Web Services Framework

Conrad Steenberg
conrad@hep.caltech.edu

July 11, 2005

Abstract

The Clarens Web Services Framework provides a specification for writing flexible Grid-enabled web services using standard protocols and tools. A reference implementation using Apache and mod_python is actively developed and maintained, while a Java/Tomcat version is being developed. A standalone Python server implementation is also available. Clients for C++, Java, Python, and JavaScript have been implemented. Clients have been integrated into the ROOT analysis framework, the Iguana CMS visualization package, and Java Analysis Studio on the IPAQ PDA.

Acknowledgments

This work supported by Department of Energy contract DE-FC02-01ER25459, as part of the Particle Physics DataGrid project, and under National Science Foundation Grant No. 0218937.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the National Science Foundation.

Contents

1 Introduction **7**

1.1 Definition 7

1.2 Implementation 7

1.3 Single sign-on 8

2 Installation **9**

2.1 Installation using `clump` 9

2.2 Manual OpenPKG installation 10

2.3 Post-Installation Configuration 11

2.3.1 Apache Web Server 11

2.3.2 Server Module Configuration 11

2.3.3 Virtual Organization 13

2.3.4 Method Access Control 14

2.3.5 File Access Control 16

3 Client Usage **19**

3.1 Command-Line Utilities 19

3.1.1 `clarens-proxy-init` 19

3.1.2 `clarens-ping` 20

3.1.3 `clarens-package-hostcert` 21

3.1.4 `clarens-package-hostkey` 22

3.1.5 `clarens-package-cacert` 22

3.1.6 `clarens-package-cadir` 23

3.1.7 `clarens-package-root` 23

3.2 Kerberos Certificate Authority 24

3.3 Python 24

3.3.1 Simple example 24
3.3.2 Procedure calls 25
3.3.3 Session termination 25
3.3.4 Complete example 25
3.3.5 Binary return data 26
3.3.6 Debugging 26
3.3.7 Using a different certificate file 26
3.4 ROOT 27
3.4.1 TCWebfile 27
3.4.2 A simple example 27
3.4.3 Different certificate and key files 28
3.4.4 Caching file reads 28
3.4.5 Multiple files on a single site 29
3.4.6 Multiple sites, single login 29
3.4.7 TCSystemDirectory 30

4 Writing Browser Interfaces 31
4.1 Files and paths 31
4.1.1 Accessing files on the server in different contexts 31
4.1.2 Browser interface path conventions 32
4.1.3 Registry files 32
4.2 Creating the interface 33
4.2.1 HTML file 33
4.2.2 Javascript file 34

5 Writing new Clarens modules 37
5.1 Introduction 37
5.2 First Example: the `echo` module 37
5.2.1 Starting out 37
5.2.2 Defining a new method 38
5.2.3 Exposing the method 38
5.2.4 The full example 39
5.3 Debugging 40
5.3.1 Python debugger 40

5.3.2 Automatic tracebacks . . . 40
5.4 Handling exceptions with build_fault . . . 42
5.5 Service initialization . . . 42
5.5.1 Example . . . 43
5.5.2 Per server initialization . . . 43
5.6 Persistent data storage using tdb . . . 44
5.6.1 Opening database handles . . . 44
5.6.2 Standard databases . . . 45
5.6.3 Printf-style debugging . . . 46

6 Service packaging and configuration 47
6.1 Service installation . . . 47
6.1.1 Service code . . . 47
6.1.2 Configuration files . . . 47
6.1.3 Minimal example . . . 47
6.2 Service configuration . . . 48
6.2.1 Configuration file format . . . 48
6.2.2 Configuration file example . . . 49
6.2.3 Command-line configuration . . . 49

7 Clarens authentication/login protocol 51
7.1 Introduction . . . 51
7.2 Two steps . . . 51
7.3 SSL based authentication . . . 55
7.3.1 Proxy authentication over SSL . . . 56
7.4 GET requests . . . 56
7.5 Other transport protocols . . . 56
7.6 Discussion . . . 56

Chapter 1

Introduction

1.1 Definition

Clarens is a framework for writing grid-enabled web service applications – both clients and servers. The terms ”web service” and ”grid” might need some clarification:

- **Web Services** are generally understood to beremote procedure calls encoded in XML and transported using the `http` protocol. This name is rather unfortunate, since the transport protocol is in fact an implementation choice. Also, web services have nothing to do with web pages as rendered by web browsers!
- The term **Grid** refers to a software and hardware infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities, according to Foster & Kesselman (1999) [4].

Though vague, the latter definition provides some clues as to the intent of grid computing: namely the ubiquitous provision and access to computational facilities. The dependability of these resources as well as the cost of access, and the power of the computational machinery that is made available are in fact determined post-facto by physical, economical and implementation factors.

Web services deployed over a wide area network such as the internet provides the ideal vehicle for implementing and deploying computational grids because of the ubiquity of access to the internet and its well-standardized infrastructure.

From the above, grid computing is part of the continuing trend of commoditization of communications protocols at a higher functional level than the internet protocol (IP) hypertext transfer protocol (HTTP) or information prepresentation in e.g. extensible markup language (XML).

1.2 Implementation

The original Clarens server described in this document implements web services using a combination of the Apache [1] web server and the Python [10] language. The choice of programming language was prompted by its wide use in the CMS experiment at CERN. Since the goal of grid computing is to provide ubiquitous access, it should be strongly emphasized that these are merely

implementation details, and there is in fact an effort to produce an equivalent server implementation using the Java [5] language.

1.3 Single sign-on

The cornerstone of ubiquitous resource access is a universal identification space. Users and providers of Clarens services are mutually identified through the use of cryptographically signed certificates using the X509 directory standard embedding so-called public keys. Through the wonders of public/private key encryption [14], these certificates are used both for identification and to establish secure communication channels.

Certificates are signed by Certification Authorities (CAs) that vouch for the indentity of a certificate through some physical verification mechanism. Security of the system depends of the accuracy of this certification, the secrecy of the private key used in communications, and dealing with breakdowns in this trust relationship by quick propagation of certificate revocation information.

In the context of web services, Clarens servers support the widely used Secure Sockets Layer (SSL) standard for establishing secure communications, but additionally uses either so-called *Cookies* or HTTP Basic authentication to exchange credentials securely. For e complete description of this exchange mechanism, see section 7. Work is also underway to add support for the Globus Security Infrastructure (GSI) version of GSSAPI by the Globus project [3].

Chapter 2

Installation

Clarens is loosely tied to the Unix/Linux platform, but has only been tested on the Linux on x86 and Solaris on SPARC implementations. This still leaves a wide variety of OS versions to contend with. Instead of relying on versions of the underlying components supplied by the operating system, Clarens is routinely built and distribted in binary form as a set of RPM [12] packages built on top of OpenPKG [7].

The home of the Clarens RPM packages is at http://hepgrid1.caltech.edu/clarens/. In the following paragraphs the installation procedure is described using manual (interactive) and automated methods.

2.1 Installation using `clump`

The Yellow dog Updater Modified (YUM) [15] is a Python-based automated installer updater that is an easy-to-use automated installation/update tool that is widely supported with numerous package repositories for operating system updates and third-party packages for Linux distributions. This tool has been adapted for use in an OpenPKG environment, and renamed to `clump` to avoid confusion with the system version of `YUM`. This is the preferred tool to install and keep a Clarens server updated.

1. To install as the superuser (root) do:
   ```
   wget -q -O - http://hepgrid1.caltech.edu/clarens/setup_clump.sh |sh
   export opkg_root=/opt/openpkg
   ```

2. To install as an ordinary user do:
   ```
   wget -q -O - http://hepgrid1.caltech.edu/clarens/setup_user.sh
   sh setup_user.sh
   ```
   You will be prompted for in installation path. The default is `$HOME/openpkg`. Then:
   ```
   export opkg_root=/my/directory
   ```

3. If your host already has a certificate/key pair, these can be copied to
   `$opkg_root/etc/grid-security`
   Make sure that both files are readable by user `nobody` in the case of a superuser installation.

4. The server can be controlled with
   `$opkg_root/etc/rc apache2 start|stop|restart`

5. To make sure that the server works, point your web browser at
   `https://my.server.name:8443/clarens/clarens_index.html`
   using the correct servername or IP address of course.

6. The base address of the server is
   `http://my.server.name:8080/clarens/` or
   `https://my.server.name:8443/clarens/` for an encrypted connection.

Startup problems are reported on the terminal when the server is started, or recorded in the log file at `$opkg_root/apache2/log/error_log`.

2.2 Manual OpenPKG installation

Since OpenPKG contains its own RPM database it must be bootstrapped in some way without using the (possibly non-existent or incompatible) system version of RPM. This is done by distributing the base package as a `uuencode`d shell archive.

The information below describes the procedure carried out by the setup shell scripts for `clump`, and is provided for completeness. If you are not comfortable with typing in commands in a terminal, please use one the above automated methods instead.

1. Point your browser at
   `http://hepgrid1.caltech.edu/clarens/openpkg-binary-redhat-7.3/`.
   Fetch the OpenPKG base archive, e.g.
   `openpkg-20030813-20030813.ix86-linux2.4-oo.sh`

2. Install the base package with the following command as root:
   `sh openpkg-20030813-20030813.ix86-linux2.4-oo.sh`

3. Initialize the OpenPKG runtime environment:
   `eval $(/opt/openpkg/etc/rc --eval all env)`
   Note that this will only work in a `bash`, Bourne, or Korn shell environment, not a C-shell.

