

Table of Contents

CREAM Client development tutorial - EMI-1 version.....	1
Preliminaries: build machine configuration.....	1
CLI Development.....	2
Building of the example source code (see attachments).....	2
COMPILATION:	2
LINKING:	2
Quick overview of the API.....	2
Step by step Job Submission explanation.....	3
Proxy Delegation.....	3
Job Registration.....	3
Job start.....	4
Cancelling, Suspending, Resuming and Purging jobs.....	5
Listing jobs in a CREAM CE.....	5
Getting Information about jobs submitted to a CREAM CE.....	5
Fast info query (JobStatus).....	5
Slow info query (JobInfo).....	5
Enabling and disabling the Job Submission Service.....	6
Renewing a Delegated Proxy.....	6
Querying job status events.....	6
Getting Service Information from a deployed CREAM CE running service.....	6
Contact info.....	7

CREAM Client development tutorial - EMI-1 version

Preliminaries: build machine configuration

Before developing a CREAM's CLI you must configure properly a SL5_64bit machine with EPEL repositories and basic EMI1 software. Instructions are here:

http://wiki.italiangrid.org/twiki/bin/view/CREAM/SystemAdministratorGuideForEMI1#1_3_Installation .

Here are summarized all the steps described in the link above plus some more rpm to install and other settings:

```
\rm /etc/yum.repos.d/dag.repo
rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/epel-release-5-4.noarch.rpm
wget http://emisoft.web.cern.ch/emisoft/dist/EMI/1/sl5/x86_64/base/emi-release-1.0.0-1.sl5.noarch.rpm
yum -y install ./emi-release-1.0.0-1.sl5.noarch.rpm
wget http://repository.egi.eu/sw/production/cas/1/current/repo-files/egi-trustanchors.repo -O /etc/yum.repos.d/emi-trustanchors.repo
wget http://forge.cnaf.infn.it/frs/download.php/101/disable_yum.sh
chmod +x disable_yum.sh
./ disable_yum.sh
yum clean all; yum update all
yum -y install yum-protectbase.noarch
yum -y install ca-policy-egi-core
yum -y install glite-ce-cream-client-api-c.x86_64
yum -y install glite-lbjp-common-gsoap-plugin
yum -y install boost-devel
yum -y install voms-devel voms-clients
```

Then, download, build and install gSOAP 2.7.6b from this URL:

<https://etics-repository.cern.ch/repository/download/registered/externals/gsoap/2.7.6b/src/>

Now create these symlinks:

```
ln -s /usr/lib64/liblog4cpp.so.4 /usr/lib64/liblog4cpp.so
ln -s /usr/lib64/libclassad.so.1 /usr/lib64/libclassad.so
ln -s /usr/lib64/libclassad_ns.so.1 /usr/lib64/libclassad_ns.so
ln -s /usr/lib64/libcares.so.2 /usr/lib64/libcares.so
ln -s /usr/lib64/libglobus_common.so.0 /usr/lib64/libglobus_common.so
ln -s /usr/lib64/libglobus_gsi_callback.so.0 /usr/lib64/libglobus_gsi_callback.so
ln -s /usr/lib64/libglobus_gssapi_gsi.so.4 /usr/lib64/libglobus_gssapi_gsi.so
ln -s /usr/lib64/libglobus_gsi_credential.so.1 /usr/lib64/libglobus_gsi_credential.so
ln -s /usr/lib64/libltdl.so.3 /usr/lib64/libltdl.so
ln -s /usr/lib64/libglobus_oldgaa.so.0 /usr/lib64/libglobus_oldgaa.so
ln -s /usr/lib64/libglobus_proxy_ssl.so.1 /usr/lib64/libglobus_proxy_ssl.so
ln -s /usr/lib64/libglobus_openssl_error.so.0 /usr/lib64/libglobus_openssl_error.so
ln -s /usr/lib64/libglobus_gsi_proxy_core.so.0 /usr/lib64/libglobus_gsi_proxy_core.so
ln -s /usr/lib64/libglobus_gsi_cert_utils.so.0 /usr/lib64/libglobus_gsi_cert_utils.so
ln -s /usr/lib64/libglobus_gsi_sysconfig.so.1 /usr/lib64/libglobus_gsi_sysconfig.so
ln -s /usr/lib64/libglobus_openssl.so.0 /usr/lib64/libglobus_openssl.so
```

CLI Development

Building of the example source code (see attachments)

In the attachments you can find small examples that explain how to use the CREAM Client API C++. You can build them by issuing the commands:

