Remote Control and Configuration Guide

August 30, 2005

Chapter 1

Remote Control and Configuration of Applications

1.1 General description

MonALISA Service has incorporated an Application Control Interface (AppControl) that allows the farm administrator to remotely start, stop, restart and configure different applications. This service starts automatically when the MonALISA service starts.

AppControl Service is modular. New modules for remote controlling desired applications can be simply developed and securely loaded in the MonALISA Service.

The security part is important for this remote administration interface. All the communication between clients and server is secured using GSS over PKI. The server has a keystore with the clients' public keys, so only the administrators can access this application.

For each controlled application there is a corresponding module. Each module can have multiple instances with **different** configuration files.

When a module is loaded you have to specify a **unique** file name which will store the specified modules configuration. After that, you have to correctly configure the module for working properly.

New modules can be added at any time by uploading a `.jar` file with classes that correspond with the module functionality and conform to a given standard. For more details, please see Section 1.3 section. The upload of the new module can be easily done from the client interface.

The AppControl has some default module:

- **Apache** module lets you start, stop and configure Apache web server remotely.
- **Bash** module lets you execute commands remotely.
- **Proc** module is for browsing the `/proc` directory.
- **MonALISA** lets you remotely configure and execute MonALISA service.

There are two *clients* that can be used: a graphical one incorporated in the MonALISA Client GUI, and a command line one.

1.2 Client-Server Protocol

The protocol is a text based, request-response one. The server messages have one of the following format:

```
+OK
<lines to be parsed by the client as response>
.
```

or

```
-ERR <error message>
```

so the server responses starts with +OK in case of a correct execution of the client request, or with -ERR in case of an error message. If the response was positive (i.e. +OK), then the client has to read the actual response until he receives a single dot on a line. If the original output contained a single dot on a single line, then this dot is transformed in two dots by the server, like .. instead of . .

Many of the commands parameters and output strings are encoded using URLEncoder with UTF-8. Whenever you will see *enc(something)* it means that *something* is encoded using this encoder.

The set of commands that the server can process are:

- **availablemodules** - lists all the available modules on the server, one per line.
- **loadedmodules** - lists all the loaded modules on the server, one per line.
- **deletemodule enc(<module_name>) enc(<configuration_file>)** - when receiving this command, the server will delete the module with <*module_name*> name and with the configuration file <*configuration_file*> from the loaded modules.
- **createmodule enc(<module_name>) enc(<configuration_file>)** - receiving this command, the server will add a new module with name <*module_name*> and with configuration file name <*configuration_file*> to the list of loaded modules.
- **start enc(<module_name> : <configuration_file>)** - this command starts the module with <*module_name*> name and with configuration file <*configuration_file*>.
- **stop enc(<module_name> : <configuration_file>)** - this command stops the module with <*module_name*> name and with configuration file <*configuration_file*>.
- **restart enc(<module_name> : <configuration_file>)** - this command restarts the module with <*module_name*> name and with configuration file <*configuration_file*>.
- **status enc(<module_name> : <configuration_file>)** - this command returns the status of the module with <*module_name*> name and with configuration file <*configuration_file*>. "0" means that the module is stopped, "1" means that the module is running, and "2" means that the satus of the module is unknown.
- **info enc(<module_name> : <configuration_file>)** - this command will return the configuration file <*configuration_file*> of the module <*module_name*> packed as an XML response. The XML response looks like this:

```
<config app=ApplicationName>
  <file name=AppConfigurationFileName>
    <key name=ConfigurationKey value=Value line=N read=true|false write=true|fals
    <section name=ConfigurationSection value=Value line=N read=true|false write=t
   ......
      Other keys in this section
   ......
    </section>
  </file>
</config>
```

For example let's look at a part of the XML for Apache's `httpd.conf`:

```
<config app="Apache">
  <file name="httpd.conf">
    <key name="ServerType" value="ceva" line="51" read="true" write="true"/>
    <key name="ServerRoot" value="%22%2Fusr%22" line="62" read="true" write="true
       ...
    <section name="IfModule" value="mod_mime_magic.c" line="506" read="true" writ
```

```
        <key name="MIMEMagicFile" value="%2Fetc%2Fapache%2Fmagic" line="507" read="true"
      </section>
         ...
      <section name="VirtualHost" value="*" line="1010" read="true" write="true">
        <key name="SeverName" value="localhost" line="1011" read="true" write="true"/>
      </section>
    </file>
  </config>
```

- **exec enc(<module_name> : <configuration_file>) enc(<command>)** - this command returns execution results of the command <*command*> on the module <*module_name*> with configuration file <*configuration_file*>. For example, execution the **`ls -l`** command on the bash module makes sense.
- **update enc(<module_name> : <configuration_file>) enc(<update_comm>)** - *
- **getconfig enc(<module_name> : <configuration_file>)** - this command returns the configuration file of the module <*module_name*> with configuration file name <*configuration_file*>.
- **updateconfig enc(<module_name> : <configuration_file>) enc(<configuration file content>)** - this command will modify the content of the configuration file for module <*module_name*> with <*configuration file content*>.
- **upload enc(<file_name>) enc(<binary file content>)** - this command creates a new available module with name <*file_name*> uploading the `.jar` archive with content <*binary file content*>.