   It might be useful to add the following to your login scripts to avoid having to type in the above every time the environment is used:

   Shell
   ```
   export opkg_root=/opt/openpkg
   alias opkg_init='eval \$(\$opkg_root/etc/rc --eval all env)'
   ```

4. Download and install all the RPMs from the above location with
   `rpm -i *.rpm`

5. Start the server with
   `/opt/openpkg/etc/rc apache2 start`

2.3 Post-Installation Configuration

As with any complex server application, Clarens has many parameters that can be changed to tune its operation to specific task at hand. In the following sections the basic setup parameters of Clarens will be covered.

2.3.1 Apache Web Server

The Apache[1] web server is controlled by one (or a series of) configuration files, starting from the file `$opkg_root/etc/apache/httpd.conf`. The format and meaning of options in this file is explained in detail at the web pages cited above, but the default configuration used by Clarens deserves some explanation.

- Firstly, the Apache server is run as the user set up by OpenPKG called `nuser`, short for `nobody user`. If the server was installed using the superuser installation procedure in 2.1, this would in fact be the `nobody` user which exists on most systems with extremely limited privileges. By default, this user cannot even log in to the system! The owner (user and groupname) of the server process is set up by the `User` and `Group` directives in the `httpd.conf` file.

  If the user installation procedure was used, the server will run as the same user that did the installation, since normal users are not able to change process ownerships and run applications as other users than themselves. In this case changing the `User` and `Group` directives will not have any effect. Except producing an error message in the logs!

- Secondly, the server is set up to listen on two ports, one for unencrypted (plaintext) connections, and one for Secure Sockets Layer (SSL) connections. The plaintext port is set up in the `httpd.conf` file by the `Listen` directive while the SSL port is set up by the same directive in the `ssl.conf` file, which is located in the same directory. These ports are set to values of 8080 and 8443 respectively by default.

  For a superuser installation the port numbers may be lower than 1000, with defaults of 80 and 443 assigned to the HTTP and HTTPS protocols by convention.

- Thirdly, variables controlling the `mod_python` module's behaviour are stored in a configuration file
  `$opkg_root/etc/clarens-config/httpd/clarens-server-default.conf`. For more information on the meaning of the options in this file, the `mod_python` documentation should be consulted. In general it should not be necessary to change this file.

  There may actually be several of these files for different Clarens installations running under the same Apache server.

2.3.2 Server Module Configuration

In the paragraphs all the setup parameters were for the purpose of telling the Apache web server to hand requests off to the `mod_python` module. Clarens itself also needs some information about where to find some files and store its own databases and so forth.

In order to make changing these parameters easier, the Clarens server package contains an application called `clarens-server-config`. This application takes care of creating a configuration file. By default the configuration file is stored in `$opkg_root/share/apache/clarens/clarens_config.p`

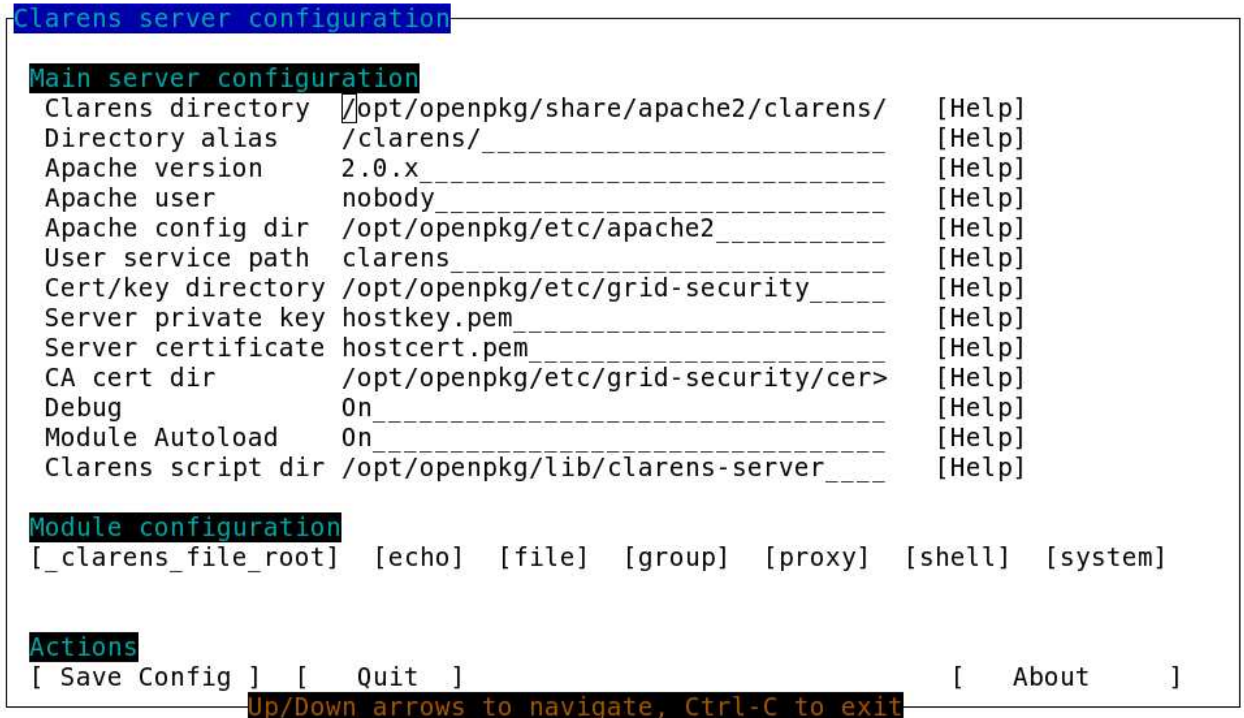

Figure 2.1: The `clarens-server-config` application

Figure 2.3.2 shows a screenshot of `clarens-server-config` in action. The section labeled *Main server configuration* shows the configuration parameters and their current values. To change the value of a parameter, use the up and down cursor keys to move the cursor to the chosen value and fill in the required text. Moving the cursor to one of the `[Help]` buttons and pressing `Enter` will bring up an explanation of the option in question.

Just as with the main configuration, all add-on modules can have their own sets of parameters. To change a module's parameters, move the cursor until the module name is highligted and press `Enter`. You will be presented with a list of options and Help values analogous to the main server screen.

After the configuration is changed it can be (and should be!) saved by pressing `Enter` on the *Save Config* option. The option values are saved in the directory
`$opkg_root/etc/clarens-config/conf/default`
The main server options are saved in the file `config` in that directory, with a subdirectory for each module also containing its `config` file.

The format of the config files consist of lines, each with comma separated fields for the option internal name, option value, option external name (displayed by the configuration application), help string, and the keyword `INTERN` or `EXTERN`. The latter values are written to a final cofiguration file named `clarens_config.py` that can be accessed by Clarens at runtime. `INTERN` values are only accessible at install time. Examples of such internal configuration values are the Apache server version and the Apache configuration file location.

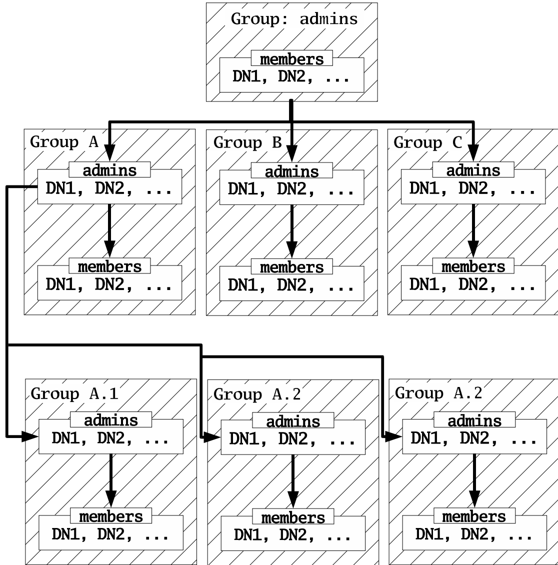

Figure 2.2: Clarens virtual organization diagram.

2.3.3 Virtual Organization

Each Clarens server instance manages a tree-like VO structure, as shown in Figure 2.2, rooted in a list of administrators. This group, named `admins`, is populated statically from values provided in the server configuration file on each server restart. The list of group members is cached in a database [2], as is all VO information. The `admins` group is authorized to create and delete groups at all levels.

Each group consists of two lists of DNs for the group members and administrators respectively. Group administrators are authorized to add and delete group members, as well as groups at lower levels. The group structure is hierarchical because group members of higher level groups are automatically members of lower level groups in the same branch.

The example in Figure 2.2 demonstrates the top-level groups `A`, `B`, and `C`, with second level groups `A.1`, `A.2`, and `A.3`.

A more concrete example might be to define groups `CMS`, `Atlas`, `LHCb`, and `Alice`, then for CMS, to define `CMS.USA`, `CMS.CERN`, `CMS.UK`, `CMS.Germany`. At the third level, one might define `CMS.USA.Caltech`, `CMS.USA.UFL`, `CMS.USA.FNAL`. Management for the latter three groups may then be delegated to the institutes themselves, thereby implementing a distributed trust model that has lower maintenance overhead as well as being more representative of the real organizational structure.

