NORDUGRID

NORDUGRID-TECH-12

6/11/2009

ARClib, a client library for ARC

Advanced Resource Connector (ARC) Client Library manual

Contents

Chapter 1

Preface

The NorduGrid's [1] Advanced Resource Connector (ARC) is a light-weight Grid solution, designed to support a dynamic, heterogeneous Grid facility, spanning different computing resources and user communities. It provides *middleware* to interface between user applications and distributed resources.

ARCLib is a client library for ARC written in `C++` with a well-defined API. The library consists of a set of `C++`-classes for

- handling proxy, user and host-certificates.
- performing resource-discovery and resource-querying.
- handling xrsl's.
- doing operations on gridftp-servers like up- and download.
- brokering with the possibility of adding user-defined brokers.
- jobsubmission.

The API is complete and simple enough to be able to perform most client operations.

`ARCLib` makes extensive use of exceptions. There is one exception-class for each class which allows for detailed exception-throwing, catching and handling. Furthermore all exceptions in `ARCLib` derives from the top-level exception, `ARCLibError`, defined in the `error.h` headerfile, so all `ARCLib` exceptions can be caught by catching this exception.

`ARCLib` defines a version macro `ARCVERSION(a,b,c)` that it itself uses to declare its version number. The version number is defined in the file `arc/version.h`. Clients of `ARCLib` can use the version number in the following way

```
#if ARCLIB_VERSION > ARCVERSION(0,1,0)
<do something>
#else
<do something else>
#endif
```

etc. In that way one can write code that works even though the API of `ARCLib` changes.

Other modules can use the macro itself to declare their own version numbers. Updates to modules should be reflected by bumping the version number according to the rule:

- c – Minor changes only. No API changes.
- b – Minor API changes.
- c – Major API changes.

In the following, we will describe the different classes in the library and supplement it with examples.

Chapter 2

Overview

2.1 Notifier

ARCLib comes with its own notifier-class that allows debug-messages to be dynamically printed according to a user-specified output-level. The different output-levels are in order

```
FATAL ERROR WARNING INFO DEBUG VERBOSE
```

Only messages with a higher output-level than the user-specified level will be printed. This means that if for example the user-defined output-level is `WARNING`, only the `WARNING`, `ERROR` and `FATAL` messages will be printed. To use the notifier, add notify statements to your code like:

```
notify(WARNING) << "This is a warning message" << std::endl;
```

```
notify(DEBUG) << "This is a debug message" << std::endl;
```

and set the output-level with

```
SetNotifyLevel(INFO);
```

If no output-level is set by the user, the default output-level is `WARNING`.

The other classes of `ARCLib` uses the notifier internally and thus more debug output from `ARCLib` can be obtained by adjusting the output-level.

2.2 DateTime

The class `Time` defined in the headerfile `datetime.h` is used for storing and manipulating times in the different `TimeFormat`'s specified by the enum

```
enum TimeFormat { MDSTime, ASCTime, UserTime };
```

These formats are of the form given in table 2.2. Passing a string in one of the valid formats to the constructor,

```
Time(std::string);
```

the constructor parses the string and saves the time internally as a `time_t` variable. The time can be printed out again in one of the `TimeFormat`'s using the `str()` method. The `TimeFormat` can be set and retrieved using the static methods

TimeFormat	format-string
MDSTime	YYYYMMDDHHMMSSZ
ASCTime	Day Month DD HH:MM:SS YYYY
UserTime	YYYY-MM-DD HH:MM:SS

```
static void SetFormat(const TimeFormat&);
static TimeFormat GetFormat();
```

Also in the `datetime.h` headerfile are methods for producing a timestamp. Calling the method

```
std::string TimeStamp(const TimeFormat& = UserTime)
```

produces a time-stamp of the current time in the passed `TimeFormat`. Calling

```
std::string TimeStamp(Time, const TimeFormat& = UserTime)
```

produces a time-stamp of the given `Time`-object in the passed `TimeFormat`.

Finally two methods are available for converting a period of seconds to a textual representation and back

```
std::string Period(unsigned long);
long Seconds(const std::string&) throw(TimeError);
```

A standard way of using the `Time` class is the following. One obtains `MDSTime` time-string's from `LdapQuery`'s and wants to convert it to a user-readable format. This can be done trivially with the `Time`-class like this

```
Time sometime("20041118221438Z");
cout << "sometime:  " << Time.str() << endl;
```

2.3 URL

The `URL`-class parses general URL-strings and retrieves information like hostname, protocol, portnumber, path etc. It supports explicitly the following protocols `http, https, ftp, gsiftp, httpg, ldap, rc, rls, file`. To use, pass a URL-string to the constructor of the class and it parses it, retrieving the information into different member-variables. If the URL-string is malformed, an `URLError` exception is malformed.

For example

```
URL url("gsiftp://hathi.hep.lu.se:2811/public/test.dat");
```

parses the given string and places `gsiftp` in the protocol-variable, `hathi.hep.lu.se` in the host-variable, `2811` in the port-variable and `/public/test.dat` in the path-variable.

