NORDUGRID-TECH-20
28/4/2009

LIBARCCLIENT

A Client Library for ARC

KnowARC

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 computing resource discovery and information retrieval,
- filtering and brokering of found computing resources,
- 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 show through examples 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 another 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`.

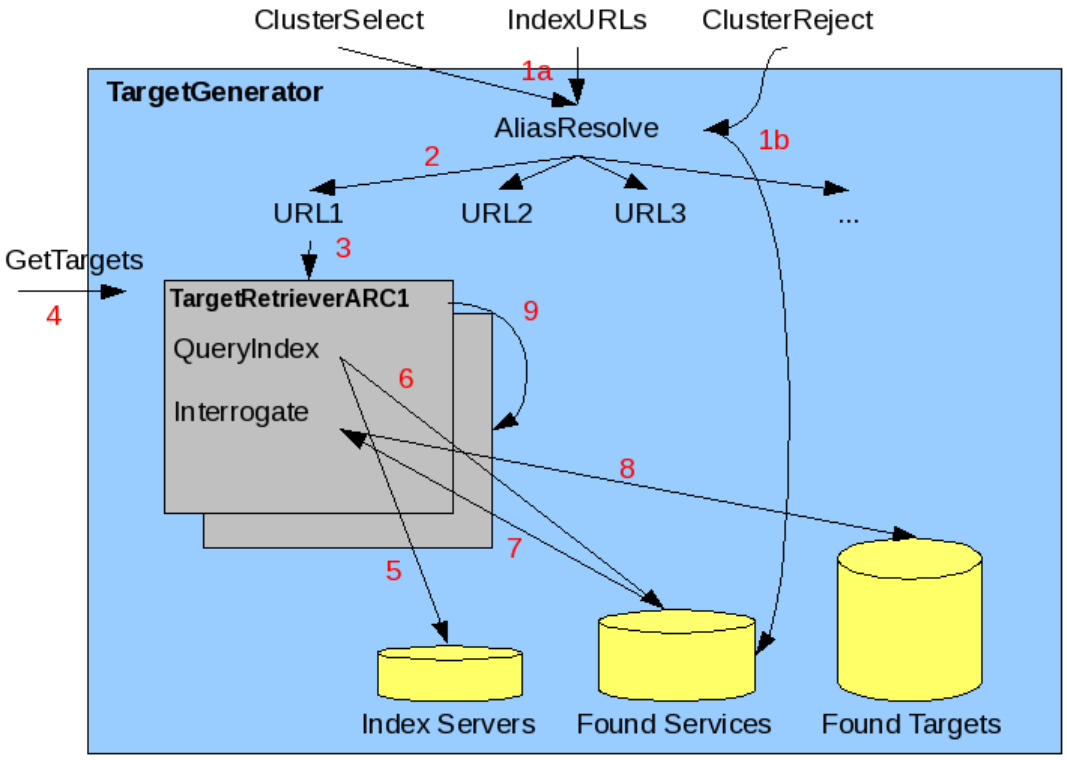

Figure 2.1: Diagram depicting the resource discovery and information retrieval process

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 `TargetRetrievers` 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 authorize 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`.