As a further optimization, the hierarchical information in the DNs may also be used to define membership, so that only the initial significant part of the DN need to be specified. DNs are structured to include information on the country (C), state/province (ST), locality/city (L), organization (O), organizational unit (OU), common name (CN), and e-mail address (Email). An example DN issued by the DOE Science Grid CA is
`/O=doesciencegrid.org/OU=People/CN=John Smith 12345`
for individuals and
`/O=doesciencegrid.org/OU=Services/CN=www.mysite.edu`

for servers. To add all individuals to a particular group, only
`/O=doesciencegrid.org/OU=People`
need to be specified as a member DN.

2.3.4 Method Access Control

Since version 0.5.0 the Clarens server enforces access control on *methods* accessed by clients based on user distinguished names, as well as groups of users.

The access control list (ACL) for the `echo` module in the above example can be specified using methods in the `system` module, or bootstrapped using a file in the same directory as the `__init__.py` file.

The `.clarens_access` file

The file specifying the ACL for a module is a Python script, named `.clarens_access` placed in the module's subdirectory. It should initialize a list with the name `access` in the main context. The access control list itself consists of a list of tuples (or lists), one for each ACL. An ACL may be specified for the entire module, or on a method by method basis.

An example to illustrate:

PYTHON

```
ORDER_ALLOW_DENY=0
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
```

The first tuple in the access list has an empty string as its first member, signifying that this ACL is to be applied to all methods in this module, unless overridden by a specific method ACL later. the second member The second member of the tuple is a list consisting of

- the ACL evaluation order
- a list of allowed distinguished names (empty in this case)
- a list of allowed groups
- a list of denied distinguished names
- a list of denied groups.
- a list with three members set to `None`. These are used for time values internally, and will be filled in by the Clarens server.

In this case the allowed DNs and groups lists are evaluated first, before the equivalent denied lists. That means that if a group or DN is present in the allowed and denied lists, the allowed list will have preference and the DN or group will be allowed. Groups can be created using the methods in the `group` module. The way that distinguished names (DNs) ar evaluated in the above lists deserve special mention: If a substring of the DN is found it is considered a match. E.g. the DOEGrids CA issues certificates with DNs of the form
`/DC=org/DC=doegrids/OU=People/CN=Joe User 12345`If access needs to be granted to all DNs issued by this CA, simply add the following to the allowed list:
`/DC=org/DC=doegrids`

Or if all DNs issued to people by this CA needs to access this module, add
`/DC=org/DC=doegrids/OU=People`

The ACL in the above example might then look like this:

PYTHON

```
ORDER_ALLOW_DENY=0
access=[("",[ORDER_ALLOW_DENY,   # Order
         ["/DC=org/DC=doegrids/OU=People"], # Allow all People trusted by
                                 # DOEGrids Certificate Authority
         [],                     # Allow group
         [],                     # Deny indiv default=all
         []  ,                   # Deny default=all
         [None, None, None]])]   # modtime, start_time, end_time
```

Since the DNs are always evaluated in the format with forward slashes as field separators, an individual DN is always matched by the string "/". Using that as an access specifier is a convenient way to allow or deny access to all users.

Individual methods

Some methods may should only be executed by certain users, like methods that perform some administrative function. In that case a tuple may be added for that method. E.g. if the `echo` module has a method `set_string` that may only be accessed by the `admin` group, but the `echo` method itself should still be executable to all, the ACL might look like this:

PYTHON

```
ORDER_ALLOW_DENY=0
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
        ("set_string",[ORDER_ALLOW_DENY,
         [],                     # Allow everybody who can log in
         ["admin"],              # Allow group
         [],                     # Deny indiv default=all
         []  ,                   # Deny default=all
         [None, None, None]])    # modtime, start_time, end_time
```

The `system` module

In the above ACL specifications, the module name was implicitly added to the method specification, e.g. `echo.set_string`. For the `system` module this is not done, implying that ACLs for other modules may be set using the `system` ACLs.

Since the modification time of the `.clarens_access` file is used to determine whether it should be parsed, an ACL specification for a module or method in its own directory and the `system` directory will mean that the most recently modified ACL will be used.

2.3.5 File Access Control

File ACLs are analogous to the method ACLs discussed above, with the addition of fields for write access. The ACL file should also be called `.clarens_access`, and such a file may be placed in each directory.

For example if the below ACL specification is placed in the Clarens file service virtual root directory, everyone will be given read access to the ”/” directory:

PYTHON

```
ORDER_ALLOW_DENY=0
ORDER_DENY_ALLOW=1
access=[("",                                # File or dir to apply ACL to
         [ORDER_DENY_ALLOW,                 # Order
         ["/"],                             # Allow indiv read
         [],                                # Allow group  read
         [],                                # Deny indiv read
         [],                                # Deny group read

         [],                                # Allow indiv write
         [],                                # Allow group write
         [],                                # Deny indiv write
         []])]                              # Deny group write
```

As with methods, an empty file or directory name indicates the current directory, while a named file of directory applies to its that entity in the current directory.

Extending this example to give wite access to the `rootfiles` subdirectory to people with DOE-Grids certificates:

PYTHON

```
ORDER_ALLOW_DENY=0
ORDER_DENY_ALLOW=1
access=[("",                                # File or dir to apply ACL to
         [ORDER_DENY_ALLOW,                 # Order
         ["/"],                             # Allow indiv read
         [],                                # Allow group  read
         [],                                # Deny indiv read
         [],                                # Deny group read

         [],                                # Allow indiv write
         [],                                # Allow group write
         [],                                # Deny indiv write
         []]),                              # Deny group write

         ("rootfiles",                      # File or dir to apply ACL to
         [ORDER_DENY_ALLOW,                 # Order
         ["/"],                             # Allow indiv read
         [],                                # Allow group  read
         [],                                # Deny indiv read
         [],                                # Deny group read

         ["/DC=org/DC=doegrids/"],          # Allow indiv write
         [],                                # Allow group write
         [],                                # Deny indiv write
         []])]                              # Deny group write
```

Avoiding possible conflicts

As the above example illustrates, it is possible to specify an ACL for a subdirectory (e.g. "rootfiles" above) at a higher level in the directory hierarchy (e.g. "/" above). Since it is also possible to do so in the subdirectory itself, there may be two files specifying the ACL for that entity.

This should be avoided at all costs, since it is impossible to determine which ACL specification will actually be used. I.e. specify the ACL of an entity in only one `.clarens_access` file!

Chapter 3

Client Usage

3.1 Command-Line Utilities

3.1.1 clarens-proxy-init

This command is used to create a so-called *proxy* certificate. This certificate actually consist of three parts: a temporary certificate, an accompanying private key, and a longer-lived certificate. The latter is used to sign the temporary certificate.

Shell

```
clarens-proxy-init -h

Usage: clarens-proxy-init [options]

Options:
  -h          Show this message
  -c FILE     Use FILE as certificate file
                  [default: /home/conrad/.globus/usercert.pem]
  -k FILE     Use FILE as private key file
                  [default: /home/conrad/.globus/userkey.pem]
  -t DAYS     Time that the proxy will remain valid in days
                  [default: 2]
  -b bits     Number of bits to use for key
                  [default: 512]
  -d          Show some debugging information
  -v          Print version number and exit
```

The use of a proxy certificate has several advantages:

- It is valid only for a limited time, shortening the time that a proxy can be used by potential attackers.
- Since it contains a true certificate and an accompanying key, it can be used for so-called *credential delegation*, where a third party can act on your behalf for a restricted time.

Of course, there are also several disadvantages:

- The limited validity period requires that the proxy be renewed if a long-running process or job makes use of the delegation feature.
- If the certificate and key are misapproriated, they may be used to impersonate the user who created the proxy. Revoking the user's certificate is the only recourse available when a proxy becomes compromised. The delays in propagating certificate revocation lists leaves enough time for an attacker to considerable damage. And of course issuing a replacement certificate may be slow.

Usually the command is issued from a shell prompt without any options. The user is then asked for the password used to decrypt the private key used to sign the newly created temporary certificate:

Shell

```
clarens-proxy-init
Enter pass phrase for /home/user/.globus/userkey.pem:
```

This process ensures that only the owner of the long-lived certificate can create proxy certificates. The default is to use the user's certificate and key files `$HOME/.globus/usercert.pem` and `$HOME/.globus/userkey.pem` To use different certificate and key files, specify the file names with the `-c` and `-k` options.

3.1.2 `clarens-ping`

In analogy to the well-known `ping` command, the *clarens-ping* command can be used to check whether a Clarens server responds to requests, and the turnaround times for those requests. The command is part of the `clarens-client-python` packages, and simply calls the server's `echo.echo` method.

Shell

```
clarens-ping --help

Usage:
 /opt/openpkg/bin/clarens-ping --<option>=<value> URL
   or
 /opt/openpkg/bin/clarens-ping --<option>

Currently options known are:
--debug    Turn on debugging
--help     Show this message
--host     Name of the host to connect to
--max      The maximum number of times to try
--path     The path to the clarens server on the remote host
--port     The port the remote server is listening on
--protocol The transport protocol: either http or https
--size     Message size, excluding header and encoding
--sleep    Delay between packets in seconds
--url      A complete URL to specify the location of the server
```

The simplest use of the `clarens-ping` command is with the URL of the server as the only argument. Pressing `Ctrl-C` aborts the process, and prints the minimum, average, and maximum response times of the server.

