SOAP Services with Clarens
Guide for Developers and Administrators

Conrad Steenberg, Joseph C. Jacob, Roy Williams

Jan 20, 2005
Version: 1.1

1 Introduction

The Clarens application server enables secure, asynchronous SOAP services to run on a Grid cluster such as one of those of the TeraGrid. There is a **Client**, who wants to use the service and understands the application domain enough to form a reasonable service request; a **Developer**, who is a power-user of the TeraGrid, who understands both Clarens and the application domain, and creates and deploys a service on a TeraGrid head node; and there is a **Root** system administrator, who controls the Clarens installation and the cluster on which it runs.

The purpose of this document is to provide all of the information a service developer needs to know in order to deploy a Clarens service, with information also provided for the system administrator of the Clarens installation. First we discuss how each of the three roles see the service.

1.1 From the Client point of view

The client is responsible for creating a request for service, together with a X.509 certificate (or other credentials; see Graduated Security description below), wrapping both in a secure SOAP envelope, and delivering this over the HTTPS protocol to the Clarens server that is running on tg-www or tg-login. Clarens will provide its own host-key certificate to assure the client that the right server is being used.

The client may be one of several types. The only requirement is that a properly-formatted SOAP message be sent. A web server could be set up with a connection to a PURSE/MyProxy installation, so that a human uses the forms on the web server, which in turn generates the SOAP request/certificate, and acts as the client in its interactions with Clarens. The client could also be a laptop machine of a user, who has a certificate stored onboard and is using the open-source Clarens client library. The client could also be another machine that is part of a workflow, executing multiple jobs on multiple machines, some grid-based (Globus), some not.

1.2 From the Developer point of view

Create the service in your home directory, beginning with some of the Python sample code, then ask Root to make a symbolic link to that directory with a name that you choose. You will need to restart the Clarens server in order to make it see your code, or any changes that you make. Therefore there is a special way for Developers to do this. A system administrator with Root privileges will make you a member of the Clarens

Service Developer Group, a Unix group that all developers join, and members of that group have permission to restart the Clarens server with `sudo clarens-restart`.

When your code runs, you can see the certificate of the requestor, which will be used to decide (a) the account under which the service is to run, and (b) if the size of the request is too much for that type of certificate. If the certificate DN has a match in the gridmap file of the host cluster, then the service will run as this account. You can find strings in the DN to decide if you want to run as a community account or some other type of account. Note that only the system administrator with Root privileges can add or remove entries in the gridmap file.

When Clarens initializes the service, a "sessionID" is created (a 32-digit hex number), together with a directory of the same name, created in a temporary area. This is the so-called "sandbox", owned by the user account that is running the service, and with the umask of that user for files created in there.

The service can see the request as a set of named arguments (dictionary of keyword-value pairs). Values should be extracted from these and parsed and checked for buffer overflow exploits in the usual way. The service may run Unix commands through the usual `os.system()` and `os.popen()` system calls, and can, for example launch jobs into the queues of the cluster. These all run as the account decided above. The return from this initialization service is the sessionID and a URL that points into the sandbox area so that the client can examine its contents.

Subsequent service requests can quote the sessionID in order to monitor the contents of the sandbox. Each file in the sandbox has a URL, and can therefore be retrieved.

1.3 From the Root point of view

When a Developer wants a service to be deployed, they will need to have a symbolic link from the Clarens main directory, and to be made a member of the Developers group so that they can restart Clarens during the edit/test cycle of development. At this stage, Developers should state their intentions: what services will be offered, what types of certificates will be accepted, what parameter in the service controls the resources that will be used, and what the policies are with respect to that parameter. Developers should also expose for auditing the code that reads the request arguments and parses them.

All of the contents of the Clarens system directory should be writable only by Root. In there is a configuration file (`clarens_config.py`), which locates the server certificate, the gridmap file for the cluster, and other protected configuration.

1.4 Graduated security

Clarens services are authenticated with graduated security, which means that the appropriate amount of authentication is required for the size of the service request, no more and no less. For instance, very small requests may be done anonymously, medium-sized requests require a "weak certificate" issued by an organization other than the grid computing organization. An example of a weak certificate is one issued by the National

Virtual Observatory, or a Hot Grid certificate. Large service requests require a strong certificate issued by an official grid computing organization. Examples of strong certificates are X.509 certificates issued by the TeraGrid or the DOE. A service request may be processed under a different user name, depending on the authentication level of the requestor (anonymous, weak certificate, or strong certificate).

2 Installation and Administration

2.1 Clarens web services framework

SOAP services are deployed under a Clarens server running on the service host. Clarens is an open source web services framework that supports a number of authentication mechanisms, including X.509, grid proxy, and Clarens proxy certificates. A client for a Clarens service may be a web browser or a stand-alone program in C, C++, Java, or Python. More information on Clarens may be found at http://clarens.sourceforge.net.