`ARCLib` uses the `URL`-class internally in most other classes — for resource-discovery, resource-querying, gridftp-server-operations and jobsubmission.

With the `http` protocol, one can specify options after the path like

```
URL url("http://www.nordugrid.org/monitor.php?debug=2&sort=yes");
```

The `URL`-class takes care of parsing such options placing them in the `httpoptions` map-variable.

With the `ldap` protocol, one can specifiy the path in two different ways. One is speciyfing it in the standard basedn way

```
URL url("ldap://grid.uio.no/mds-vo-name=local, o=grid");
```

with `path` equal to `mds-vo-name=local, o=grid`. The other is specifying the `path` as a standard path

```
URL url("ldap://grid.uio.no/o=grid/mds-vo-name=local");
```

The `URL`-class takes care of parsing both of these formats and provides methods, `BaseDN2Path()` and `Path2BaseDN()`, for converting between the two.

```
#include <iostream>
#include <list>
#include <arc/certificate.h>
#include <arc/notify.h>
using namespace std;

int main() {
    try {
        Certificate mycert; // my user certificate
        notify(INFO) << "My certificate:" << endl;
        notify(INFO) << "Subject   : " << mycert.GetSN() << endl;
        notify(INFO) << "Issuer    : " << mycert.GetIssuerSN() << endl;
        notify(INFO) << "ExpiryTime: " << mycert.ExpiryTime() << endl << endl;

        Certificate issuer = mycert.GetIssuerCert();
        notify(INFO) << "My issuer's certificate:'' << endl;
        notify(INFO) << "Subject   : " << issuer.GetSN() << endl;
        notify(INFO) << "Issuer    : " << issuer.GetIssuerSN() << endl;
        notify(INFO) << "ExpiryTime: " << issuer.ExpiryTime() << endl;

        if (issuer.IsSelfSigned())
            notify(INFO) << "The issuer's certificate is self-signed" << endl;
        notify(INFO) << endl;

        list<Certificate> calist = GetCAList();
        list<Certificate>::iterator it;
        notify(INFO) << "List of CA certificates installed:" << endl;
        for (it = calist.begin(); it != calist.end(); it++)
            notify(INFO) << it->GetSN() << endl;
    } catch (CertificateError e) {
        notify(ERROR) << "Error: " << e.what() << std::endl;
    }
}
```

Figure 2.1: An example of using the `Certificate` class for printing information about the user's certificate, the issuer and the list of CA-certificates installed on the machine.

2.4 Certificate

The `Certificate` class provides methods for handling and obtaining information from proxy-, user- and hostcertificates. To use define a `Certificate`-object using the constructor:

```
Certificate cert(certtype type = USERCERT, std::string filename = "");
```

where `certtype` is an enum defined by

```
enum certtype { PROXY, USERCERT, HOSTCERT };
```

specifying the certificate-type. If the filename is not given, the constructor looks for the certificate specified in standard places like `$HOME/.globus/usercert.pem` for user-certificates.

The constructor parses the certificate and stores information about it in the object. This can then be accessed by the user. Useful methods are

```
std::string GetSN(SNFormat format = PLAIN);
```

returning the subject-name of the certificate in the format specified. `SNFormat` is an enum with values

```
enum SNFormat { PLAIN, X509, LDAP1, LDAP2 };
```

Other useful methods include

```
std::string GetIssuerSN(SNFormat format = PLAIN);
```

returning the subject-name of the issuer of the certificate,

```
std::string GetCertFilename();
```

returning the filename in which the certificate is stored,

```
std::string ExpiryTime();
```

returning the expiry-time of the certificate,

```
std::string ValidFor();
```

returning a text respresentation of the validity time of the certificate and

```
bool IsSelfSigned();
```

returning a boolean specifying whether the certificate is self-signed or not.

The full list of CA-certificates installed on the computer can be obtained through the

```
std::list<Certificate> GetCAList();
```

method. This can be used to check whether a given CA-certificate is already installed on the computer.

An example of using the `Certificate`-class is shown in figure 2.1.

2.5 LdapQuery

`ARCLib` provides an `LdapQuery` class for performing ldap-queries. It is basically a simple `C++`-wrapper around the ldap API extended with a method to perform parallel ldap-queries to several servers at once and is defined in the headerfile `ldapquery.h`.

To perform an ldap-query, first construct an `LdapQuery` object using the `LdapQuery` constructor

```
LdapQuery(const std::string& ldaphost,
          int ldapport,
          bool anonymous = true,
          const std::string& usersn = "",
          int timeout = 20);
```

where `hostname` is the hostname of the ldapserver, `port` is the corresponding port-number, `anonymous` specifies whether the query should be anonymous, `usersn` is the `SN` of the user querying the server and `timeout` is the timeout. By default `port` is equal to the default ldapserver port, 389, while the rest corresponds to an anonymous query with a timeout of 20 seconds.