COMPILATION:

```
g++ -c -I/usr/local/include <EXAMPLE>.cpp
```

LINKING:

```
g++ -o <EXAMPLE> <EXAMPLE>.o -lglite_security_gsoap_plugin_276b \
-lglite_ce_cream_client_soap -lglite_ce_cream_client_util \
/usr/local/lib/libgsoap++.a -lboost_thread -llog4cpp \
-lvomsapi -lclassad_ns -lgridsite -lboost_regex
```

Quick overview of the API

Currently, the C++ API implements the following operations:

- JobRegister
- JobStart
- JobCancel
- JobSuspend
- JobResume
- JobPurge
- JobList
- JobInfo
- JobStatus
- ProxyDelegation
- ProxyRenew
- ServiceInfo
- Enable/Disable JobSubmission
- Query JobSubmission enable status

The API architecture is founded on a super and abstract class `AbsCreamProxy` that exposes 3 relevant public methods: `AbsCreamProxy::setCredential(...)`, `AbsCreamProxy::setConnectionTimeout(...)`, `AbsCreamProxy::execute(...)`. The method `execute(...)` is pure virtual.

There're several `AbsCreamProxy`'s subclasses, one for each operation mentioned above: `CreamProxy _Start`, `CreamProxy _Register`, `CreamProxy _Cancel`, and so on. Each of them implements its own `execute(...)` method, specific for the kind of operation the subclass itself represents; this implementation is responsible for the connection to the remote Web Service (CREAM), sending the SOAP request, receiving the SOAP response and unserializing it and hides the user from any SOAP communication and authentication detail. All the subclasses have protected constructors, so the developer cannot directly create a subclass, but she/he must use a special factory named `CreamProxyFactory`. For each operation the factory has a `make_<OPERATION_NAME>(...)` method. The factory's 'make' methods return a pointer to an `AbsCreamProxy` object (polimorphic object) that allows the developer to handle more instances of different subclasses in the same way (for example to put all of them in a container). The 'make' methods take different parameters depending on the kind of operation is performed by the returned object.

When the user got a pointer to `AbsCreamProxy` she/he has to invoke the `setCredential(...)` method in order to set the authentication credentials, optionally invoke the `setConnectionTimeout(...)` in order to set a maximum socket timeout for the connections with the CREAM service, and then must invoke `execute(...)` to actually communicate with the service (send SOAP request, receive and parse SOAP response).

It is clear that the developer has to know nothing about subclasses: she/he just uses factory's make methods and the two methods `setCredential(...)` and `execute(...)`; then the subclasses are not documented at all. The API documentation only describes the `AbsCreamProxy`, the factory and the data structures to pass as argument of the 'make' methods of the factory.

At the end, the user must delete the pointer to `AbsCreamProxy` in order to free the heap-memory that the factory allocated.

Step by step Job Submission explanation

Proxy Delegation

In order to register a job, the user must previously delegate a proxy into the remote CREAM CE service. The procedure is a matter of a few steps:

- define an arbitrary string containing a delegation identifier (it will be used later as job parameter for registration of one or more jobs); **please consider the importance of re-use a single delegation identifier for multiple job submissions; in fact the delegation process takes some time and can be a large overhead for the submission process**
- create an instance of a subclass of `AbsCreamProxy` with the invocation of the static method `CreamProxyFactory::make_CreamProxyDelegate(...)`
- invoke the `setCredential(...)` method on the instance created above
- invoke the `execute(...)` method on the instance created above
- delete the `AbsCreamProxy` instance

Download the example code in attach (`tryDelegate.cpp`) and build it with the instructions described above.

Job Registration

The steps for the `JobRegister` operation are:

- Prepare one or more strings containing the JDL descriptions of one or more jobs to submit
- Obtain a delegation identifier of a pre-delegated proxy (see Proxy Delegation)
- Prepare one `JobDescriptionWrapper` object for each job to submit to the CREAM CE
- Put all the `JobDescriptionWrapper` objects in a list (see below)
- Create an instance of a subclass of `AbsCreamProxy` with the invocation of the static method `CreamProxyFactory::make_CreamProxyRegister(...)`
- Invoke the `setCredential(...)` method on the instance created above
- Invoke the `execute(...)` method on the instance created above
- Process the output
- Delete the `AbsCreamProxy` instance

The input and output arguments of the `CreamProxyFactory::make_CreamProxyRegister(...)` are a bit more complicated than in the case of the Proxy Delegation. As described in the API reference, the input and output arguments are pointers to:

- `AbsCreamProxy::RegisterArrayRequest`
- `AbsCreamProxy::RegisterArrayResult`

They are typedef for (respectively):