Clarens is implemented as a mod_python extension to the Apache web server. It provides access control by proxy (certificate) and virtual organization management and an API for service developers.

If a user of a service has an X.509 certificate, the certificate's distinguished name (DN) is mapped to a local user on the service host. Clarens then performs any required job execution or file I/O as the local user.

We assume that Clarens has been installed and is running. It means that a port is open for the HTTPS protocol – for example, the service host at Caltech is `tg-www.cacr.caltech.edu`, with the Clarens service container running on port 8443. Client codes can connect to the Clarens service container at:

```
https://tg-www.cacr.caltech.edu:8443/clarens/
```

There is also an install directory for Clarens, and we abbreviate this with the variable `$CLARENS_ROOTCLARENS_ROOT`.

2.2 Important file and directory locations

Some important files and directories for Clarens services include the following:

? The variable `CLARENS_ROOT` should be set to the install directory of Clarens. At CACR `$CLARENS_ROOT` is set to `~clarens/openpkg/share/apache2/clarens`

? `$CLARENS_ROOT/.clarens_file_root/shell` : The sandbox directory. Each service request is assigned a unique identifier string and a working "sandbox" subdirectory here. The subdirectory is <ID>/<identifier>, where <ID> is the first 2 characters in the identifier string and <identifier> is the full identifier string.

? `$CLARENS_ROOT/shell/grid-mapfile` : The grid map file that maps certificate distinguished names to usernames on the service host.

? `$CLARENS_ROOT/clarens_config_local.py` : A dictionary that defines the local environment setup, common to all services. Only the system administrator with Root privileges can change this dictionary.

? `~clarens/openpkg/lib/python/site-packages/gridsession.py` : a python module implementing an API for useful functions that all services can use.

3 Writing and deploying a TeraGrid SOAP service

3.1 Service installation

Clarens SOAP services are written in Python and installed in `$CLARENS_ROOT/<ServiceName>`, where `<ServiceName>` is the name of the service. Usually `<ServiceName>` is a symbolic link that points to an external service source directory, so that any system user can maintain ownership of the source code. Two files are required in this subdirectory, the service code itself in a file named `__init__.py`, and an access control file named `.clarens_access`. A sample `.clarens_access` file is as follows:

```
ORDER_ALLOW_DENY=0
ORDER_DENY_ALLOW=1
access=[("",[ORDER_ALLOW_DENY,   # Order
             ["/"],                                    # Allow everybody
who can log in
           [],                                        # Allow group
             [],                                        # Deny indiv
default=all
             []  ,                                      # Deny
default=all
             [None, None, None]]), # modtime, start_time, end_time
            ]
```

3.2 The service code module, __init__.py

As mentioned above, the service code module should be written in Python and reside in `$CLARENS_ROOT/<ServiceName>/__init__.py`, where `<ServiceName>` is the name of the service. This file should import the `gridsession.py` module and should contain a function that implements the service and a dictionary that maps this *function name* to a *service name*, which is the name of this service as exposed to the client code. The function must take 3 arguments, all of which are automatically assigned by Clarens when it accepts the client-side call and makes the server-side call to this function. The arguments are:

? `req:` A Clarens request object

? `method_name:` The name of the function being called

? `args:` The user supplied arguments

The service implementation must define the PBS time required, which is used in the graduated security policy (see the API description for `run_job()` below). Finally, the service implementation must launch the job with a call to `run_job()`.

3.3 Service developers API

3.3.1 gridsession.py

The `gridsession.py` module provides an API that service developers may use to build their services. The useful methods and fields provided are:

`user_name = **getSystemUser**(req, pbs_time) :`
Given a Clarens request object and the processing time in minutes needed to handle the request, returns the system user that will run the job.

`[session_id, sandbox] = **getSessionInfo** (req, method_name, pbs_time) :`
Given a Clarens request object, a method name, and the processing time in minutes needed to handle the request, returns the session identification string and sandbox directory that have been assigned to the job.

`sandbox = **getSandbox** (session_id) :`
Given a session identification string, returns the sandbox directory for the request.

`output = **getStatus**(session_id):`
Given a session identification string, returns a string containing status information about the request, including the PBS queue status and the sandbox directory listing.

`status = **runCmd**(req, method_name, call_args, cmd) :`
This runs a command in a unix shell on the service host. The first 2 arguments (`req` and `method_name`) are simply the required arguments to the service function, as described above. The third argument (`call_args`) is a list of 1 element, the processing time in minutes needed to handle the request. The fourth argument (`cmd`) is a string containing the command to run with arguments (usually an external script or program that handles the service request).

`status = **runJob**(req, method_name, call_args, cmd):`
This submits the job to the PBS queue for execution. The first 2 arguments (`req` and `method_name`) are simply the required arguments to the service function, as described above. The third argument (`call_args`) is a list of 1 or 2 elements:

- the processing time in minutes needed to handle the request (required)
- the session identification string (optional; if omitted, a session identification string will be assigned by Clarens)

The fourth argument (`cmd`) is a string containing the command to run with arguments (usually an external script or program that handles the service request).