Use this object to query the server using

```
Query(const std::string& basedn,
      const std::string& filter,
      const std::vector<std::string>& attrs,
      Scope scope);
```

where `basedn` is the `DN` of the entry at which to start the search, `filter` is a string representation of the filter to apply, `attrs` is a null-terminated array of attributes to return from search and `scope` is the scope of the search with value one of `base`, `one-level` or `subtree`. `filter, attrs` and `scope` has sensible default values for a full-scale search on all attributes.

To retrieve the results of the query, call Result

```
LdapQuery::Result(ldap_callback callback, void* ref);
```

where `ldap_callback` is a pointer to a callback-function with signature

```
void (*ldap_callback)(const std::string& attr, const std::string& value, void* ref);
```

and a user-defined void-pointer which is passed to the callback-function. The callback is called for each line in the ldap-response with the corresponding attribute and its value. The void-pointer can then be used to store information about the attributes and values.

A typical way of using the LdapQuery class is to first define a `URL` that holds the server, port and basedn of the query like

```
URL url("ldap://quark.hep.lu.se/o=grid/mds-vo-name=local");
```

and use it like this

```
LdapQuery ldapq(url.Host(), url.Port());
ldap.Query(url.BaseDN());
ldap.Result(callback, this);
```

For an example of how to use the LdapQuery class, see figure 2.2. This piece of code queries the `NorduGrid VO` ldapserver, parses the response in `Callback`-function pulling out the VO-members `DN` and puts them in the passed pointer. After the call to `ldapq.Result()`, the `vosubjects` vector contains all the `DN`'s of the NorduGrid VO.

When performing ldap-queries to several clusters at once, it is too slow to query and gather results for every cluster after the other. Therefore a method for performing parallel ldap-queries is provided. It works by querying all the clusters first and then only gathering the results. This way of doing it can lead to a drastic increase in performance of such a ldap-query. The parallel ldap-query method looks like this:

```
void PerformLdapQueries(std::list<URL> clusters,
                        std::string filter,
                        std::vector<std::string> attrs,
                        ldap_callback callback,
                        void* ref,
                        LdapQuery::Scope scope = LdapQuery::subtree,
#include <string>
#include <iostream>
#include <vector>
#include <arc/url.h>
#include <arc/ldapquery.h>
#include <arc/notify.h>
using namespace std;

void Callback(const string& attr, const string& value, void *ref) {
    if (attr!="description") return;
    vector<string>* vo = (vector<string>*)ref;

    string val(value.substr(8));
    while (val[0]==' ') val = val.substr(1);
    vo->push_back(val);
}

int main() {
    SetNotifyLevel(DEBUG);

    URL NGVO("ldap://grid-vo.nordugrid.org/dc=org/dc=nordugrid/ou=people");

    LdapQuery ldapq(NGVO.Host(), NGVO.Port());
    notify(INFO) << "Querying " << NGVO.str() << " for the NorduGrid VO" << endl;

    vector<string> vosubjects;
    try {
        ldapq.Query(NGVO.BaseDN());
        ldapq.Result(Callback, (void*)&vosubjects);
    } catch (LdapQueryError e) {
        notify(ERROR) << "Error: " << e.what() << endl;
        return 1;
    }

    cout << "Members of the NorduGrid VO:" << endl;
    for (int i=0; i<vosubjects.size(); i++)
        cout << vosubjects[i] << endl;
}
```

Figure 2.2: Example of how to use the LdapQuery class to obtain the members of the NorduGrid VO by by querying the NorduGrid VO ldapserver.

```
                          std::string usersn = '',
                          bool anonymous = true,
                          int timeout = 20);
```

where one specifies the list of clusters to query together with the standard set of ldap-variables. This method is used extensively by the resource-discovery and resource-query methods below.

2.6 Resource-discovery

Resource-discovery is the process of finding resources be it clusters, storage-elements or the like. In NorduGrid all resources register themselves to at least one `GIIS` – either a country-level `GIIS` or a VO-specific `GIIS`. `ARCLib` supports querying one or several `GIIS`'es for the resources that register to these `GIIS`'es using a standard `LdapQuery`.

Resource-discovery is done through methods defined in the `mdsdiscovery.h` header file.

```
std::list<URL> GetResources(std::list<URL> giis_urls = std::list<URL>(),
                            resource id = cluster,
                            bool anonymous = true,
                            std::string usersn = "",
                            int timeout = 20);
```

and a similar one taking only one `GIIS-URL`-string. These methods returns a list of contact ldap-urls for the resources registering to the `GIIS`'es. By default, the methods finds clusters but setting the `id`-variable, which is a `resource`-enum defined by

```
enum resource { cluster, storageelement, replicacatalog };
```

to either `storageelement` or `replicacatalog` the methods will find StorageElements or ReplicaCatalogs instead.