- `std::list< JobDescriptionWrapper* >`
- `std::map< std::string, boost::tuple< JobIdWrapper , std::string> >`

The first one is simply a C++ STL list where the user has to insert pointers to `JobDescriptionWrapper` objects; the second one is a complex structure based on the `boost::tuple` library. Each `JobDescriptionWrapper` object is built with a `JobDescription` identifier (an arbitrary string chosen by user) and other parameters that can be seen in the example source code `tryRegister.cpp`. After the invocation of the `execute(...)` method, the second argument (passed to the `CreamProxyFactory::make_CreamProxyRegister(...)` function as output parameter) will be filled as follows:

- The key of the map is the `JobDescription` identifier (as defined by the user)
- The value corresponding to the key is a `boost::tuple`

this tuple groups three elements:

- The result of the operation, `JobIdWrapper::RESULT`
- The Job (represented by a `JobIdWrapper` object, use the `getCreamJobID` method to obtain the Cream Job ID that you will need to start later)
- A string representing an error message (non empty if the first element is different than `JobIdWrapper::OK`)

As shown in the example source code `tryRegister.cpp`, a mandatory argument is `autostart`. In the example it is set to `false`; this means that the job is **ONLY REGISTERED** and **NOT STARTED**. Below it is explained the usefulness of `autostart` set to `false`. If she/he needs to start the job immediately after registration she/he can set to `true` the `autostart` parameter. An array of properties (implemented as a `std::map < string, string >`, i.e. couples key -> value) is embedded in the `JobIdWrapper` object returned by `JobRegister` operation. At the moment the only relevant property returned by CREAM is the remote path in the CE the user can send its `InputSandbox` to, and name `'CREAMInputSandboxURI'` (see again the example source code).

- Example code for **single job submission** is here
- Example code for **multiple job registration** with a single remote call is here

Job start

In the previous example, jobs are simply registered in the CREAM CE (`autostart` is set to `false`). This is useful if the user needs to do something between job registration and job start (e.g.: sending an input sandbox in the remote path specified by the CREAM CE). To explicitly start a job the user must:

- Obtain the Cream Job IDs of the jobs to start (as result of `JobRegister` operation)
- Build the `JobIdWrapper` object representing the jobs to start (one for each Cream Job ID to start)
- Define a time range (specified by two variables `fromDate` and `toDate`) which will allow her/him to only start the jobs that were registered in this time range [`fromDate`, `toDate`]
- Build a `JobFilterWrapper` with all the `JobIdWrapper` objects representing the jobs to start, and the time range
- Create the proper subclass of `AbsCreamProxy` by invoking the static factory method `CreamProxyFactory::make_CreamProxyStart(...)`
- Invoke, as usual, the `setCredential(...)` and `execute(...)` methods on the previously created `AbsCreamProxy`'s instance

To start all her/his jobs, the user has to use an empty vector of `JobIdWrapper` objects as argument of the `JobFilterWrapper`'s constructor.

Canceling, Suspending, Resuming and Purging jobs

The code for these four operations is basically the same. The user has to

- prepare a `JobFilterWrapper` object (the same for the `JobStart` operation) and pass a pointer to it as first argument of the static method
`CreamProxyFactory::make_CreamProxyCancel(...)/CreamProxyFactory::make_CreamProxySuspend(...)/Crea`

The second argument of the factory methods is a pointer to a `ResultWrapper` object that will be filled-in with the results sent back by CREAM.

Like in the `JobStart` case, to cancel/suspend/resume/purge all her/his jobs, the user has to use an empty vector of `JobIdWrapper` objects as argument of the `JobFilterWrapper`'s constructor.

The source code example is here. Please note that it is only for `JobCancel` operation; adapting it for the other three operations is straightforward.

Listing jobs in a CREAM CE

The code for listing all the jobs submitted to a CREAM CE is very simple; a few simple parameters are needed for the factory `CreamProxyFactory_makeCreamProxyList(...)`. A look at the source code should be enough to understand what the user has to do.

Getting Information about jobs submitted to a CREAM CE

A user can invoke two operations to get information on one or more jobs submitted to a CREAM CE: `JobInfo` and `JobStatus`. The latter is quicker but provides less information than the former. In both cases the user must prepare a `JobFilterWrapper` object that collects all the identifier strings of the jobs to query and the conditions the jobs must satisfy in order to be included in the result. For example a user might need the query status of all jobs submitted between 8:00am and 15:00am of a particular day (if they were not purged out yet). Or she/he might need information on all the jobs that are in the "RUNNING" OR "DONE-OK" status... and so on. Please see the documentation of `JobFilterWrapper` to see what filters a user can define.