1.3 Writing New Modules for AppControl

1.3.0.1 The `lia.app.AppInt` interface

All the modules must implement the `lia.app.AppInt` interface and must be packaged in `.jar` files that exactly respect the package structure.

The definition for `lia.app.AppInt` is:

```
package lia.app;

public interface AppInt {

    public boolean start();
    public boolean stop();
    public int     status();
    public String  info();
    public String  exec(String sCmd);
    public boolean update(String sUpdate);
    public boolean update(String sUpdate[]);

    public String  getConfiguration();
    public boolean updateConfiguration(String s);

    public boolean init(String sPropFile);

    public String  getName();
    public String  getConfigFile();

} // end of interface AppInt
```

`start()` This function should start the service and return *true* if the service could be started and *false* if the service could not be started.

`stop()` This function should stop the service and return *true* if the service could be stopped and *false* if the service could not be stopped.

`status()` Returns one of the following codes:

- `lia.app.AppUtils.APP_STATUS_STOPPED` (0) - the application is not running
- `lia.app.AppUtils.APP_STATUS_RUNNING` (1) - the application is running
- `lia.app.AppUtils.APP_STATUS_UNKNOWN` (2) - application status could not be determined

`info()` Returns a string with the application configuration files as an XML. See the examples above to see how the XML looks like.

`exec(String)` Executes the given command and returns the output of the command. You can return null if the application you are controlling does not accept any user commands.

`update(String)` Changes the application configuration files according to the given argument. You should implement the commands explained in the Client-Server protocol document. The return value must be true if the requested update could be done or false if the configuration could not be updated.

`update(String [])` Executes a set of updates. It's implementation might be as simple as:

```
for (int i=0; i<sUpdate.length; i++) update(sUpdate[i]);
```

`getConfiguration()` Returns the content of the module's configuration file as a string value. You should use `lia.app.AppUtils.getConfig(Properties prop, String sFile)`

`updateConfiguration(String)` Replaces the content of the configuration file with the given string. You should use `lia.app.AppUtils.updateConfig(String sFile, String sContent)`

`init(String)` This function is called by the main program when the module is loaded. The parameter is the module's configuration file. You should use `lia.app.AppUtils.getConfig(Properties prop, String sFile)` to read the contents of this file.

`getName()` Should return the complete name of the module to make sure that there is no conflict in names.

`getConfigFile()` Returns the configuration file name given as parameter to `init(String)`.

1.3.0.2 The `lia.app.AppUtils` class

This class offers functions that will ease the writing of new modules. We strongly encourage you to use these functions so whenever there is a change in the main code all the modules will keep working.

The classes public, static functions are:

```
String enc(String s);
String dec(String s);
String getOutput(String s);
String getOutput(String vs[]);
java.util.Vector getLines(String s);
void getConfig(Properties prop, String sFile);
boolean updateConfig(String sFile, String sContent);
```

`enc(String s)` Returns the URLEncoded value of the given parameter using the UTF-8 charset.

`dec(String s)` Returns the URLDecoded value of the given parameter using the UTF-8 charset.

`getOutput(String s)` Returns the output generated by the given system command or null if the command could not be executed. You can separate the parameters by spaces and you can enclose a large parameter (with spaces) between " characters. This function only builds a `String[]` of the command tokens and calls `getOutput(String vs[])`.

`getOutput(String vs[])` Returns the output generated by the given system command or null if the command could not be executed.

`getLines(String s)` Returns a `java.util.Vector` having each line of s as an element. It saves you from parsing the output of a command or the content of a text configuration file.

`getConfig(Properties prop, String sFile)` Loads the contents of the sFile file from `conf/` folder into the prop `Properties` object. You should use this method to load the configuration file instead of directly using the `conf/sFile` file.

`updateConfig(String sFile, String sContent)` Writes the value of `sContent` into `sFile`. You should use this method instead of directly writing the string to `conf/sFile` file because the configuration files' location might change in the future.

1.4 Remote Interface for MonALISA Modules Management

As an administrator, you have also access for modules management in MonALISA Service. Using this interface, which is also integrated with the MonALISA Client, you can start, stop, restart or upload a module in MonALISA Service.

1.5 Clients for the Application Control Interface

There are two clients: *a graphical one* and *a command line one*.

The simple command line client has a command, **help** which shows all the available commands and how to use them. For details, please see Section 1.1.

The graphical interface is integrated in the MonALISA client, but, for accessing it, you must have the right keystore. It presents Application Control server commands in a nice and friendly way.