Simple helper methods, `GetClusters, GetSEs` and `GetRCs` are also provided:

```
std::list<URL> GetClusters(std::list<URL> giis_urls = std::list<URL>(),
                           bool anonymous = true,
                           std::string usersn = "",
                           int timeout = 20);

std::list<URL> GetSEs(std::list<URL> giis_urls = std::list<URL>(),
                      bool anonymous = true,
                      std::string usersn = "",
                      int timeout = 20);

std::list<URL> GetRCs(std::list<URL> giis_urls = std::list<URL>(),
                      bool anonymous = true,
                      std::string usersn = "",
                      int timeout = 20);
```

They work exactly like the `GetResources` method except that one does not have to explictly specify the type of resource to query.

If during a `GIIS`-query, other `GIIS`'es are found, these `GIIS`'es are also queried for resources. In this way, a whole tree of `GIIS`'es are queried.

If the list of `URL`'s passed to `GetResources` is empty, the following files are read in order for a `GIIS`-list: `$HOME/.nggiistlist`, `$NORDUGRID_LOCATION/etc/giislist` and `/etc/giislist`. If no `GIIS`-list is found, a `GIIS`-exception is thrown.

Figure 2.3 shows an example of finding the clusters that registers to the `ATLAS GIIS`.

2.7 Resource-querying

Having obtained a list of resources through resource-discovery as described above, be it clusters or storageelements, one can continue and query each of these resources for information. There are several methods defined in `mdsquery.h` that can do this. All these methods fill in the `Cluster`, `Queue`, `Job`, `User`, `StorageElement` or `ReplicaCatalog` structures defined in `resource.h` for later processing by the user.

The methods all take a list of ldap-contact `URL`'s for the resources, an mds-filter specially suited for the particular query (see below for examples), a boolean specifying whether the query should be anonymous and if not the user's `DN` and finally a timeout variable.

```
#include <list>
#include <iostream>
#include <arc/mdsdiscovery.h>
#include <arc/url.h>
using namespace std;

int main() {
    URL atlasgiis("ldap://atlasgiis.nbi.dk:2135/o=grid/mds-vo-name=Atlas");

    list<URL> clusters;
    try {
        clusters = GetClusters(atlasgiis);
    } catch (MDSDiscoveryError e) {
        cout << "Error: " << e.what() << endl;
        return 1;
    }

    cout << "The following clusters are registering to the ATLAS GIIS:" << endl;
    list<URL>::iterator it;
    for (it = clusters.begin(); it != clusters.end(); it++)
        cout << it->Host() << endl;
}
```

Figure 2.3: Example of how to use the `GetClusters` call to obtain the list of clusters that register to the `ATLAS GIIS`.

Technically all the methods use the `PerformLdapQueries` method from `ldapquery.h` for performing the parallel ldap-queries.

The `Cluster`-methods in `mdsquery.h` are

```
Cluster GetClusterInfo(const URL& cluster,
                       std::string filter = MDS_FILTER_CLUSTERINFO,
                       const bool& anonymous = true,
                       const std::string& usersn = "",
                       unsigned int timeout = 20);
```

and

```
std::list<Cluster> GetClusterInfo(std::list<URL> clusters = std::list<URL>(),
                                  std::string filter = MDS_FILTER_CLUSTERINFO,
                                  const bool& anonymous = true,
                                  const std::string& usersn = "",
                                  unsigned int timeout = 20);
```

These methods returns a `Cluster`-object with information filled in for each cluster queried. The mdsfilter `MDS_FILTER_CLUSTERINFO` is defined to be the ldapquery filter

```
"(|(objectclass=nordugrid-cluster)(objectclass=nordugrid-queue)(nordugrid-authuser-sn=%s))"
```

with the user's `DN` replaced at the end. Similar methods are

```
std::list<Queue> GetQueueInfo(const URL& cluster,
                              std::string filter = MDS_FILTER_CLUSTERINFO,
                              const bool& anonymous = true,
                              const std::string& usersn = "",
                              unsigned int timeout = 20);
```

and

```
std::list<Queue> GetQueueInfo(std::list<URL> clusters = std::list<URL>(),
                              std::string filter = MDS_FILTER_CLUSTERINFO,
                              const bool& anonymous = true,
                              const std::string& usersn = "",
                              unsigned int timeout = 20);
```

These methods performs the same queries as above but rearranges the result before returning it to the user. For each `Cluster`, all `Queue`'s are extracted and the cluster-information for these `Queue`'s are added explicitly.