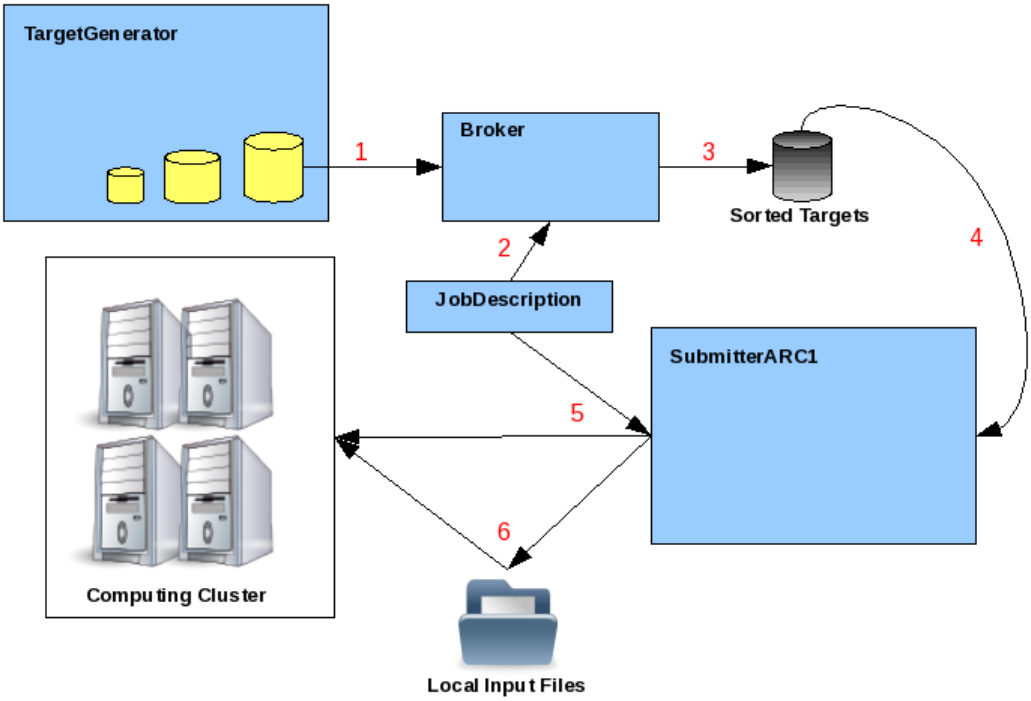

Figure 2.2: Diagram depicting the submission of a job to a computing service.

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` filters and ranks the `ExecutionTargets` according to the ranking method chosen by the user.

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 stores this 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