Shell

```
clarens-ping http://localhost:8080/clarens/

Contacting http://localhost:8080/clarens/...
OK
Received 289 bytes, time = 3.752 ms
Received 289 bytes, time = 4.278 ms
Received 289 bytes, time = 64.776 ms
Received 289 bytes, time = 4.043 ms
Received 289 bytes, time = 4.015 ms
rtt min/avg/max = 3.752/16.173/64.776 ms
```

3.1.3 `clarens-package-hostcert`

Since it is convenient to keep track of files stored as part of a particular installation via the RPM file database, this command provides a way to package a host certificate into an RPM. The RPM will also ensure that the certificate is installed correctly.

Shell

```
clarens-package-hostcert
Usage: clarens-package-hostcert <certfile.pem>
```

The command should be supplied with the file name of a PEM-encoded certificate, e.g.:

Shell

```
clarens-package-hostcert hostcert.pem
Wrote: /opt/openpkg/RPM/PKG/hostcert-ServerName-5e4666aa-1.noarch.rpm
```

The name of the package is constructed out of the certificate's distinguished name (DN), with the version number being a hash value calculated from the certificate. This ensures that the RPM package name should be unique. The resultant RPM package can be installed as usual:

Shell

```
rpm -i /opt/openpkg/RPM/PKG/hostcert-ServerName-5e4666aa-1.noarch.rpm
```

The certificate will be installed in `$opkg_root/etc/grid-security/hostcert.pem`.

3.1.4 `clarens-package-hostkey`

In analogy to the previous command, the host key can also be packaged:

Shell

```
clarens-package-hostkey
Usage: clarens-package-hostcert <keyfile.pem> <certfile.pem>
```

The key will be encrypted inside the RPM package since host keys are usually stored unencrypted in the server filesystem. This provides some modicum of security for storing the key in paces that may not be entirely secure.

E.g.:

Shell

```
clarens-package-hostkey hostkey.pem hostcert.pem

For security reasons we will encrypt the key inside the package
Enter Encryption Password:
Verifying - Enter Encryption Password:
Wrote: /opt/openpkg/RPM/PKG/hostkey-ServerName-5e4666aa-1.noarch.rpm
```

Upon installation of the resultant RPM package, the user will be prompted for the password used to encrypt the key. If the correct password is entered, the unencrypted key will be installed in `$opkg_root/etc/grid-security/hostkey.pem`. The encrypted key file will be stored in `$opkg_root/etc/grid-security/hostkey_unenc.pem`.

3.1.5 `clarens-package-cacert`

This command provides a convenient way to package certificate authority (CA) certificates.

Shell

```
clarens-package-cacert
Usage: clarens-package-cacert <certfile.pem>
```

E.g.

Shell

```
clarens-package-cacert cacert.pem
Wrote: /opt/openpkg/RPM/PKG/cacert-CAName-5e4666aa-1.noarch-anyos-hco.rpm
```

As with the host certificate, the package name is constructed in such a way that it should be unique. In addition, the dependencies of the RPM package will be set up to that the whole certificate chain must be installed. E.g. if the CA with the distinguished name `CAName` was issued by a higer level CA with a DN of `CA-pki`, an RPM package named
`cacert-CA-pki-XXXXXX-1.noarch.rpm`
will be required to be installed first. This might seem onerous, but using a CA certificate without having the full certificate chain available makes it virtually useless!

The RPM package will install the certificate in the directory
`$opkg_root/etc/grid-security/certificates/5e4666aa.0`, linked to the real file in `$opkg_root/etc/grid-security/certificates.real/cacert-CAName-5e4666aa` in the above example.

3.1.6 `clarens-package-cadir`

CA certificates are often distributed as a bundle, or are available as a set of files in a single directory. To package all the CA certificates in such a directory, use

Shell

```
clarens-package-cacert /my/dirname
```

The RPM files will be created exactly as if the `clarens-package-cacert` command was invoked for each of the files.

3.1.7 `clarens-package-root`

The ROOT[13] analysis package is distributed in binary form as a compressed tar archive. Since manually compiling this package from source may result is a package that is binary incompatible with the released binary versions, this command makes it possible to repackage the tar archive as an RPM package.

Usage is simply e.g.:

Shell

```
clarens-package-root root_v3.10.02.Linux.RH10.0.gcc33.tar.gz
```

The resultant RPM package will install files in `$opkg_root/lib/root/`. Upon initializing the OpenPKG environment, the binary and library paths and environment variables will be set up so that the ROOT application can be invoked from the command line by simply typing `root`.

3.2 Kerberos Certificate Authority

In a Kerberos environment users are authenticated by a central authority and a strong cryptographic *ticket* is created on the client system. This ticket is very similar to a proxy certificate, containing a public and private keypair. The Kerberos infrastructure can be leveraged to provide an authentication environment similar to that provided by a certificate authority that issues certificates by converting the temporary ticket information into an X509 certificate and private key pair.

This functionality is provided by the `kx509` utility, which can be downloaded and installed using the command `clump install kx509`.

To create a proxy certificate from a Kerberos ticket:

Shell

```
kinit
kx509
kxlist -p
```

The proxy certificate will be valid for the same period as the ticket.

3.3 Python

In the `clarens-client-python` package, the file `Clarens.py` contains a `Clarens` base class that can be used from the command line or from more complex scripts or programs. This base class has a dependency on the M2Crypto package, which is also used on the server.

3.3.1 Simple example

To initiate a client session, import the `Clarens` module, and instantiate a `clarens_client` object:

PYTHON

```
import Clarens
dbsvr=Clarens.clarens_client('http://localhost:8080/clarens/')
```

Note that the URI does not have a fixed form like some other XMLRPC servers use (e.g. always ending in the path `/RPC2`).

The object constructor has the following options:

OPTION	VALUES	default	DESCRIPTION
debug	0 or 1	0	Show requests responses when set.
certfile	Filename	/tmp/x509up_u$UID then	
		$HOME/.globus/usercert.pem	Path to a PEM-encoded certificate.
keyfile	Filename	/tmp/x509up_u$UID	Path to a PEM-encoded private key.
		then	
		$HOME/.globus/userkey.pem	
callback	Function	util.passphrase_callback	The name of a function
			that returns a password.
passwd	string	None	The private key password.

3.3.2 Procedure calls

Procedure calls implemented by the remote server may now be called as methods of the client object. The simplest method is the `echo.echo` method, which merely returns its argument. Return values are part of a list by convention of the XMLRPC protocol:

PYTHON

```
>>> dbsvr.echo.echo('Hello')
['Hello']
```

In this case, the string `Hello` would be returned as `['Hello']`.

There are a whole list of `system.`*`method`* methods implemented to query the server as to the methods it implements. These are discussed elsewhere.

3.3.3 Session termination

Using the `system.logout` method of the client object:

PYTHON

```
>>> dbsvr.system.logout()
0
```

3.3.4 Complete example

In summary, the complete example will look as follows:

PYTHON

```
import Clarens
dbsvr=Clarens.clarens_client('http://localhost:8080/clarens/')
dbsvr.echo.echo('Hello')
dbsvr.system.logout()
```

3.3.5 Binary return data

While encoding requests and responses simplifies server and client interactions, it does have one serious drawback: large binary responses are difficult to handle. Such a response must be returned as a string encoded in a text-friendly way, e.g. so-called base-64 encoding. This increases the CPU requirements on both the server and client. Since the data encoding increases the size of the response, more bandwidth is required to transport the data.

For that reason, Clarens returns binary data responses in raw form. This might confuse a client application expecting XML-encoded text. To enable or disable automatic XML deserialization, use the `enable_deserialize` and `disable_deserialize` methods as follows:

PYTHON

```
import Clarens
dbsvr=Clarens.clarens_client('http://localhost:8080/clarens/')
dbsvr.disable_deserialize()
content=dbsvr.file.read("/file.txt",0,-1)
dbsvr.enable_deserialize()
dbsvr.system.logout()
```

The variable `contents` should contain the content of the remote file after this code is executed.

3.3.6 Debugging

To see the requests being passed to the server, and the return values from the server enable the `debug` option:

PYTHON

```
import Clarens
dbsvr=Clarens.clarens_client('http://localhost:8080/clarens/',debug=1)
dbsvr.echo.echo('Hello')
dbsvr.system.logout()
```

3.3.7 Using a different certificate file

To use a different certificate and key file combination, use the `certfile` and `keyfile` options:

```
PYTHON
import Clarens
dbsvr=Clarens.clarens_client('http://localhost:8080/clarens/',
                             certfile='usercert.pem',
                             keyfile='userkey.pem')
dbsvr.echo.echo('Hello')
dbsvr.system.logout()
```

The certificate and key filenames must be accessible from the current working directory in the way they are specified. There are no special directories searched for these files.

3.4 ROOT

The ROOT client is written in C++, for use in both compiled and CINT-interpreted code. The client provides both the basic infrastructure to log into an Clarens server and execute RPC calls, as well as a high-level interface to the `file.<method>` methods. The latter can be used as a replacement for the standard `TFile` class.

This interface depends on Root, OpenSSL and the `cURL` HTTP transport library being present on the system.

3.4.1 TCWebfile

The most useful part of the client at this stage is the ability to open a remote file using this class, which inherits from `TFile` details are given in section *FIXME*

3.4.2 A simple example

The following example shows how to open the file named `hsimple.root` in the top-level directory on the remote server:

```
C++
{
 gSystem.Load("clarens.sl");
 gSystem.Load("clarens_file.sl");

 UserRSA *rsa=new UserRSA();
 Clarens *clarens=new Clarens("http://localhost:8080/clarens/",rsa);
 TCWebFile f("/hsimple.root",clarens);
 TBrowser T;
}
```

The first two `gSystem.Load` statements loads the Clarens client extensions into memory, making them available for the script of program to use.