Completely similar methods as above for `StorageElement`'s and `ReplicaCatalog`'s exist e.g.

```
std::list<StorageElement> GetSEInfo(std::list<URL> urls = std::list<URL>(),
                                    std::string filter = MDS_FILTER_SEINFO,
                                    const bool& anonymous = true,
                                    const std::string& usersn = "",
                                    unsigned int timeout = 20);

std::list<ReplicaCatalog> GetRCInfo(const URL& url,
                                    std::string filter = MDS_FILTER_RCINFO,
                                    const bool& anonymous = true,
                                    const std::string& usersn = "",
                                    unsigned int timeout = 20);
```

There are also methods for obtaining job-information. The `GetAllJobs` methods finds all jobs running by the user on a given set of clusters

```
std::list<Job> GetAllJobs(std::list<URL> clusters = std::list<URL>(),
                          bool anonymous = true,
                          const std::string& usersn = "",
                          unsigned int timeout = 20);

std::list<Job> GetAllJobs(const URL& cluster,
                          bool anonymous = true,
                          const std::string& usersn = "",
                          unsigned int timeout = 20);
```

while the `GetJobInfo` methods returns job-information about specific jobs

```
std::list<Job> GetJobInfo(std::list<std::string> jobids,
                          std::string filter = MDS_FILTER_JOBINFO,
                          const bool& anonymous = true,
                          const std::string& usersn = ",
                          unsigned int timeout = 20);
```

```
Job GetJobInfo(std::string jobid,
               std::string filter = MDS_FILTER_JOBINFO,
               const bool& anonymous = true,
               const std::string& usersn = '',
               unsigned int timeout = 20);
```

2.8 FTPControl

The `FTPControl` class provides methods for interacting with gridftp-servers. There are methods for downloading and uploading files, downloading a complete directory, listing directories and getting sizes of files as well as sending general commands to the gridftp-server. All methods take a timeout-parameter and a parameter specifying whether the operation should disconnect from the gridftp-server after the operation. This makes nesting of operations on the same server easier without having to connect all the time.

```
void Upload(const std::string& localfile,
            const URL& url,
            bool disconnectafteruse = true,
            int timeout = 20) throw(FTPControlError);
```

uploading the local file `localfile` to the location pointed to by `url`,

```
void Download(const URL& url,
              const std::string& localfile = "",
              bool disconnectafteruse = true,
              int timeout = 20) throw(FTPControlError);
```

downloading the file specified by the `url` to the local file `localfile`,

```
unsigned long long Size(const URL& url,
                        bool disconnectafteruse = true,
                        int timeout = 20) throw(FTPControlError);
```

returning the size of the file pointed to by the `url`. The `ListDir` and `RecursiveListDir` methods return a list of `FileInfo` with information about files in the given directory and — for the `RecursiveListDir` method — all subdirectories thereof. The `FileInfo` structure looks like this

```
struct FileInfo { std::string filename,
                  unsigned long long size,
                  bool isdir };
```

while the two listing methods looks like this

```
std::list<FileInfo> ListDir(const URL& url,
                            bool disconnectafteruse = true,
                            int timeout = 20) throw(FTPControlError);
```

```
std::list<FileInfo> RecursiveListDir(const URL& url,
                                     bool disconnectafteruse = true,
                                     int timeout = 20) throw(FTPControlError);
```

The method `DownloadDirectory` downloads all files from a directory including all files in subdirectories thereof. It uses the `RecursiveListDir` method for obtaining all files and downloads each in turn

```
void DownloadDirectory(const URL& url,
                       const std::string& localdir = "",
                       bool disconnectafteruse = true,
                       int timeout = 20) throw(FTPControlError);
```

2.9 Xrsl

A xrsl consists of a number of xrsl-relations with attributes and values. Manipulations of these using the Globus RSL API is both cumbersome and non-intuitive. Therefore two wrapper-classes for xrsl-handling — the `Xrsl`- and `XrslRelation` classes — have been constructed. These make xrsl-handling much easier.

2.9.1 `XrslRelation`-class

`XrslRelation`'s can be constructed using the `XrslRelation` constructors. The standard one is (as shown above)

```
XrslRelation(const std::string&, const xrsl_operator&, const std::string&);
```

where the first argument is the, the second the actual `xrsl_operator` to put in and the last is the value of the relation. The `xrsl_operator` is an enum taking the values

```
enum xrsl_operator { operator_eq = GLOBUS_RSL_EQ,
                     operator_neq = GLOBUS_RSL_NEQ,
                     operator_gt = GLOBUS_RSL_GT,
                     operator_gteq = GLOBUS_RSL_GTEQ,
                     operator_lt = GLOBUS_RSL_LT,
                     operator_lteq = GLOBUS_RSL_LTEQ,
                     operator_and = GLOBUS_RSL_AND,
                     operator_or = GLOBUS_RSL_OR,
                     operator_multi = GLOBUS_RSL_MULTIREQ };
```

where the first six are applicable for `XrslRelation`'s. The following

```
XrslRelation rel("executable", operator_eq, "/bin/echo");
```

constructs the relation

```
(exectable=/bin/echo)
```

To construct a `XrslRelation` with a list-value use the constructor

```
XrslRelation(const std::string&, const xrsl_operator&, const std::list<std::string>&);
```

which constructs a `XrslRelation` with a list of values — like

```
(arguments=1 2 3 4 5)
```