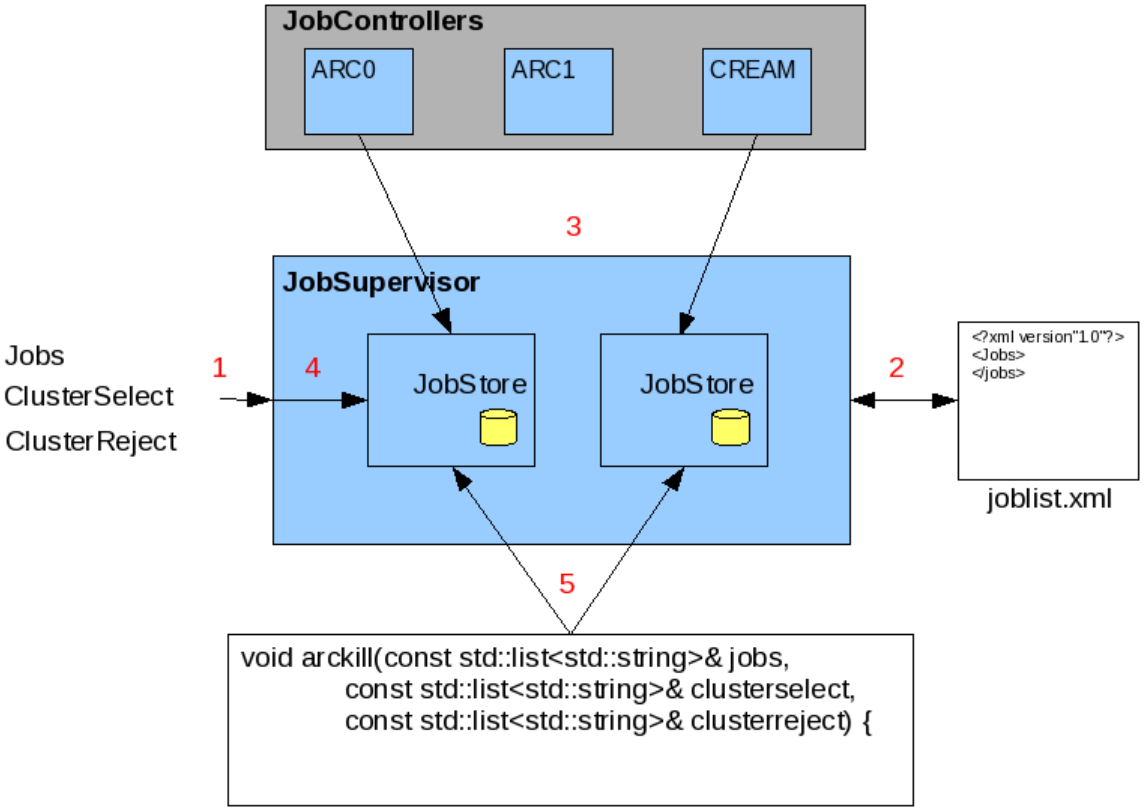

Figure 2.3: Diagram depicting how job controlling plugins, `JobControllers`, are loaded and initialized.

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.

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 resolving 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 `ExecutionTarget`s use the `TargetGenerator` object and invoke its method

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

The `TargetGenerator` will pass this request to the loaded `TargetRetriever`s running them as individual threads for improved performance. The `TargetGenerator` keeps records of index servers and computing services found by the `TargetRetriever`s in order to avoid multiple identical queries. Accepted `ExecutionTarget`s 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 `ExecutionTarget`s can be printed by

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

3.1.3 TargetRetriever

The `TargetRetriever` is the base class for grid middleware specialized `TargetRetriever`s 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 able to filter and rank different `ExecutionTarget`s 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;
```

All information needed to submit a job is stored in the ExecutionTarget, and the ExecutionTarget's method

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

returns a Submitter which is used to submit jobs. 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 Broker and it implements the method for reducing a list of resources found by resource discovery (the list of ExecutionTargets residing within the TargetGenerator) to a list of resources capable of running a certain job:

```
void PreFilterTargets(const TargetGenerator& targen, const JobDescription& jd);
```

The PreFilterTargets method compares every hardware and software requirement given in the job description against the computing resource (cluster) specifications stored in the ExecutionTarget. If theExecutionTarget doesn't fulfil the requirements imposed by the job description, or it is impossible to carry out the matchmaking due to incomplete information published by the computing resource, the ExecutionTarget will be removed from the list of possible targets. A detailed overview of the matchmaking between JobDescription and ExecutionTarget attributes is given in Appendix B.

Once prefiltered, the remaining ExecutionTargets should be ranked in order to return the "best" ExecutionTarget for job submission. Different ranking methods are implemented by the specialized brokers, but for usability and consistency reasons these methods are encapsulated by the Broker base class method

```
ExecutionTarget& GetBestTarget(bool &EndOfList);
```

which invokes the SortTargets method implemented by the specialized broker (see Section 3.2.4) and returns the best target. The GetBestTarget method is "incremented" at each call, thus upon a second call the second best ExecutionTarget will be returned.

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 which parses job descriptions given in the XRSL [?], JDL [?] or JSDL [?] formats into an internal representation based upon the upcoming PGI-JSDL proposal.

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

The setSource method takes a source job description string as input and identifies it's format (language) through pattern matching (the `JobDescriptionOrderer` class). Once the format is known, the source elements are mapped from the original format (XRSL, JDL or JSDL) into an internal representation (the `JobInnerRepresentation` class). Mapping details are given in Section C.

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 matching performed by the `JobDescriptionOrderer`, and uses the back-end module for parsing and translation (if needed) of the job descriptions.

Once the job description language required by the `ExecutionTarget` chosen for job submission is known, the parsed and (if needed) translated job description can be retrieved through the GetProduct function as shown below:

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

If the inner representation is empty or the source job description is not valid, then the GetProduct output is empty too and the method returns with false. In order to obtain the automatic job description modifications similar to those of the production ARC client, the GetProduct output is always generated from the internal representation using the `JobDescriptionOrderer` class to carry out the translation. The original source string can be regained by calling the `getSourceString` method:

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

The translation of a job description basically means a syntactical transformation of the job description from one language to another. Thus the `JobDescription` class can do nothing in those cases where there is not enough data available to assemble a given attribute or when information can be lost because the received attribute has no equivalent in the output language. In these cases the GetProduct method returns with a false value and an empty product string. For limitations in the translation see Section C.

For the JSDL output generation the `JobDescription` class uses 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.

In case of job resubmission the internal representation will contain a list of the jobid's of the previous jobs. This element will only be parsed to the actually job description in case JSDL-POSIX output is requested. A new URL can be added to the list of old jobid's by using the method:

```
bool addOldJobID(const URL& oldjobid);
```

3.1.7 Submitter

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

```
virtual bool Submit(const JobDescription& jobdesc, XMLNode& info) const = 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.

```
virtual bool Migrate(const URL& url,
                     const JobDescription& jobdesc,
                     bool forcemigration,
                     XMLNode& info) const = 0;
```

The Migrate method behaves in much the same way as the Submit method, but instead of sending a new job request to the cluster it sends a migration request of a given job, currently this request is only supported for ARC1 clusters. The URL and jobdesc specifies the job-id and JobDescription of the job which should be migrated.

Note that whereas the Migrate method make provisions for automatically getting files from the migration source cluster, this does not apply to the job description which instead should be passed to Migrate explicitly in the form of a JobDescription object.

The migration request will create a new job where the local input files are transferred from old cluster. Once the input files have been transferred the new cluster will send a request to kill the old job. If the kill request is unsuccessful the new job will be terminated unless the forcemigration boolean is true.

The Migrate method will also fill the info XMLNode with information about the job the same way as the Submit method. However for the Migrate method this XMLNode also contain the job-id of the migrated job.

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 which house 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 loaded by the JobSupervisor the JobController objects actually reside 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<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). The 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 jobs requested by the user.
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 `jobids` and `clusterselect` are both empty lists, fill `JobStore` with all jobs except those running on rejected clusters if there are any.

The steps above complete 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 use the virutal methods implemented by the grid middleware specialized `JobController`s 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.

The other public methods of the `JobController` class are: The `Get` which is used to download jobs

```
bool Get(const std::list<std::string>& status, const std::string& downloaddir,
         const bool keep, const int timeout);
```

```
bool Kill(const std::list<std::string>& status, const bool keep, const int timeout);
```

```
bool Clean(const std::list<std::string>& status, const bool force, const int timeout);
```

```
bool Cat(const std::list<std::string>& status, const std::string& whichfile,
         const int timeout);
```

```
bool RemoveJobs(const std::list<URL>& jobids);
```

```
std::list<std::string> GetDownloadFiles(const URL& dir);
```

```
bool CopyFile(const URL& src, const URL& dst);
```

The `GetJobDescriptions` method

```
std::list<Job> GetJobDescriptions(const std::list<std::string>& status,
                                  const bool getlocal, const int timeout);
```

which is used to get the job descriptions of `Job`s in the `JobStore` with the specified status'. In case `getlocal` is true the job description will first be searched for the users local XML file. If a local copy of the job description is found the checksum of the local input files will be computed and compared with the ones listed in the users local XML file. If all checksums match the job description is returned.

In case `getlocal` is false or the retrieval of the local job description fails an attempt will be made to retrieved the job description from the cluster. This has currently only been implemented for ARC(both generations) clusters. Note that the job descriptions retrieved from a cluster will most likely have been changed since the submission – either by the cluster* or by the submission client. Therefore, the only way of retrieving the original job description is through the users local XML file.

The `GetJobDescriptions` returns a list of the `Job`s where job description have been found. The found job descriptions are kept in the `Job.JobDescription` attribute.

