM: GlueCEPolicyPreemption (GlueVOView || GlueCE)

Period EstimatedAverageWaitingTime

CREAM: GlueCEStateEstimatedResponseTime (GlueVOView || GlueCE)

Period EstimatedWorstWaitingTime

CREAM: GlueCEStateWorstResponseTime (GlueVOView || GlueCE)

std::string ReservationPolicy

A.5 ComputingManager attributes

std::string ManagerType

ARC0: nordugrid-cluster-lrms-type (nordugrid-cluster-lrms-version)

bool Reservation

bool BulkSubmission

int TotalPhysicalCPUs

ARC0: nordugrid-queue-totalcpus || nordugrid-cluster-totalcpus

int TotalLogicalCPUs

ARC0: nordugrid-queue-totalcpus || nordugrid-cluster-totalcpus

int TotalSlots

ARC0: nordugrid-queue-totalcpus || nordugrid-cluster-totalcpus

bool Homogeneity

ARC0: nordugrid-queue-homogeneity || nordugrid-cluster-homogeneity

std::string NetworkInfo

bool WorkingAreaShared

int WorkingAreaFree

ARC0: nordugrid-cluster-sessiondir-free

Period WorkingAreaLifeTime

ARC0: nordugrid-cluster-sessiondir-lifetime

int CacheFree

ARC0: nordugrid-cluster-cache-free

A.6 ExecutionEnvironment attributes

std::string Platform

ARC0: nordugrid-queue-architecture || nordugrid-cluster-architecture

bool VirtualMachine

std::string CPUVendor

ARC0: nordugrid-queue-nodecpu || nordugrid-cluster-nodecpu

std::string CPUModel

ARC0: nordugrid-queue-nodecpu || nordugrid-cluster-nodecpu

std::string CPUVersion

ARC0: nordugrid-queue-nodecpu || nordugrid-cluster-nodecpu

int CPUClockSpeed

int MainMemorySize

std::string OSFamily

ARC0: nordugrid-queue-opsys || nordugrid-cluster-opsys

std::string OSName

ARC0: nordugrid-queue-opsys || nordugrid-cluster-opsys

std::string OSVersion

bool ConnectivityIn

bool ConnectivityOut

std::list¡Benchmark¿ Benchmarks

ARC0: nordugrid-queue-benchmark || nordugrid-cluster-benchmark

A.7 ApplicationEnvironment

std::list¡ApplicationEnvironment¿ ApplicationEnvironments

ARC0: nordugrid-cluster-runtimeenvironment
CREAM: GlueHostApplicationSoftwareRunTimeEnvironment

Appendix B

Job

B.1 Information stored in the job list file

std::string Flavour

URL JobID

URL Cluster

URL SubmissionEndpoint

URL InfoEndpoint

URL ISB

URL OSB

URL AuxURL

std::string AuxInfo

B.2 Information retrieved from the informtaion system

std::string Name

ARC0: nordugrid-job-jobname

std::string Type

URL IDFromEndpoint

std::string LocalIdFromManager

std::string JobDescription

std::string State

ARC0: nordugrid-job-status

std::string RestartState

ARC0: nordugrid-job-rerunable

int ExitCode

ARC0: nordugrid-job-exitcode

std::string ComputingManagerExitCode

std::list¡std::string¿ Error

ARC0: nordugrid-job-errors

int WaitingPosition

ARC0: nordugrid-job-queuerank

std::string UserDomain

std::string Owner

ARC0: nordugrid-job-globalowner

std::string LocalOwner

Period RequestedWallTime

ARC0: nordugrid-job-reqwalltime

Period RequestedTotalCPUTime

ARC0: nordugrid-job-reqcputime

int RequestedMainMemory

int RequestedSlots

std::string StdIn

ARC0: nordugrid-job-stdin

std::string StdOut

ARC0: nordugrid-job-stdout

std::string StdErr

ARC0: nordugrid-job-stderr

std::string LogDir

ARC0: nordugrid-job-gmlog

std::list¡std::string¿ ExecutionNode

Arc0: nordugrid-job-executionnodes

std::string ExecutionCE

ARC0: nordugrid-job-execcluster

std::string Queue

ARC0: nordugrid-job-execqueue

Period UsedWallTime

ARC0: nordugrid-job-usedwalltime

Period UsedTotalCPUTime

ARC0: nordugrid-job-usedcputime

int UsedMainMemory

ARC0: nordugrid-job-usedmem

std::list¡std::string¿ UsedApplicationEnvironment

ARC0: nordugrid-job-runtimeenvironment

int UsedSlots

ARC0: nordugrid-job-cpucount

Time LocalSubmissionTime

Time SubmissionTime

ARC0: nordugrid-job-submissiontime

Time ComputingManagerSubmissionTime

Time StartTime

Time ComputingManagerEndTime

Time EndTime

ARC0: nordugrid-job-completiontime

Time WorkingAreaEraseTime

ARC0: nordugrid-job-sessiondirerasetime

Time ProxyExpirationTime

ARC0: nordugrid-job-proxyexpirationtime

std::string SubmissionHost

ARC0: nordugrid-job-submissionui

std::string SubmissionClientName

ARC0: nordugrid-job-clientsoftware

Time CreationTime

ARC0: Mds-validfrom

Period Validity

ARC0: Mds-validto − Mds-validfrom

std::string OtherMessages

ARC0: nordugrid-job-comment

B.3 Associations

URL JobManagementEndpoint

URL DataStagingEndpoint

B.4 ExecutionEnvironment (condensed)

bool VirtualMachine

std::string UsedCPUType

std::string UsedOSFamily

std::string UsedPlatform