The UserRSA class above holds the user's authentication credentials and can be reused multiple times for connecting to different servers.

The Clarens class represents a connection to one server. A connection may be established explicitly using the open method.

Finally, the TCWebfile class represents a remote file. The object f can be used the same as any read-only TFile, including browsing its contents using the builting object browser, as is demonstrated above.

Note that multiple files can be opened on the server using the same Clarens object, which will remain instantiated until all references to it have been removed, after which it can be deleted. Deleting the Clarens object will also log the user out of the server using the system.logout remote method.

3.4.3 Different certificate and key files

The default user certificate and private key files are looked up in a proxy certificate, or in the files $HOME/.globus/usercert.pem and $HOME/.globus/userkey.pem. To use a different set of files, add the certificate and key filenames as arguments to the UserRSA object constructor:

C++

```
{
 gSystem.Load("clarens.sl");
 gSystem.Load("clarens_file.sl");

 UserRSA *rsa=new UserRSA("mycert.pem","mykey.pem);
 Clarens *clarens=new Clarens("http://localhost:8080/clarens/",rsa);
 TCWebFile f("/hsimple.root",clarens);

 TBrowser T;
}
```

This example uses the same file for both the private key and the certificate. This happens to be the way that the grid-proxy-init command stores its temporary certificate and an unencrypted private key for the user with ID 510.

3.4.4 Caching file reads

Since the call latency for an RPC call on a wide area network (WAN) can be significant, it is always useful to to some client-side caching of the file in question. By default, a cache size of 1 MB is used. To change the cache size we instruct the the TCWebFile constructor to not create a cache through an optional fourth flags parameter:

C++

```
{
 gSystem.Load("clarens.sl");
 gSystem.Load("clarens_file.sl");

 UserRSA *rsa=new UserRSA("mycert.pem","mykey.pem);
 Clarens *clarens=new Clarens("http://localhost:8080/clarens/",rsa,0);

 f.UseCache(10,512*1024);  // Set cache size
 f.Init(kFALSE);            // Initialize the object

 TBrowser T;
}
```

In this example the `TCWebfile` object is set to use a cache of a 10 pages, with a page size of 512 kB each. The cache is not persistent, so that data downloaded by the client is lost when the `TCWebFile` is destroyed.

3.4.5 Multiple files on a single site

Multiple files can be opened using the same connection to the remote server.

C++

```
{
 gSystem.Load("clarens.sl");
 gSystem.Load("clarens_file.sl");

 UserRSA *rsa=new UserRSA();
 Clarens *clarens=new Clarens("http://localhost:8080/clarens/",rsa);
 TCWebFile f("/hsimple.root",clarens);
 TCWebFile f("/mydata.root",clarens);

 TBrowser T;
}
```

3.4.6 Multiple sites, single login

Opening files from multiple sites could be useful for doing simultaneous analysis on these files. This certificate and private key are read only once. If the private key is encrypted, the password is only needed once.

```
C++
{
 gSystem.Load("clarens.sl");
 gSystem.Load("clarens_file.sl");

 UserRSA *rsa=new UserRSA();

 // Create two connections
 Clarens *c1=new Clarens("http://localhost:8080/clarens/",rsa);
 Clarens *c2=new Clarens("https://otherhost:8443/clarens/",rsa);

 // Open one file on each server
 TCWebFile f1("/hsimple.root",c1);
 TCWebFile f2("/mydata.root",c2);

 TBrowser T;
}
```

3.4.7 TCSystemDirectory

The `TCSystemDirectory` class can be used to browse a remote filesystem through Clarens.

The constructor takes 5 arguments, the last of which is optional. Here is an example:

```
C++
{
 gSystem.Load("clarens.sl");
 gSystem.Load("clarens_file.sl");

 UserRSA *rsa=new UserRSA();
 TCSystemDirectory s("/","localhost","/",
            "http://localhost:8080/clarens/",rsa);
 TBrowser T;
 s.Browse(&T);
}
```

This should make a connection to the server running on the localhost at the URL specified in the fourth argument. The UserRSA object is used once again to hold authentication information.

The first and second arguments corresponds to the `Name` and `Title` arguments of a Root `TNamed` object, and is used to construct a tree in the TBrowser. The third argument is the full path of the directory being browsed on the Clarens server, as seen from the client side. In this case we browse the root directory.

A `TBrowser` object is instantiated, and the `Browse` method of the `TCSystemDirectory` is called. If the remote server was contacted successfully, a folder named `localhost` should show up in the left panel of the browser window.

Chapter 4

Writing Browser Interfaces

One of the most common applications on any modern desktop computer is the ubiquitous web browser. For this reason the Clarens framwework provides extensive support for writing browser-based interfaces that can be used from virtually anywhere using a standard browser.

This chapter will explain the basics of how this can be done using the facilities provided by the server-side Javascript and dunamic HTML framework in the `clarens-server-web<` package.

This chapter describes how to write services that use the Javascript Object Notation (JSON), which is conveniently handled natively by most modern browsers.

4.1 Files and paths

4.1.1 Accessing files on the server in different contexts

In order to have the files describing the client interface accessible by a browser they must be placed under the virtual file service root directory. On a default installation this will be under `$opkg_root/share/apache2/clarens/.clarens_file_root` or as set up by the `clarens-server-conf` utility. Files placed under this top-level directory will be available using the `file.`*`method`* calls with a virtual `/` directory prefix.

These files are also available using standard HTTP GET requests under the server URL, typically of the form `http://machine.name/clarens/`. This is summarized in table 4.1.

Table 4.1: Mapping of paths in different contexts

Context	Path
Filesystem	`$opkg_root/share/apache2/clarens/.clarens_file_root`
File service	`/`
Browser URL	`http://machine.name/clarens/`

4.1.2 Browser interface path conventions

In order to organize interfaces to different services, certain conventions are used for the placement of files. The paths given are relative to the `file` service virtual root:

Path	Description
`/`	Index file to display 'shell' of different services
`/web`	Files that make up the navigation panel and system-provided content
`/web/registry`	Registry files used to generate navigation panel
`/web/javascript`	System-provided Javascript files
`/web/images`	System-provided image files
`/web/images`	System-provided stylesheet files
`/web/`*service_name*	Service directories

4.1.3 Registry files

The web browser interface automatically creates a navigation panel on the left of the shell using files placed in the `/web/registry` directory as mentioned above.

These files are in the form of JSONRPC response methods as if generated by web service calls. They are text files that can be created with any text editor, or automatically using a scripting language like Python.

Each registry file may describe one or more service interfaces, with an associated category, name, description and a link to the interface HTML page. In the registry file these properties are as follows:

Element	Description
`name`	Short name to be displayed in the menu
`desc`	Longer description to be displyed as a 'tooltip'
`cat`	A category to place the item in
`file`	A relative link to the interface HTML file

The file containing these definitions must have the `.json` extension to be used in the navigation menu.

As an example, let's create a registry file for a `test` service, with a user interface in `/web/test`:

JSON


```
{"id":0,
"error": null,
"result":[{
"desc":"A test service",
"name":"Test Service"
"file":"test/test_interface.html",
"cat": "Test Category"}],
}
```


For anyone familiar with Javascript or Python, the above should be immediately obvious: the file consists of a mapping with elements `id`, `error`, and `result`. The `id` tag is ignored, while the `error` must always be `null`.

The `result` element contains a list of mappings, in this case only one, each with the information needed to describe one menu item.

4.2 Creating the interface

In this section an interface to the `echo` service will be created that allows a user to type in a string, press a button and have the server send back the string in question, to be displayed in a message window.

4.2.1 HTML file

The HTML file acts as a visual interface layout description from which the browser creates a page and loads the Javascript code that performs the needed actions.

HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
<head>

<script type="text/javascript" src="../javascript/json.js"></script>
<script type="text/javascript" src="test.js"></script>

</head>
<body>
<script type="text/javascript" src="../javascript/json_html.js"></script>

<form name="requestform">
Type a word: <input type="text" name="word">
<input type="button"
onclick="do_echo()"
value="Send!">
</form>

</body>
```

This page does a few things:

1. Loads the `../javascript/json.js` and `test.js` files in the `head` section of the page.

2. Loads the `javascript/json_html.js` file in the `body` of the page.

3. Defines a form with a text input box and a submit button, which when pressed, will call the `do_echo()` method.

4.2.2 Javascript file

Once the interface is defined, we need to respond to the submit button being pressed in the form, and call the server's `echo.echo` method using the word the user typed in as an argument. This is accomplished with the `do_echo()` callback defined in the `test.js` file quoted below:

Javascript

```
// Send echo request
function do_echo(){
  word=document.requestform.word.value;
  jsonrpc("echo.echo", word, echo_callback, null);
}