The `Migrate` method

```
bool Migrate(TargetGenerator& targetGen,
             Broker* broker,
             const bool forcemigration,
             const int timeout);
```

*For both generations of ARC it is possible to recover the initial job description send to the cluster. However modifications can still have been made to the job description before sending it to the cluster.

will try to migrate the jobs contained in the object (`JobController`). The jobs will be migrated to the targets specified in `TargetGenerator targetGen` by using the broker specified by `Broker broker`. The boolean `forcemigration` specifies whether a migrated job should persist if the new cluster (`ExecutionTarget`) chosen by the `TargetGenerator` does not succeed sending a kill/terminate request for the job. The integer `timeout` argument has not yet been implemented.

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.

All attributes of the `Job` can be printed by the method (longlist = true will give more details (long listing))

```
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 and reads 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 & 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 items to the latter.

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

- If the `$X509_USER_PROXY` environment variable is set, use its value to define a `ProxyPath` tag in `ccfg`. If the file does not exist or can't be read exit with an error.
- Otherwise, if the `$X509_USER_CERT` environment variable is set, use its value and the value of the `$X509_USER_KEY` environment variable to define `CertificatePath` and `KeyPath` tags in `ccfg`. If `$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 `ccfg`. 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 `ccfg` 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 `ccfg` pointing to it.
- Otherwise, add `CertificatePath` and `KeyPath` tags to `ccfg` pointing to `$HOME/.globus/usercert.pem` and `$HOME/.globus/userkey.pem`, respectively. If either file does not exist exit with error.
- If the `$X509_CERT_DIR` environment variable is set, use its value to define a `CACertificatesDir` tag in `ccfg`. If the directory does not exist, exit with an error.
- Otherwise, if the user ID is not zero and the directory `$HOME/.globus/certificates` exists add a `CACertificatesDir` tag in `ccfg` pointing to it.
- Otherwise, add a `CACertificatesDir` tag in `ccfg` 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.

3.1.14 Sandbox

The `Sandbox` class, consisting of the two methods `Add` and `GetCkSum`, have been created to assure a reliable resubmission of jobs.

```
static bool Add(JobDescription& jobdesc, XMLNode& info);
static std::string GetCksum(const std::string& file);
```

The `Add` method adds the job description and the checksum of the local input files to a `XMLNode`. Currently this `XMLNode` is the one stored in the user's local XML file upon job submission. The job description which is stored is the initial job description before any parsing has been performed. To calculate the checksum of the input files the `GetCkSum` method is used. The `GetCkSum` method returns the MD5 checksum of a given file

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 custom made ARC schema. The information index server used is the Globus Toolkit® GIIS. Both these servers use 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 is very similar to 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 `RandomBroker` ranks the `ExecutionTarget`s randomly.

`FastestCPUBroker`

The `FastestCPUBroker` ranks the `ExecutionTarget`s according to their CPU performance. The most reliable comparison is achieved through the usage of a benchmark scenario, and the `FastestCPUBroker` uses the CINT2000 (Integer Component of SPEC CPU2000)[†] benchmark for this purpose.

The SortTargets method has two steps:

1. `ExecutionTarget`s not publishing the specint2000 benchmark information are removed from the list of possible targets as their performance is unknown.
2. The `ExecutionTarget`s are ranked such that the `ExecutionTarget` with highest specint2000 value receives top ranking

`FastestQueueBroker`

The `FastestQueueBroker` ranks the `ExecutionTarget`s according to their queue length measured in percentage of the `ExecutionTarget`'s size (i.e. the queue length divided by the number of total slots/CPUs). If more than one `ExecutionTarget` has zero queue, the `FastestQueueBroker` makes use of a basic load balancing method to rearrange the zero queue `ExecutionTarget`s depending on their number of free slots/CPUs.

The SortTargets method has three steps:

1. `ExecutionTarget`s not publishing WaitingJobs, TotalSlots and FreeSlots are removed from the list of possible targets as it is impossible to determine their queue length and perform load balancing.
2. The `ExecutionTarget`s are ranked according to their queue length relative to the number of slots/CPUs. Zero queue `ExecutionTarget`s is ranked first.
3. If more than one `ExecutionTarget` has zero queue, the targets are randomly reshuffled based upon how many free slots they have, e.g. if two `ExecutionTarget`s have zero queue and they have 90 and 10 free slots respectively, then the first `ExecutionTarget` has a 90% probability of becoming the top ranked cluster.

`DataBroker`

The `DataBroker` ranks the `ExecutionTarget`s according to how many megabytes of the requested input files that already stored in the cache of the computing resource the `ExecutionTarget` represents. The broker is motivated by that jobs should be submitted to `ExecutionTarget`s where the data already is, thus reducing

[†] http://www.spec.org/cpu2000/CINT2000/

the network load on both the computing resource and client side. The ranking method is based upon the A-REX[‡] interface CacheCheck for querying for the presence of the file in the cache directory. This interface has a limitation in the current implementation and does not support per user caches.

The SortTargets method has four steps:

1. Only the A-REX service has CacheCheck method, thus `ExecutionTarget`s not running A-REX are removed.
2. Information about input files requested in the job description is collected from the `JobInnerRepresentation`.
3. Each `ExecutionTarget` is queried (through the CacheCheck method) for the existence of input files. A single query is used for achieving all the necessary information and file sizes are summarized. If there are problems determining file sizes, then the summarized size will be zero.
4. The possible `ExecutionTarget`s are ranked in a descending order according to the amount of input data they have in their cache.

Example of a CacheCheck request that can be sent to an A-REX service:

```
<CacheCheck>
   <TheseFilesNeedToCheck>
       <FileURL>http://knowarc1.grid.niif.hu/storage/Makefile</FileURL>
       <FileURL>ftp://download.nordugrid.org/test/README</FileURL>
   </TheseFilesNeedToCheck>
</CacheCheck>
```

Example CacheCheck response from the A-REX service:

```
<CacheCheckResponse>
   <CacheCheckResult>
   <Result>
         <FileURL>http://knowarc1.grid.niif.hu/storage/Makefile</FileURL>
         <ExistInTheCache>true</ExistInTheCache>
         <FileSize>190</FileSize>
   </Result>
   <Result>
         <FileURL>ftp://download.nordugrid.org/test/README</FileURL>
         <ExistInTheCache>true</ExistInTheCache>
         <FileSize>176</FileSize>
   </Result>
  </CacheCheckResult>
</CacheCheckResponse>
```

`PythonBroker`

This `PythonBroker` allows users to write their customized broker in python. To use this broker the user should write a python class which should contain:

- an __init__ method that takes a Config object as input, and
- a SortTargets method that takes a python list of ExecutionTarget objects and a JobInnerRepresentation object as input.

[‡]http://www.knowarc.eu/download/D1.2-3_documentation.pdf

The `Config`, `ExecutionTarget` and JobInnerRepresentation classes are available in the swig generated arc python module.

To invoke the `PythonBroker`, the name of the python module defining the broker class and the name of the broker class must be given. If e.g. the broker class MyBroker is defined in the python module SampleBroker the command line option to arcsub to use this broker is:

```
-b PythonBroker:SampleBroker.MyBroker
```

Additional arguments to the python broker can be added by appending them after an additional colon after the python class name:

```
-b PythonBroker:SampleBroker.MyBroker:args
```

Extracting these additional arguments should be done in the python broker class's __init__ method.

For a complete example of a simple python broker see the SampleBroker.py file that comes with your arc python installation.

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.
7. `arcsync` - for synchronising the information in the user's local XML file* about active jobs.
8. `arcresub` - for starting an entire job over on either the same or a new computing resource.
9. `arcmigrate` - for moving a queuing job from one A-REX server to another.

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) {
```

*By default $HOME/.arc/jobs.xml

```
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

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/doc/client/ExecutionTargetMapping.html

Appendix B

Broker mapping

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/doc/client/broker-mapping.ods

Appendix C

Job Inner Representation

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/doc/client/JobInnerRepresentation.ods