To construct a `XrslRelation` with a double list-value, use the constructor

```
XrslRelation(const std::string&, const xrsl_operator&, const std::list<std::list<
std::string> >&);
```

which constructs a `XrslRelation` of the form

```
(inputfiles=("input1" "fil.txt")("input2" "fil2.txt"))
```

The values of a given `XrslRelation` can be retrieved again using one of the three methods

```
std::string GetSingleValue() throw(XrslError);
std::list<std::string> GetListValue() throw(XrslError);
std::list<std::list<std::string> > GetDoubleListValue() throw(XrslError);
```

2.9.2 `Xrsl`-class

The `Xrsl`-class provides several methods for manipulating `XrslRelation`-objects in the xrsl. One can construct an `Xrsl` using either the string-constructor

```
Xrsl(const std::string& xrsl);
```

or

```
Xrsl(xrsl_operator = operator_and);
```

that constructs an empty `Xrsl` with a leading `xrsl_operator` — usually `operator_and`. For example

```
Xrsl xrsl("&(executable=/bin/echo)(arguments=hello,grid)");
```

constructs an `Xrsl`-object containing the given xrsl.

There are methods in the `Xrsl`-class for retrieving the different `XrslRelation`'s and their values, add extra relations, remove relations and so on. First of all, one can check if the xrsl contains a given attribute — with the `IsRelation` method

```
bool IsRelation(const std::string& attribute);
```

It returns true if a relation in the xrsl has the corresponding attribute. False if not.

Retrieving relations is done with the `GetRelation` or `GetAllRelations` methods. The first method

```
XrslRelation GetRelation(const std::string& attribute) throw(XrslError);
```

returns the first relation found with the specified attribute. It throws an exception if no relation with the specified attribute exists. The second method

```
std::list<XrslRelation> GetAllRelations(const std::string& attribute);
```

returns a list of all relations with the specified attribute. Adding extra relations is done with the `AddRelation` and the `AddSimpleRelation` methods. The `AddRelation` method

```
void AddRelation(const XrslRelation&, bool force = true) throw(XrslError);
```

adds the given relation — forcing it (if `force` is set to true) if another relation with the same attribute already exist. `AddSimpleRelation`

```
void AddSimpleRelation(const std::string& attr,
                       xrsl_operator op,
                       const std::string& value,
                       bool force = true) throw(XrslError);
```

works in the same way except that the user explicitly specifies attribute, operator and value. Finally the user can remove relations with the `RemoveRelation` method

```
void RemoveRelation(const std::string& attr) throw(XrslError);
```

which removes the first relation with the specified attribute.

2.10 Brokering

`ARCLib` comes with its own extensible brokering-framework. Brokering is the process of choosing the right target for jobsubmission from a list of available targets. In `ARC` a jobsubmission-target is basically a batch-queue on some cluster. Information about such queues is obtained very easily with the `GetQueueInfo()`-method described earlier. The `Queue`'s returned from this method are potential targets and brokering is thus the process of choosing the right `Queue` for the job based on the user submitting the job and her job-specification.

After having obtained a list of possible `Queue`'s through the `GetQueueInfo()`-method, one should proceed to construct the list of possible targets with

```
std::list<Target> ConstructTargets(std::list<Queue> queues,
                                   Xrsl axrsl = Xrsl());
```

passing the list of `Queue`'s and the `Xrsl`. A `Target` is basically a `Queue` with the user-supplied `Xrsl` included. This `Xrsl` is needed to be able to perform brokering on the `Target`'s.

The `ConstructTargets` method above performs a few simple checks on the `Queue`'s. It checks whether the user is at all authorized to run jobs on the cluster in question, whether the `Queue` is active, has any CPU's at all and that the number of queued jobs is not larger than the maximum number of allowed queued jobs. Finally it checks that the cluster's CA-certificate is in fact installed locally on the user's machine (this is needed for authentication with the cluster). If any of these criteria is not met, the `Queue` is not included in the list of `Target`'s.

Brokering can now be performed by calling `PerformStandardBrokering` from `standardbrokers.h`

```
void PerformStandardBrokering(std::list<Target>& targets);
```

This method performs the standard brokering over a set of `Xrsl`-attributes including `cluster`, `queue`, `cputime`, `memory`, `count`, `nodeaccess`, `architecture` plus the `RuntimeEnvironment`-attributes, `middleware`, `runtimeenvironment` and `opsys`. Finally it sorts the remaining targets in a preferred order first checking that the remaining targets has enough free CPU's for the job. If several targets has enough free CPU's for the job, the list is sorted after the fastest CPU-speed.

The `PerformStandardBrokering` method returns the list of possible submission-targets sorted so that the most suitable target (after the criteria mentioned above) is first and the least suitable last.