// echo callback
function echo_callback(result){
  alert(result);
}
```

First the value of the `word` form element is obtained, followed by a call to the `jsonrpc` method provided by the Clarens framework.

The `jsonrpc` method takes the following arguments:

Type	Name	Description
`string`	`method`	The name of the remote method
`list`	`arguments`	A list of arguments
`function`	`success callback`	Function to be called upon success
`function`	`fault callback`	Function to be called when a fault is returned

The arguments to the remote method is usually passed as a list, although in our example we passed a single string as an argument. The success callback, named `echo_callback` in the example, takes a result object as its only argument. The type of the result object is determined by the remote method.

The fault callback is called with the full JSONRPC result object as was demonstrated in section 4.1.3 for the registry files. In the case of a fault, or exception, returned by the server, the `error` element of the JSONRPC return object will contain information about the fault, while the `result` element will be set to `null`.

Chapter 5

Writing new Clarens modules

5.1 Introduction

Writing a Clarens server-side module runs the gamut from trivial to complex. Complexity mainly stems from the multi-process architecture of the Apache server, dealing with persistent database connections, and of course security considerations.

No specific Python programming experience is required, following and modifying the example should be sufficient for most purposes. For more information on interacting with Apache in this, environment see the `mod_python`[6] documentation.

5.2 First Example: the `echo` module

This module has only one method, `echo.echo` which simply returns its arguments. The module is a standard Python module script, stored in `$clarens-toplevel/echo/__init__.py` where it will be picked up by the Clarens server automatically when any method in the module is invoked.

5.2.1 Starting out

Like most Python modules, some secondary modules need to be `import`ed first:

PYTHON

```
import sys
from mod_python import apache
import clarens_util
```

This imports the sytem and `clarens_util` modules, as well as the Apache part of `mod_python`

5.2.2 Defining a new method

A method inside the module is a Python ¡em¿function¡/em¿:

PYTHON

```
def echo(req,method_name,args):
```

This define the `echo` function, with its arguments:

- `req` - the Apache request object
- `method_name` - the textual name of the method being invoked `echo.echo` in this case.
- `args` - A list of arguments passed to the function. In Python lists can contain any type of object, and the arguments will already be de-serialized from XML into Python objects.

Next, all methods should be documented so that the can be discovered by remote clients:

PYTHON

```
    """Returns the method argument"""
```

For our simple method, we construct a response and write the response to the client:

PYTHON

```
    response=clarens_util.build_response(req,method_name,args)
    req.write(response)
```

Note the Python language structure of indenting the contents of the function. Finally, return from the function:

PYTHON

```
    return apache.OK
```

This lets the Apache server continue with its processing chain (including possibly compressing the output).

5.2.3 Exposing the method

In order to allow methods in the script to be hidden, only those functions specifically exposed to the outside world can be called by clients. Publishing information about the function is done as follows:

PYTHON

```
methods_list={'echo':echo}
methods_sig= {'echo':['string,string',
                     'int,int',
                     'double,double',
                     'boolean,boolean',
                     'array,array',
                     'struct,struct']}
```

The `methods_list` variable is a dictionary, with a string, `'echo'`, identifying the method, and the callable method object that we defined ealier as data.

The `methods_sig` is another dictionary that describes the `echo` method signature, with its data being a list of possible arguments and return values. Each list element is a comma-separated string, with the first value being the type of thereturn value, and the folowing values are the types of the arguments. E.g.:

```
['string,string']
```

is a list with one element for a method that takes a string as argument, and returns a string.

In the case of the `echo` method is polymorphic, and each argument type is the same as the return type.

5.2.4 The full example

The full example would then look like this:

PYTHON

```
from mod_python import apache
import clarens_util

def echo(req,method_name,args):
    """Returns the method argument"""
    response=clarens_util.build_response(req,method_name,args)
    req.write(response)
    return apache.OK

methods_list={'echo':echo}
methods_sig= {'echo':['string,string',
                     'int,int',
                     'double,double',
                     'boolean,boolean',
                     'array,array',
                     'struct,struct']}
```

5.3 Debugging

There are two ways of debugging server modules: traditional printf-style messages that are recorded in the log file, or using the command line Python debugger.

5.3.1 Python debugger

Put the directive
`PythonEnablePdb On`
in the configuration file
`$opkg_root/etc/clarens-config/httpd/clarens-server-default.conf`.

Then start the Apache server with only one process:
`$opkg_root/sbin/httpd -X -f /opt/openpkg/etc/apache2/httpd.conf`
Any requests to `mod_python` handler will cause the Python debugger prompt to appear on the terminal where the server was started from.

Remember to remove the `PythonEnablePdb On` again when debugging is finished, otherwise an error will be reported by the server as follows:
`Handler 'clarens_server' returned invalid return code.`

5.3.2 Automatic tracebacks

As of version 0.6.9 of the `clarens-server` package, any exceptions generated by the server during the execution of the server-side module code above will be reported to the client. The amount of information returned depends on the value of the `PythonDebug` directive in the server configuration. The value of this directive can be set using the `clarens-server-config` utility described in section 2.3.2.

If debugging is turned on a traceback of the code, along with the called identity and client machine IP address is sent back. Consider the code snippet below, which uses the standard Python `xmlrpclib` module:

PYTHON

```
try:
  dbsvr.newservice.amethod("example")
except xmlrpclib.Fault,v:
  print v.faultString
```

The code tries to call the `newservice.amethod` server-side method. Imagine that the code that implements the module contains generates a division by zero error. In that case the above code could print something like the following:

output

```
Error in method call newservice.amethod made by
/DC=org/DC=doegrids/OU=People/CN=Conrad Steenberg 178947 from IP 127.0.0.1
Traceback (most recent call last):
  File "/opt/openpkg/share/apache2/clarens/system/__init__.py", line 1712,
in exec_method
    return method_object(req,method,args)
  File "/opt/openpkg/share/apache2/clarens/newservice/__init__.py", line 33,
in amethod
    sys.stderr.write(1/0)
ZeroDivisionError: integer division or modulo by zero
```

This would give the service developer engough information to know that there is a problem in line 33 of the code for the `newservice` module.

The raw XML message looks like this:

XML

```
<?xml version="1.0"?>
<methodResponse>
        <fault>
                <value>
                        <struct>
                                <member>
                                        <name>faultCode</name>
                                        <value><int>400</int></value>
                                </member>
                                <member>
                                        <name>faultString</name>
                                        <value><string>
Error in method newmodule.amethod made by
/DC=org/DC=doegrids/OU=People/CN=Conrad Steenberg 178947 from IP 127.0.0.1
Traceback (most recent call last):
  File "/opt/openpkg/share/apache2/clarens/system/__init__.py", line 1712,
in exec_method
    return method_object(req,method,args)
  File "/opt/openpkg/share/apache2/clarens/newmodule/__init__.py", line 33,
in amethod
    sys.stderr.write(1/0)
ZeroDivisionError: integer division or modulo by zero
</string></value>
                                </member>
                        </struct>
                </value>
        </fault>
</methodResponse>
```

If the `PythonDebug` directive is turned off, the output will look like this:

output

```
integer division or modulo by zero
```

This is obviously not very helpful for the developer, but does hide some information from any potential attackers.

5.4 Handling exceptions with `build_fault`

It is also possible for the service to generate its own fault responses using the `clarens_util.build_fault` method inside a `try - except` clause:

PYTHON

```
try:
  response=clarens_util.build_response(req,method_name,args)
except:
  response=clarens_util.build_fault(req,method_name,
             apache.HTTP_BAD_REQUEST,
            "Bad request echo %s"%(mod_name,args))
req.write(response)
return apache.OK
```

The arguments of the method is `build_fault(req,method_name,error_code,error_string)`. Always return `apache.OK` from the function, otherwise Apache will generate its own error message in `text/html` format, which may not be handled elegantly by all clients. If you do not want to supply your own error handling code, the Clarens server will also catch exceptions, and send a generic error message to the client.

5.5 Service initialization

The usual mode of operation for the `mod_python` module is for so-called *handlers* to be called once a request is received by the server that matches the requirements set in the configuration file. This may mean that the Python module that implements the handler is only imported once such a request is received.

The Clarens server augments this behaviour by also having the service modules be imported upon server startup to allow some initilization to be done if needed. Some examples of this includes populating a database of known services and methods, starting a process that advertises the services offered using the a discovery service, and making sure the file and method access control lists are loaded in the database for quicker access.

The logical way for services to initialize global variables, database connections etc. would be to put the initilization code in the module's global name space, so that it can be executed when the module is imported. The reality is that each module may be imported multiple times, e.g. by the main Clarens server as well as by other server modules.

To handle this in a more coordinated fashion, the main Clarens server will try to import all the modules it knows about once per process. It will then call a method named `_startup_init` in each module with three arguments:

- `config`: A dictionary containing the server configuration key value pairs. Both keys and values are strings.
- `modnames`: A sorted list of service names found by the server.
- `modules`: A dictionanry of service names and service module objects.

5.5.1 Example

An example `_startup_init` method might look like this:

PYTHON

```
def _startup_init(config, modnames, modules):
  dbdir=path_join([clarens_config.config['clarens_path'],".clarens_logins"])
  names=["db_file_indiv","db_file_group","md5db","sha1db"]
  clarens_util.register_open_dbs(clarens_util.dbd, names, dbdir)
```

See section 5.6 below for an explanation of what the above code achieves.

5.5.2 Per server initialization

The above initilization code will be called once per process, which is quite useful for database connections that cannot be shared between processes. In many cases the initilization code actually need to be called only once when the server starts up. The ACL database update is a good example of that.

In that case it is prudent to protect the initilization code with a global lock that precludes concurrent access by multiple processes. This can be achieved in several ways, one of which is to lock files. This method is fairly portable and has the desirable property that such locks are released when a process terminates, reducing the probability of a deadlock occurring.

The code in the example above may be protected with such a lock as follows:

PYTHON