Fast info query (JobStatus)

With the invocation of the `JobStatus` remote operation CREAM will return a minimal set of information about the jobs: the current status of the job, the timestamp of the last status change, the exit code of the job and the failure reason (if the job finished or aborted). To query the states of some jobs, the user must prepare a `JobFilterWrapper` object and fill it with `JobIdWrapper` objects (one for each job to query) and with other constraints to select a particular set of jobs (see the `JobFilterWrapper` documentation).

As usual a user can query for the states of all her/his jobs; this can be achieved by using an empty vector of `JobIdWrapper` objects as argument of the `JobFilterWrapper`'s constructor. See the `CreamProxyFactory::make_CreamProxyStatus(...)` and `JobStatusWrapper` documentation for more details about the structure of the information returned by CREAM.

As usual the example code is the best way to explain how it works.

Slow info query (JobInfo)

To get complete information about jobs the user must invoke the remote call `JobInfo`. The procedure is much similar to the fast call `JobStatus`: the user has to prepare a `JobFilterWrapper` object to select jobs and filters; then the result will be put in a `JobInfoWrapper` structure instead of a `JobStatusWrapper`. The data structure containing the output of the `JobInfo` operation is very similar to that one for the `JobStatus` operation. See the

CreamProxyFactory::make_CreamProxyInfo(...) and JobInfoWrapper documentation for more details about the structure of the information returned by CREAM.

Example source code.

Enabling and disabling the Job Submission Service

Depending on the role of the user, she/he can enable/disable the job submission on a remote CREAM CE. The code is so simple that, as usual, the example speaks for itself. Relevant API documentation is clearly the factory method CramProxyFactory::make_CreamProxyAcceptNewJobSubmissions(...).

Renewing a Delegated Proxy

To register a job to a CREAM CE a client must first delegate a proxy to the CE and save an identifier string associated with this delegation; this identifier will be used to register jobs. But the proxy are not valid forever. At some point the delegated proxy must be renewed. The procedure is as simple as in the delegation case. The user has just to put the delegation identifier string into the factory method that creates the proper AbsCreamProxy subclass and invoke, as usual, setCredential(...) and execute(...). Taking a look at the example source code is the fastest way to understand this very simple procedure. Relevant API documentation is clearly the factory method CramProxyFactory::make_CreamProxy_ProxyRenew(...).

Querying job status events

QueryEvent is a convenient CREAM's operation that returns a selected range of particular events. At the moment in the CREAM framework an Event is a job's status change; in future an evolved version of CREAM could put different kind of information inside an event.

A user that invokes a QueryEvent on a CREAM CE, receives all her/his jobs's status changes. The query has three filters:

- time range (from, to)
- ID range (from ,to)
- job's states

Time range has a quite intuitive meaning, the ID a few less. Each event generated in a CE for a certain user, has an incremental ID (64bit unsigned integer). A user can be interested to a certain subset of all her/his job events, identified by a particular ID range. This kind of filtering is particularly useful to the ICE component that memorizes the last event's ID received in the last call for each couple (user_DN, CE_URL), resulting in a quite small information exchange with the CE at each remote call.

Finally, the filter related to job's states simply select those events that carry a job status that is present in a list specified by the user.

As usual, in order to perform a QueryEvent the user must obtain a pointer to an AbsCreamProxy object by mean of CreamProxyFactory::make_CreamProxy_QueryEvent(...), invoke setCredential(...) and execute(...) methods on it, and delete it. The invocation of execute(...) will fill up a list of EventWrapper pointers (that is an argument of the CreamProxyFactory::make_CreamProxy_QueryEvent function).

Please see the example source code to understand how to use the above stuff.

Getting Service Information from a deployed CREAM CE running service

This remote operation is very simple; the factory CreamProxyFactory::make_CreamProxyServiceInfo(...) just

needs a simple parameter that is a pointer to a ServiceInfoWrapper object and a verbosity; ServiceInfoWrapper exposes methods to obtain information on the CREAM service which are be self-explanatory.

Example source code.

Contact info

For any problem or doubt, to report bugs or any kind of error in the text or in the source code example, please contact me at

alvise <DOT> dorigo <AT> pd <DOT> infn <DOT> it

-- AlviseDorigo - 2011-06-14

This topic: CREAM > ClientApiCppDoc

Topic revision: r11 - 2011-07-14 - AlviseDorigo



Copyright © 2008-2024 by the contributing authors. All material on this collaboration platform is the property of the contributing authors.

Ideas, requests, problems regarding TWiki? Send feedback