LIBARCCONET

A Client Library for ARC

KnowARC
# Contents

1 Preface 5

2 Functionality Overview 7
   2.1 Resource Discovery and Information Retrieval 7
   2.2 Job Submission 8
   2.3 Job Management 9

3 The ARC Brokering Module 11
   3.1 Broker plugins 11
      3.1.1 Benchmark 11
      3.1.2 FastestQueueBroker 12
      3.1.3 Data 12
      3.1.4 PythonBroker 13

4 Job Description 15

5 Grid Flavour Plugins and Commons 17
   5.1 ARC0 Plugins 17
   5.2 ARC1 Plugins 18
   5.3 gLite Plugins 18

A ExecutionTarget 19

B Job Status mapping 21
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 [9] (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, Web Service ARC and gLite [1]) 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. This documentation is not intended to document the developed CLI but the concept of how to build command line tools will be demonstrated. Readers interested in the user interface are recommended to read the client user manual [8].
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. Readers interested in the library API are recommended to look up the online API documentation [3].

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 URLs 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 TargetRetrievers 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.
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.

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.
2.3. JOB MANAGEMENT

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
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).

Figure 2.3: Diagram depicting how job controlling plugins, JobControllers, are loaded and initialized.
Chapter 3

The ARC Brokering Module

The ARC brokering module is made up of two kinds of classes: one brokering base class and specialized classes derived thereof. The brokering base class which 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:

```cpp
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 the ExecutionTarget 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.

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

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

which invokes the SortTargets method implemented by the specialized broker (see Section 3.1) 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 Broker plugins

**Random**

The Random ranks the ExecutionTargets randomly.

#### 3.1.1 Benchmark

The Benchmark ranks the ExecutionTargets according to their benchmark performance. Through the command line tool (see the user manual[8] for reference) this specialized broker takes a user specified benchmark as input and ranks the ExecutionTargets according to their published benchmark performance. If
no benchmark is specified the CINT2000 (Integer Component of SPEC CPU2000) benchmark is used as default.

3.1.2 FastestQueueBroker

The FastestQueueBroker ranks the ExecutionTargets 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 ExecutionTargets depending on their number of free slots/CPUs.

3.1.3 Data

The Data ranks the ExecutionTargets 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 ExecutionTargets where the data already is, thus reducing 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 ExecutionTargets 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 ExecutionTargets 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://example.org/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://example.org/storage/Makefile</FileURL>
      <ExistInTheCache>true</ExistInTheCache>
      <FileSize>190</FileSize>
    </Result>
  </CacheCheckResult>
</CacheCheckResponse>
```


†http://www.knowarc.eu/download/D1.2-3_documentation.pdf
3.1.4 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.

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

Job Description

Since new WS-ARC utilizes a middleware plugin structure, it also need to support the various job description languages used by these middlewares. Therefore the JSDL, XRSL and JDL job description languages are supported. In order to provide support for the three languages listed above, an internal job description data structure is needed which is the union of the supported languages. The internal job description data structure of new WS-ARC is described in a separate document, the ARC Job Description Internal Representation \cite{2}.
Chapter 5

Grid Flavour Plugins and Commons

With a library utilizing plugins a mapping to a common set of job states is needed in order to be able to treat jobs from different middlewares in the same way. In WS-ARC the following job states comprises the state model which plugins should map their job states:

**UNDEFINED** Job state could not be resolved,

**ACCEPTED** Job was accepted by the service,

**PREPARING** Job is preparing,

**SUBMITTING** Job is being submitted to a computing share,

**HOLD** Job is put on hold,

**QUEUING** Job is on computing share waiting to run,

**RUNNING** Job is running on computing share,

**FINISHING** Job is finishing,

**FINISHED** Job has finished,

**KILLED** Job has been killed,

**FAILED** Job failed,

**DELETED** Job have been deleted,

**OTHER** Any job state which does not fit the above states.

See Appendix [B] for the actual job state mapping of each of the below described plugins

5.1 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® [10] GRIS with a custom made ARC schema. The information index server used is the Globus Toolkit® GIIS. Both these servers use the LDAP [14] protocol. The specialization of the TargetRetriever class for ARC0 is implemented using the ARC LDAP Data Management Component (DMC) (see [7] for technical details).

Jobs running on an ARC 0.x computing element are handled by the ARC grid-manager [13]. Job submission and job control are done using the gridftp [5] 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 [5] protocol. This means that proper functionality of the ARC0 plugins requires the gridftp DMC.
5.2 ARC1 Plugins


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.

5.3 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 [4]. 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 BDIs. 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 theSubmitter 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.
Appendix A

ExecutionTarget

# Appendix B

## Job Status mapping

<table>
<thead>
<tr>
<th>Internal</th>
<th>ARC0</th>
<th>ARC1</th>
<th>BES</th>
<th>CREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPTED</td>
<td>ACCEPTED</td>
<td>ACCEPTED</td>
<td>Pending</td>
<td>REGISTERED, PENDING</td>
</tr>
<tr>
<td>PREPARING</td>
<td>PREPARING</td>
<td>PREPARING</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>SUBMITTING</td>
<td>SUBMIT</td>
<td>SUBMIT</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>HOLD</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>HELD</td>
</tr>
<tr>
<td>QUEUING</td>
<td>INLRMS:Q</td>
<td>INLRMS:Q</td>
<td>None</td>
<td>IDLE</td>
</tr>
<tr>
<td>FINISHING</td>
<td>FINISHING</td>
<td>FINISHING</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>FINISHED</td>
<td>FINISHED</td>
<td>FINISHED</td>
<td>Finished</td>
<td>DONE-OK</td>
</tr>
<tr>
<td>KILLED</td>
<td>KILLED</td>
<td>KILLED</td>
<td>None</td>
<td>CANCELLED</td>
</tr>
<tr>
<td>FAILED</td>
<td>FAILED</td>
<td>FAILED</td>
<td>Failed</td>
<td>DONE-FAILED, ABORTED</td>
</tr>
<tr>
<td>DELETED</td>
<td>DELETED</td>
<td>DELETED</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>OTHER</td>
<td>Any other state</td>
<td>Any other state</td>
<td>None</td>
<td>Any other state</td>
</tr>
</tbody>
</table>
Acknowledgements

This work was supported in parts by the EU KnowARC project (Contract nr. 032691) and the EU EMI project (Grant agreement nr. 261611).
Bibliography