```
# Global variable
lockfile

def _startup_init(config, modnames, modules):
  dbdir=path_join([clarens_config.config['clarens_path'],".clarens_logins"])
  names=["db_file_indiv","db_file_group","md5db","sha1db"]
  clarens_util.register_open_dbs(clarens_util.dbd, names, dbdir)

  try:
    lockfile=open(path_join([config['clarens_path'],'file','.lockfile']),"w")
    fcntl.flock(lockfile,fcntl.LOCK_EX|fcntl.LOCK_NB)
  except:
    return

  # Now update ACLs once per server startup
  update_acls(clarens_util.dbd["db_file_indiv"])
```

The ACL initilization code will only be called once per server startup. Of course a real implementation would also add some warnings to the exception handler code to warn of other exceptions that may occur when an attempt is made to open the lockfile.

Note that in the above code we do not close the `lockfile` object so that the lock is held for the entire time while the process is running.

5.6 Persistent data storage using `tdb`

It is often useful for services to store data in an organized way that may not quite need to full power of a RDBMS, but would be tedious to implement using flat files.

Clarens provides a high-performance key-value datastore for this purpose, which is also used to store most of the server's internal data structures. The `tdb` database stores key-value mapping per file, with only one open connection per file possible for a single process. For this reason opening and closing database *handles* need to be done in a coordinated fashion to prevent deadlocks and server process crashes.

5.6.1 Opening database handles

The `clarens_util` module provides a method for opening and maintaining a registry of `tdb` database instances. The method `register_dbs` should be called with three arguments:

- `registry`: a dictionary to hold the database registry. Always use `clarens_util.dbd` which is a resgistry managed by the main Clarens server.
- `names`: a list of database names that will be opened. Arguments are strings, or any hashable object that can be used for a dictionary key.

- path: a directory where the database file will be stored. The default is to use the .clarens_file subdirectory under the main server deployment directory, see the example below.

PYTHON

```
import clarens_util
import clarens_config

dbdir=path_join([clarens_config.config['clarens_path'],".clarens_logins"])
names=["db_file_indiv","db_file_group","md5db","sha1db"]
clarens_util.register_open_dbs(clarens_util.dbd, names, dbdir)

md5keys=clarens_util.dbd["md5db"].keys()
```

This example starts by importing two supporting modules that contain the utility methods and main configuration respectively.

5.6.2 Standard databases

The system module sets up certain standard databases with session data. These databases are available in the req.clarens dictionary:

1. db_env - the Berkeley db base handle, use this to create new databases
2. db_logins - key: user_nonce value
   value: list of server_nonce (server connection ID), connection startup time in seconds since 1970 (float), and the remote IP address req.connection.remote_ip The data gets serialized into the object by doing:

   PYTHON

   ```
   req.clarens["db_logins"][user_nonce]=
      pickle.dumps([public_server_nonce,
                    time.time(),
                    req.connection.remote_ip])
   ```

   The data can be deserialized as follows:

   PYTHON

   ```
   public_server_nonce, connection_time, remote_ip =
     pickle.loads(req.clarens["db_logins"][user_nonce])
   ```

   The user_nonce value is the authentication username, obtainable via

   PYTHON

   ```
   user_nonce=req.connection.user
   ```

3. db_certs - key: user_nonce', value: the user certificate, in PEM format

4. `db_methods` - key: method name, value a list of `[signature,document string]`
   The signature is the list discussed above5.2.3. Also use `pickle.loads()` to deserialize the data.

5.6.3 Printf-style debugging

To print messages in the log file, import the `clarens_util` module at the top of the new module's source file. Messages can then be logged with e.g.
`clarens_util.err_msg("myvalue=%s\n"%myvalue)`
The message should appear in the log file (e.g. `$opkg_root/var/apache2/log/error_log`).

Exceptions are usually handled by modules themselves which causes Python to log error messages to the log file. If a module fails to load in the first place, Clarens handles the resulting exception, and only reports that the module failed to load.

To log the exception in full in the logs, the exception must be raised in the file
`system/__init__.py`, at around line 998, after the message *Failed to load the module %s* was printed. Just add the statement
`raise`
to the exception handler.

In future a configuration switch will be provided to do this automatically.

Chapter 6

Service packaging and configuration

This chapter is neccesarily short becuase in versions of the Clarens aserver after 0.7.0 installation of new services became exeedingly simple: install the service files in the reight directory, and install one or more configuration files in the configuration directory.

6.1 Service installation

6.1.1 Service code

In order for a new service to be accessible it should be installed as a Python module under the Clarens toplevel service directory. Usually this means creating a subdirectory under
`/opt/openpkg/share/apache/clarens`
containing a file called `__init__.py` and `.clarens_access` as described in Chapters 5 and 2.

6.1.2 Configuration files

Services needing extra configuration parameters may create a file named `config` in a directory structure used by the `clarens-server-config` program.

By convention, each `config` file should be in a subdirectory with the same name as the service itself, under the toplevel directory. Multiple Clarens server instances can be run under the same Apache server under different virtual server paths, but it is assumed that there will always be one instance named *default*, with a set of configuration files under the toplevel directory `/opt/openpkg/etc/clarens-config/conf/default/`.

6.1.3 Minimal example

As an example, we create a service with the name `picalc` that calculates a large number of digits of the number π. The service contains one method that takes the number of digits required and returns the appropriate answer.

The service is also smart enough to store the answers in a small database so that it doesn't need to

recalculate the answers it already knows. It stores the answers in a database that can be specified by the server administrator.

If the default installation is used, the new service would consist of the following files:

File list

```
/opt/openpkg/share/apache2/clarens/picalc/__init__.py
/opt/openpkg/share/apache2/clarens/picalc/.clarens_access
/opt/openpkg/etc/clarens-config/conf/default/picalc/config
```

When the service is packaged, these files should be included in the package so that thay can be installed in the same locations.

6.2 Service configuration

6.2.1 Configuration file format

A short description of the configuration file format was given in section 2.3.2. A more elaborate description and example will be given here.

Each service has it's own configuration file, making it possible to easily distribute services without requiring the user to edit a central configuration file. Configuration files are text files that can be created with a text editor. Each file contains a set of options and their descriptions, one per line, that serves as input to the `clarens-server-config` utility that in turn will create a single Python configuration file that the Clarens server can use directly.

Each configuration line consist of a comma-separated list of five values as shown in Table 6.1.

Table 6.1: Configuration file description

Name	Description
Variable name	A gloablly unique configuration variable name. Because this is a string, it should be quoted
Value	The value of the variable, also a string
Label	A short human-readable string label of the configuration item. This label will be visible next to the item value in the `clarens-server-config` display
Help text	A longer description string of the configuration option. This will be displayed as the *Help* text for the option. Line breaks may be inserted using `\n`
Status	Either `INTERN` or `EXTERN`. The latter will be included in the final configuration made available to the Clarens server

All five configuration items should be on a sigle line, one per variable. The configuration file can also contain comments, denoted by the hash symbol, # at the beginning of the line.

Lines not conforming to the above format will simply be ignored by `clarens-server-config`.

6.2.2 Configuration file example

A short example configuration file might look as follows:

Configuration file

```
# Auto-generated configuration file created by clarens-server-config
'gridmapfile','/my/gridmap-file','Gridmap','Globus-style gridmap file',EXTERN
'suexec_path','/bin',"Suexec dir','Path to the clarens-suexec file',EXTERN
```

This file contains three lines, one for an initial comment and two containing definitions for the `gridmapfile` and `suexec-path` variables.

6.2.3 Command-line configuration

The configuration utility can also be run in a non-interactive batch-oriented mode which can be useful in scripts and to obtain debugging information printed during execution. In order to activate this batch mode, use the command
`clarens-server-config --batch --save`.

To print out the debugging information generated as the utility is running add the `--debug` option to the above. To obtain a full list of options, run the command using the `--help` option.

Chapter 7

Clarens authentication/login protocol

7.1 Introduction

The Clarens web service layer performs user authentication using X509 certificates issued by a certificate authority. It does so within the confines if the http Basic authentication protocol. This means that authentication information is passed along in the http header information using the `AUTHORIZATION` field. E.g.: `AUTHORIZATION Basic a80a844c376705cd8ecb8debdae0`

The string following the Basic keyword is a Base64 encoding of some information that the user wishes to pass to the server, usually the string `user:password`. In Clarens usernames and password are not used, but since the Basic authentication scheme is known by most http client programs and libraries, it eases the implementation of new clients to Clarens services. In server versions 0.6.3 or greater the cookies named `clarens_username` and `clarens_password` may alternatively be used from within clients that are unable to set the authentication headers.

Note that the following assumes some knowledge of encryption using public/private keys. Also note that the authentication can be done without using an encrypted link (i.e. using http instead of https), but https is recommended if security of transferred data is required. For the primary HEP application where Clarens is used http is normally used since the data is not ordinarily confidential, and the encryption/decryption slows down data transfer significantly.

7.2 Two steps

In order to do proper authentication of the client by the server, and vice versa, a challenge-response (or two-step) authentication must be used. The Clarens scheme is based on GSI, with the exception that does it was designed to fit into the http protocol without changes to servers or libraries.

The information interchange is described below, with some implementation examples interspersed written in Python. There is also a C++ implementation available which uses OpenSSL directly, see the Clarens layer for the Root package, which is part of the Clarens package.

1. Create a session em nonce value to identify the session. This should be as random as possible, but is combined with an IP address at the server to identify the session. E.g.:

PYTHON

