Table of Contents

CREAM client API for Python

Maciej Sitarz created a CREAM Client API for Python language.

The `pyCream` project is available on github (https://github.com/maciex/pyCream), so anyone can download, use it or participate in development. It's GPLv3 licensed.

If you have any problems or questions regarding building/installation/usage of the pyCream module, you can contact Maciej via github or via e-mail (macieksitarz AT wp DOT pl).

CREAM Client development tutorial

Preliminaries: build machine configuration

Before developing a CREAM's Py CLI you must configure properly a SL5_64bit machine. The prerequisites are the same as for Cpp Client. Additional requrements are:

- python (tested with version 2.7)
- boost (tested with version 1.45)

Cpp API vs. Python API -- differences

STL collections

All of the STL objects passed to the API calls need to be of one of the wrapped types. Here's a table of those corresponding types.

Cpp API	Python API	Treat as
std::pair< long, long >	stdPairLongLong	tuple?
std::pair< std::string, long >	stdPairStringLong	tuple?
std::pair< std::string, std::string >	stdPairStringString	tuple?
std::map< std::string, std::string >	stdMapStringString	map
std::vector< std::string >	stdVectorString	list
std::allocator< char >	stdAllocatorChar	?
boost::tuple< JobIdWrapper::RESULT, JobIdWrapper, std::string >	TupleJobIdWrapper	tuple
boost::tuple< JobStatusWrapper::RESULT, JobStatusWrapper, std::string >	TupleJobStatusWrapper	tuple
boost::tuple< JobInfoWrapper::RESULT, JobInfoWrapper, std::string >	TupleJobInfoWrapper	tuple
RegisterArrayResult is std::map< std::string, boost::tuple<JobIdWrapper::RESULT, JobIdWrapper, std::string > >	RegisterArrayResult	map
StatusArrayResult is std::map< std::string, boost::tuple<JobStatusWrapper::RESULT, JobStatusWrapper, std::string > >	StatusArrayResult	map
InfoArrayResult is std::map< std::string, boost::tuple<JobInfoWrapper::RESULT, JobInfoWrapper, std::string > >	InfoArrayResult	map
std::vector< JobIdWrapper >	stdVectorJobIdWrapper	list
std::vector< JobPropertyWrapper >	stdVectorJobPropertyWrapper	list
std::vector< JobStatusWrapper >	stdVectorJobStatusWrapper	list
RegisterArrayRequest is std::list< JobDescriptionWrapper* >	RegisterArrayRequest	list

Freeing memory

When using the Cpp API the user had to delelte the pointer in order to free the heap-memory that the AbsCreamProxy factory allocated. In Python one can use 'del' statement, probably it's not needed because the memory should be cleaned by Python's garbage collector (need to check that somehow!).

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 Python.

Quick overview of the API

Currently, the Python API implements the following operations:

- JobRegister

- JobStart
- JobCancel
- JobSuspend
- JobResume
- JobPurge
- JobList
- JobInfo
- JobStatus
- ProxyDelegation
- ProxyRenew
- ServiceInfo
- Enable/Disable JobSubmission (No Python bindings created yet)
- Query JobSubmission enable status (No Python bindings created yet)

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 returns an object of one of the classes subclassing AbsCreamProxy . The 'make' methods take different parameters depending on the kind of operation is performed by the returned object.

When the user got an object of class subclassing AbsCreamProxy class 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, when using the Cpp API the user had to 'del' the object of AbsCreamProxy class 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.py) 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

In Cpp API they are typedef for (respectively):

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

Which in Python is represented by:

- list of JobDescriptionWrapper objects
- map of string and tupel, the tuple consists of JobIdWrapper object and string

The first one is simply a Python list where the user has to insert 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 been seen in the example source code tryRegister.py . 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 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.py , 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 map of string pairs, 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. One needs to note that as Cpp API takes JobIdWrapper object in a STL vector, the Python API uses special class 'stdVectorJobIdWrapper', which is equivalent to std::vector< JobIdWrapper >.
- 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 list(stdVectorJobIdWrapper) of JobIdWrapper objects as argument of the

JobFilterWrapper's costructor.

Cancelling, 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(...)/Cream

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 list(stdVectorJobIdWrapper) of JobIdWrapper objects as argument of the JobFilterWrapper's costructor.

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 costructor. 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 ServiceInfoWrapper object and a verbosity; ServiceInfoWrapper exposes methods to obtain information on the CREAM service which are be self-explanatory.

Example source code .

-- MassimoSgaravatto - 2011-07-30

This topic: CREAM > ClientApiPython
Topic revision: r3 - 2011-08-31 - MaciejSitarz