`status = **jobMonitor**(req, method_name, args):`
A generic service implementation for job monitoring; writes a status message showing queue status and sandbox contents back to the client. The first 2 arguments (`req` and `method_name`) are simply the required arguments to the service function, as described above. The third argument (`args`) is a list with a single element, the session ID for the job to be monitored.

3.3.2 clarens_config_local.py

The clarens_config_local.py file contains a dictionary assigning values to some configuration keys that can be used in the service code. The configuration keys that are defined are the following:

- `grist_strong_cert_max_time` : maximum processing time with a strong certificate
- `grist_weak_cert_max_time` : maximum processing time with a weak certificate
- `grist_anon_max_time` : maximum processing time with anonymous access
- `grist_weak_user` : The user to run the job as for weak access
- `grist_anon_user` : The user to run the job as for anonymous access
- `grist_qsub_cmd` : The unix command to run to submit a job to the queue
- `grist_qstat_cmd` : The unix command to run to check queue status

In Python, these configuration keys are accessed with `config["key"]`, where `key` is the configuration key (e.g., `config["grist_anon_max_time"]`).

3.4 Restarting the Clarens server

Any change to a service requires restarting the Clarens server in order for it to take effect. To restart the Clarens server, ask the system administrators to add you to the Developers group (called tgcl at Caltech). Then you may add `/usr/local/adm/bin` to your path and run:

```
sudo clarens-restart
```

You should see something like the following:

```
% sudo clarens-restart
OpenPKG: stop: apache2.
OpenPKG: start: apache2.
```

3.5 Sample service: sleepyAdd

Below is the source listing for a sample service definition called `sleepy_add`, which launches a job in the PBS batch queue to sleep for a user specified number of seconds and then add two user specified numbers together. The dictionary at the bottom of the source listing maps the service function name of `sleepy_add` to an exposed service name of `sleepyAdd`, which the client code would call.

```
# Import Grist service API functions from gridsession.py
from gridsession import *
#
# Function to implement the sleepyAdd service.  This is called by
Clarens
# using the 3 arguments given.
#
def sleepy_add(req, method_name, args):
    #
    # Parse user supplied arguments
    #
```

```
    tme = args[0]    # sleep time in seconds
    n1 = args[1]        # first number
    n2 = args[2]        # second number
    #
    # Graduated security implementation
    #
    pbs_tme = tme/60 + 1
call_args = [pbs_tme]
    #
    # Specify the command to be executed.  This is a complicated way to
    # sleep and add two numbers on a single line.
    #
    cmd = "date; sleep %s;  date; echo \"%s %s\" | awk \'{print
\"sleepyadd \", $1, \" + \", $2, \" = \", $1+$2}\'" % (tme, n1, n2)
    #
    # Ask Clarens to submit the job to the PBS batch queue.
    #
    retval = run_job(req, method_name, call_args, cmd)
    return retval
#
# Method dictionary mapping function names to exposed client
# service names.
methods_list={‘sleepyAdd’ : sleepy_add}
```

4 Client-side call to a Clarens web service

4.1 Required client software

The client software needed for Clarens web services consists of a recent Python installation and three additional files:

- The Clarens client library, `ClarensDpe.py`
- The Clarens proxy certificate generator, `clarens-proxy-init`
- The service-specific Python client.

4.2 Service specific Python client sample: sleepyAdd

The service-specific Python client imports the Clarens client library and uses it to connect to the Clarens server, as follows:

```
import ClarensDpe as Clarens
svrObj =Clarens.client('https://tg-www.cacr.caltech.edu:8443/clarens/',
debug=0)
```

The `svrObj` object can then be used to make call to the service, with a unique service identification string returned, as follows:

```
# sleep for 20 secs, then add 3 and 5
sessionID = svrObj.sleepyServices.sleepyAdd(20, 3, 5)
```

The sessionID string can then be used to monitor that status of the job, as follows:

```
statusMsg = SvrObj.sleepyServices.monitor(sessionID)
print statusMsg
```

4.3 Certificates and proxies

If you have a strong certificate, make sure it is in your `~/.globus` directory as two files, `usercert.pem` and `userkey.pem`. If you have a certificate as `usercert.p12` only, you can split it into these two files be running the following commands:

```
openssl pkcs12 -clcerts -in certkey.p12 -nokeys -out
~/.globus/usercert.pem

openssl pkcs12 -nocerts -in certkey.p12 -out ~/.globus/userkey.pem
```

Run the `clarens-proxy-init` program once at the start of your session to get a short term Clarens proxy certificate. You will be asked for your certificate pass phrase once and will not have to enter it again for the session. If you don't do this, you will be asked to enter your pass phrase each time the client code connects to Clarens.