The user can also write her own broker and include it in the brokering. This requires writing a broker inheriting from the virtual `Broker`-class in `broker.h` and pass the broker to the `PerformBrokering`-method

```
void PerformBrokering(std::list<Broker*> brokers,
                      std::list<Target>& targets);
```

This method performs brokering using the broker(s) passed. It can advantageously be used after a call to `PerformStandardBrokering` since `PerformStandardBrokering` performs the necessary brokering that has to be done anyway.

2.11 Job-submission

Job-submission is done quite simply: First call `PrepareJobSubmission` from `jobsubmission.h`:

```
std::list<Target> PrepareJobSubmission(Xrsl axrsl) throw(MDSQueryError, XrslError);
```

This method calls in turn the methods `GetQueueInfo()`, `ConstructTargets()`, `PerformXrslValidation()` and `PerformStandardBrokering()` and returns the list of targets obtained from the brokering step. One can of course also call the four methods oneself with the same result.

Now call `SubmitJob` with the `Xrsl` and the list of targets:

```
std::string SubmitJob(Xrsl axrsl,
                      std::list<Target> targetlist,
                      bool dryrun = false) throw(JobSubmissionError, XrslError);
```

This method submits the job specified by the `Xrsl` to the first target in the list of targets. If jobsubmission to the first target fails for some reason, the next target in the list is tried and so on. If jobsubmission succeeds, the jobid of the job is returned.

Note that before submitting the job and uploading the `Xrsl`, `SubmitJob` rewrites the `Xrsl`. It adds several attributes and rewrites others. For example, it adds `queue` with the name of the chosen target, adds `hostname` with the local hostname and rewrites the `executables` attribute to make sure that the `executable` is in this list. Several other attributes are added and rewritten — all such that the grid-manager on the chosen cluster can interpret the `Xrsl` correctly.

2.12 JobControl

Technically jobsubmission uses the `JobFTPControl`-class for submitting jobs. `JobFTPControl` derives from `FTPControl` and extends it with several job-specific gridftp-methods. Those include methods for killing jobs, cleaning jobs and renewing credentials for jobs. These methods are public and thus accessible but the following helper methods can be used instead

```
void CancelJob(const std::string& jobid) throw(JobFTPControlError, FTPControlError);
```

for cancelling jobs,

```
void CleanJob(const std::string& jobid) throw(JobFTPControlError, FTPControlError);
```

for cleaning jobs and

```
void RenewCreds(const std::string& jobid) throw(JobFTPControlError, FTPControlError);
```

for renewing credentials for jobs.

2.13 Job-listing

The `ARC` User-Interface writes the jobids and job-names of all submitted jobs to the file `.ngjobs` in the user's home-directory and it removes them again when the user cleans finished jobs. This functionality is provided by three methods in the `joblist.h` header file. The method `GetJobIDs`,

```
std::list<std::string> GetJobIDs(bool all = true,
       const std::list<std::string>& jobs = std::list<std::string>(),
       const std::list<std::string>& clusterselect = std::list<std::string>(),
       const std::list<std::string>& clusterreject = std::list<std::string>());
```

extracts all the jobids in the `.ngjobs` file consistent with the search criteria specified by the user. If `all` is true, all jobids are extracted. Otherwise jobids consistent with the `jobs` parameter, which can contain a regular expression with wildcards and the `clusterselection` and `clusterrejection` parameters are extracted.

The jobid and jobname of a new job is added with the `AddJobID` method

```
void AddJobID(const std::string& jobid, const std::string& jobname);
```

and a jobid is removed again with the `RemoveJobID` method

```
void RemoveJobID(const std::string& jobid);
```

Chapter 3

ARCLib and python

ARCLib uses SWIG to wrap the ARCLib methods and classes in python. In that way the whole API is exposed in a very simple way to python. In this chapter we will give several examples of how to import the ARCLib python module and use it from within python.

3.1 Importing the ARCLib module

ARCLib can be imported by setting the PYTHONPATH environment variable to /opt/nordugrid/lib or by

```
arc:~$ python
Python 2.3 (#2, Aug 31 2003, 17:27:29)
[GCC 3.3.1 (Mandrake Linux 9.2 3.3.1-1mdk)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path.append("/opt/nordugrid/lib")
>>> from arclib import *
>>>
```

3.2 Certificates

Certificates are handled very easily. To get the user's certificate, construct an object of the Certificate-class and call methods on it

```
>>> c = Certificate()
>>> c.GetSN()
'/O=Grid/O=NorduGrid/OU=nbi.dk/CN=Jakob Langgaard Nielsen'
>>> c.GetCertFilename()
'/home/langgard/.globus/usercert.pem'
>>> c.ValidFor()
'9 weeks, 1 day, 13 hours, 21 minutes, 12 seconds'
>>>
```

Proxy-certificates are handled by including the specifier PROXY in the constructor:

```
>>> c = Certificate(PROXY)
>>> c.GetSN()
'/O=Grid/O=NorduGrid/OU=nbi.dk/CN=Jakob Langgaard Nielsen/CN=proxy'
>>> c.GetCertFilename()
'/tmp/x509up_u500'
>>> c.ValidFor()
```

```
'expired'
>>> c.IsExpired()
True
>>>
```