```
random.seed()
ustring_raw="%s_%f_%f"%(os.getpid(),time.time(),random.random())
sha1=MessageDigest('sha1')
sha1.update(ustring_raw)
ustring=encodestring(sha1.digest())
```

That is, ustring is the SHA1 digest of `ustring_raw`.

2. Load the user's X509 certificate from a text file (in PEM-encoded format):

PYTHON

```
text_file=open(certfile)
text_ucert=text_file.read()
text_file.close()
```

3. Construct the first XML-RPC call to the server, calling the "system.auth" method. In the process the http header must be set as: `AUTHORIZATION: Basic <string>` where `<string>` is the Base64 encoding of `ustring:text_ucert`.

PYTHON

```
h.putheader("AUTHORIZATION",
            "Basic %s"%encodestring("%s:%s" % (ustring,text_ucert)))
```

Note that the above also removes any linefeed character from the resultant string to make sure the http server does not get confused, since a CRLF sequence is used to indicate the end of a header line in the http protocol.

A typical system.auth call would look like this::

HTTP

```
POST /xmlrpc/clarens_server.py HTTP/1.0
Host: localhost
User-Agent: xmlrpclib.py/0.9.9 (by www.pythonware.com)
Content-Type: text/xml
Content-Length: 105
AUTHORIZATION: Basic MkhVTm9VazYxbXArVEZLS0dCY2tIRlA3bjVzPQo6RnJvbSBi
<?xml version='1.0'?>
<methodCall>
  <methodName>system.auth</methodName>
  <params>
  </params>
</methodCall>
```

Most http libraries in one form or another allows you to set the username and password of the http request, and you would not need to do the header construction manually.

PYTHON

```
mtrans=BasicAuthTransport(ustring,text_ucert)
xmlrpclib.Server.__init__(self,url,transport=mtrans)
values = self.system.auth()
```

That is, a transport with the username and password of `(ustring,text_ucert)` is constructed and the xmlrpc client object is initialized using that transport.

4. The server will send back three strings in response to the system.auth call above:

XML-RPC

```
<?xml version=1.0?>
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value><string>STRING1</string></value>
            <value><string>STRING2</string></value>
            <value><string>STRING3</string></value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodResponse>
```

These three strings are as follows: STRING1: The PEM-encoded server certificate. (PEM-encoding is basically Base64) STRING2: A server 'nonce' value that was encrypted using the client's public key, and then Base64 encoded. STRING3: The client's 'nonce' value encrypted using the server's private key, and then Base64 encoded.

PYTHON

```
text_scert,
crypt_server_nonce_64,
crypt_user_nonce_64 = self.system.auth()
```

5. Read the server certificate contained in STRING1 and extract the public key contained therein. E.g.:

PYTHON

```
server_cert=X509.load_cert_bio(BIO.MemoryBuffer(text_scert))
server_pub_key=server_cert.get_pubkey()
server_pub_rsa=RSA.RSA_pub(m2.rsa_from_pkey(server_pub_key))
```

Verify that the key is authentic and still valid:

```
PYTHON

server_cert.verify()
```

6. Reverse the Base64-encoding of STRING2 and decrypt the resultant data to see if it matches the user nonce that we have constructed. This verifies that the server can encrypt data using its private key, and that we in turn can decrypt that data using its public key.

```
PYTHON

server_ustring =
   server_pub_rsa.public_decrypt(decodestring(crypt_user_nonce_64),
                                              mpadding)
```

   Then check if `server_ustring` is equal to ustring

7. Reverse the Base64-encoding of STRING3 and decrypt the resultant data using the client's private key. The resultant value is a secure session ID provided to us by the server.

```
PYTHON

server_nonce =
  user_priv_rsa.private_decrypt(decodestring(crypt_server_nonce_64),
                                             mpadding)
```

8. Calculate the SHA1 hash of the resultant data, and set the password in subsequent http request headers to the Base64 encoding of this hash value:

```
PYTHON

sha1=MessageDigest('sha1')
sha1.update(server_nonce)
mcrypt_server_nonce=encodestring(sha1.digest())
mtrans.set_password(mcrypt_server_nonce)
```

   Note that when the http header is constructed the user and password (or in our case the ustring and `mcrypt_server_nonce` values are again combined into a string (separated by a colon) and Base64 encoded when sent to the server as part of each RPC call.

9. Finally, to end the session, call the `system.logout` method of the server, which produces an XML request that looks like this:

HTTP

```
POST /xmlrpc/clarens_server.py HTTP/1.0
Host: localhost
User-Agent: xmlrpclib.py/0.9.9 (by www.pythonware.com)
Content-Type: text/xml
Content-Length: 107
AUTHORIZATION: Basic MkhVTm9VazYxbXArVEZLS0dCY2tIRlA3bjVzPQo6UFRHdHcrRVlX
<?xml version='1.0'?>
<methodCall>
  <methodName>system.logout</methodName>
  <params></params>
</methodCall>
```

The server responds with and integer of value 0 when the call succeeds:

HTTP

```
HTTP/1.1 200 OK
Server: Sourcelight Technologies py-xmlrpc-0.8.8.2
Content-Type: text/xml
Content-length: 220
<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value><int>0</int></value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodResponse>
```

7.3 SSL based authentication

Since the SSL key exchange algorithm makes it possible to do certificate-based authentication, some of the above procedure is redundant in that case. Furthermore, some clients may not have access to cryptographic functions, even though they are able to make SSL connections with certificate-based authentication, e.g. a JavaScript client running inside a web browser.

To allow for authentication under these circumstances, the `system.auth2` server-side function was created. The function is called without an argument, and returns the server certificate, client certificate and new password to be used as strings. In addition, the user and password combination in the Authentication headers or cookies should be set to a session key as in the non-SSL case and the string `BROWSER` respectively.

The username and new password should be used in subsequest RPC calls, in either the Authentication headers or cookies in the same way as in the non-SSL case. E.g.:

PYTHON

```
server_cert, user_cert, new_passwd = self.system.auth2()
mtrans.set_password(new_passwd)
```

7.3.1 Proxy authentication over SSL

As of version 0.6.3 of the Clarens server, any SSL connections made using an accepted certificate/key pair or a so-called 'proxy certificate' will automatically result in a session being created on the server, without the need to call one of the `system.auth/auth2` methods.

Proxy authentication requires that the Gridsite Apache module be installed and configured on the server. On an OpenPKG-based installation the command
`clump install gridsite`
will install and configure this package automatically.

7.4 GET requests

At the moment HTTP GET requests (as opposed to POST requests used for RPC calls) do not result in a session ID being created automatically, except in the case of an authenticated SSL connection. Instead, non-authenticated GET requests are assigned an internal distinguished name of '/' (forward slash) and ACLs are evaluated as if a `file.read` method was called.

7.5 Other transport protocols

Although web services are, as the name implies, mostly implemented using the HTTP protocol, the stateless nature of this protocol makes it unsuitable for some server-side applications. An example of such an application is a secondary network service such as a storage server, e.g. the Storage Resource Broker (SRB) or dCache where reconnecting to the secondary server upon receipt of each HTTP request would be prohibitively slow.

For this reason, future Clarens services would also be usable from a stateful protocol, e.g. FTP.

7.6 Discussion

The above protocol is designed to prove the user and server identities to both parties without prior knowledge of each other, except through the ability to check certificate signatures against known CA signatures.

Since the http protocol is designed to handle multiple requests from the same client using one or more socket connections, the server and user session ids remain valif between requests, and the

server should store those session ids. This makes it possible to have a single sign-on for a client to use different applications, or different instances of the same application to connect to the server and make requests, as long as every application knows the two session ids. Obviously the server should keep track of who makes these reqeusts, and this is done by ensuring that a pair of session ids are unique to a unique IP address.

It is also possible to connect to multiple servers from a single client application, the only requirement is that a session id pair must be negiotiated with every server. Note that most private encryption keys are themselves encrypted, so that a password is usually required to enable the client to perform that negotiation process.

Note that Globus works around this inconvenience by creating a temporary unencrypted public/private keypair through the `globus-proxy-init` command, which in turn presents its own set of problems. It is also once again recommended that SSL encryption (https) be used if data confidentiality is important.

Bibliography

[1] The Apache web server web site, `http://www.apache.org`.

[2] The Berkeley DB web site, `http://www.sleepycat.com`.

[3] Globus alliance web site, `http://www.globus.org`.

[4] Foster, I. and C. Kesselman, *The Grid: Blueprint for a new Computing Infrastructure*, Morgan-Kaufman, 1999.

[5] The Java page of Sun Inc., `http://java.sun.com`.

[6] The `mod_python` web page `http://www.modpython.org`.

[7] OpenPKG web page, `http://www.openpkg.net`.

[8] Pacman web page, `http://physics.bu.edu/ youssef/pacman/`.

[9] Particle Physics DataGrid, `http://www.ppdg.net`.

[10] The Python language web site, `http://www.python.org.`

[11] The Python binding to the Berkeley DB web site, `http://pybsddb.sf.net.`

[12] Redhat Package Manager, `http://www.rpm.org`.

[13] ROOT analysis application, `http://root.cern.ch`.

[14] Telecom Glossary 2000, Asymmetric encryption, Document T1.523-2001, `http://www.its.bldrdoc.gov/projects/devglossary/`.

[15] Yellow dog Updater Modified web page, `http://linux.duke.edu/projects/yum/`.

Copyright © 2005 California Institute of Technology.

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License v1.0.

Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.

Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.