The full list of installed CA-certificates is retrieved using the `GetCAList`-method.

```
>>> cas = GetCAList()
>>> for i in cas:
...     print i
...
/O=Grid/O=NorduGrid/CN=NorduGrid Certification Authority
/C=DE/O=GermanGrid/CN=GridKa-CA
/C=CH/O=CERN/OU=GRID/CN=CERN CA
/C=IT/O=INFN/CN=INFN Certification Authority
>>>
```

3.3 Resource-discovery

Resource-discovery can be done through the `GetClusterResources()` and `GetSEResources()` methods. These methods take a list of `GIIS`'es to query but if left out, the default list of top-level NorduGrid `GIIS`'es (as specified in the file `.nggiislist`) are used. So to obtain the LDAP contact URL's of all clusters registering to these `GIIS`'es, use

```
>>> cls = GetClusterResources()
>>> len(cls)
49     (number of clusters registering to the NorduGrid top-level GIIS'es)
>>> print cls[0]
ldap://alice.grid.upjs.sk:2135
>>>
```

To obtain the LDAP contact URL's of all storage elements registering to the `ATLAS GIIS`, use

```
>>> atlasgiis = URL('ldap://atlasgiis.nbi.dk:2135/o=grid/mds-vo-name=Atlas')
>>> ses = GetSEResources(atlasgiis)
>>> len(ses)
16
>>> print ses[0]
ldap://atlas.hpc.unimelb.edu.au:2135
>>>
```

3.4 Resource-querying

Having obtained a list of cluster- or storage-contact URL's, one can proceed and query them. To do this, use `GetClusterInfo` or `GetSEInfo` like this

```
>>> cinfo = GetClusterInfo(cls)
>>> totalcpus = 0
>>> for i in cinfo:
>>> totalcpus = 0
>>> for i in cls:
...     totalcpus += i.total_cpus
...
>>> print totalcpus
5473
>>>
```

One can also call `GetClusterInfo()` with no arguments. In that case a list of contact URL's is retrieved by doing a standard resource-discovery first. This means that the single call

```
>>> cinfo = GetClusterInfo()
```

does full resource-discovery and resource-querying at the same time!

There are also methods for retrieving information about jobs. For example, `GetAllJobs()` retrieves information about all jobs run by the user.

```
>>> jobinfo = GetAllJobs()
>>> for i in jobinfo:
...     print i.id, i.status
...
gsiftp://atlas.fzk.de:2811/jobs/3095311148934971167065950 FINISHED
gsiftp://atlas.fzk.de:2811/jobs/2855711148934451560148339 FINISHED
gsiftp://benedict.grid.aau.dk:2811/jobs/63611142436961760568961 DELETED
gsiftp://hagrid.it.uu.se:2811/jobs/861711148927331730484646 FINISHED
>>>
```

Finally `GetQueueInfo()` retrieves cluster information like `GetClusterInfo()` but reorganizes the result so that the returned is a list of `Queue`'s.

3.5 FTPControl

`ARCLib` provides a set of methods for interacting with gridftp-servers. For example for listing directories and up- and downloading files.

```
>>> ftpc = FTPControl()
>>> dir = "gsiftp://quark.hep.lu.se/demo"
>>> files = ftpc.ListDir(dir)
>>> print files[0]
/demo/test  80  dir
>>> print files[2]
/demo/figure10.pnm  176200  file
>>> file = dir + "/figure10.pnm"
>>> ftpc.Size(file)
176200
>>> ftpc.Download(file, "figure10.pnm")
>>>
```

3.6 Job-submission

Job-submission is done in a series of steps, define a `Xrsl`, do resource-querying and brokering and then perform the jobsubmission. The following is an example of how it is done

```
>>> xrsl = Xrsl("&(executable=/bin/echo)(arguments=\"hello, grid\")")
>>> qi = GetQueueInfo()
>>> targets = ConstructTargets(qi, xrsl)
>>> targetsleft = PerformStandardBrokering(targets)
>>> jobid = SubmitJob(xrsl, targetsleft)
>>> jobid
'gsiftp://atlas.fzk.de:2811/jobs/7281114978962272805613'
>>>
```

The `GetQueueInfo()`, `ConstructTargets()` and `PerformStandardBrokering()` steps can also be done with the single call to `PrepareJobSubmission()` after having defined the `Xrsl`.

```
>>> xrsl = Xrsl("\&(executable=/bin/echo)(arguments=\"hello, grid\")")
>>> targets = PrepareJobSubmission(xrsl)
>>> jobid = SubmitJob(xrsl, targets)
>>> jobid
'gsiftp://morpheus.dcgc.dk:2811/jobs/17104111497912861773299l'
>>>
```

Bibliography

[1] The NorduGrid Project. [Online]. Available: http://www.nordugrid.org