MonALISA User Guide

January 25, 2005

Preface

The MonALISA framework provides a distributed monitoring service system using JINI/JAVA and WSDL/SOAP technologies. Each MonALISA server acts as a dynamic service system and provides the functionality to be discovered and used by any other services or clients that require such information.

MonALISA is entirely written in java. The distribution is packed in several self contain jar files. Practically it is required to have only java installed on the system hosting the MonALISA Service (NOT on the monitored nodes!). The way to configure and use the service is described in this guide. MonALISA also allows to dynamically add new monitoring modules, Filters or Agents.

IMPORTANT

Running a MonALISA service does not require any root privilegies, and we strongly suggest to do not use the service from a root account.

WARNING

Do not run MonALISA on a distributed file system!

Chapter 1

MonALISA Installation Guide

1.1 What you need for running MonALISA service?

For running a MonALISA service you need to have the java runtime environment (j2se 1.4.2 or higher) installed on one system that will run the Monitoring Service for an entire farm. For development of dedicated modules or agents the user should install the entire JDK.

Setting the environment to run java may look like this:

```
JAVA_HOME=$HOME/JAVA/jdk
export JAVA_HOME
export PATH=$JAVA_HOME/bin:$PATH
```

1.2 Tools used for getting monitoring information

Collecting the monitoring information can be done in several ways using dynamically loadable modules.

It is possible to collect information using:

- SNMP demons;
- Ganglia;
- LSF or PBS batch queueing systems;
- Local or remote procedures to read `/proc` files;
- User modules based on dedicated scripts or procedures.

1.2.1 SNMP

MonALISA has dedicated modules to collect values provided by snmp demons. Using snmp modules requires that the snmpd demons are installed and properly configured on the nodes or network elements (switches or routers) the user want to monitor.

Please see Section 6.1 to see how to test the well installation of snmp.

1.2.2 Kernel /proc files

Modules to collect the system monitoring information from the kernel are part of the MonALISA distribution. These modules are mainly design to be used on the node MonALISA service is running but they may also be used on remote systems via rsh or ssh.

1.2.3 Ganglia

Ganglia is a well known monitoring system which is using a multi cast messaging system to collect system information from large clusters. MonALISA can be easily interfaced with Ganglia. This can be done using the multicast messaging system or the gmon interface which is based on getting the cluster monitoring information in XML format. In the MonALISA distribution we provide modules for both these two possibilities. If the MonALISA service runs in the multicast range for the nodes sending monitoring data, we suggest using the Ganglia module which is a multicast listener. The code for interfacing MonALISA with Ganglia using gmon is `Service/usr_code/GangliaMod` and using the multicast messages is `Service/usr_code/GangliaMCAST`. The user may modify these modules. Please look at the service configuration examples to see how these modules may be used.

1.3 Installing the MonALISA Service

The MonALISA Service package comes with installation scripts which can be followed step-by-step for an automatic installation procedure, or you can manually configure the service.

The automatical installation of MonALISA Service:

- The scripts from the distribution that help you install and configure MonALISA are `install.sh` and `install2.sh`.
- The `install.sh` script will check to see whether the user is root or not. If the user is not root, then the `install2.sh` script is executed.
- If the user is root then he will be asked for an account from which MonaLisa will run.
- If the specified account does not exist then the script will attempt to create one, it will create a `monalisa_install` directory in this account, it will copy the `install2.sh` and `MonaLisa.v1.2.tar.gz` in the user's `~/monalisa_install` directory, and will start `install2.sh` from there. After `install2.sh` finishes its execution, the `~/monalisa_install` will be deleted.

The `install2.sh` script will ask for a destination folder, unpack the `MonaLisa.v1.2.tar.gz` file and ask for the farm configuration options. Make sure that the destination folder chosen is not `.(monalisa_install)` or something like (`./monalisa_install/<destination_folder>`) because the `monalisa_install` directory will be deleted after the installation procedure.

After unpacking the service,the `install2.sh` script will ask the user for abasicconfiguration. You will be asked to set the farm name (default to thehostnameof the computer) and you have to be sure that this name is unique in theMonALISA environment. You will also be asked to set the Latitude and Longitudeof the server. Aproximative values can be found at <`http://geotags.com/`>.

If the destination folder already contains an older MonALISA installation then all the configuration files are kept unchanged. In fact the `CMD`, `TEST` and `VDTFarm` folders remain untouched, which is bad if you runan older version of MonALISA. So if you want to upgrade from an older MonALISAinstallation please choose another destination folder and then heck if thenewly generated configuration is correct.

1.3.1 What does the service directory contain ?

After unpacking the service archive, you can find the following structure:

Service CMD this directory contains a set of scripts that are used for the management of the MonALISA service execution:

- the `MLD` script is used to start the Monitoring Service from `init.d`. See details in Section 2.6 section from this guide.
- the `CHECK_UPDATE` script is used to automatically update the Monitoring Service. See details in Section 2.7 section from this guide.
- the `ML_SER` script is used to start, stop or restart the Monitoring Service.
- in the `ml_env` file you have to set some environment variables. This is the only file from this directory that can be modified. See details about how to set these variables in Section 2.1 section from this guide.

SSecurity this directory contains the farm keystore (`FarmMonitor.ks`). As an administrator of the farm service, you can export and import certificates or create keys using `exportCert`, `importCert` and `genKey` scripts.

lib this directory contains the service packages

ml_dl it contains the dl jar for registering in reggie jini service.

myFarm In this directory you can set your farm. It contains the files that defines the characteristics of your farm. You can rename it with the name of the farm.

usr_code this directory contains sources of some module examples. See details in the Section 2.9 section from this guide.

bin contains tools used from the scripts from `Service/CMD`.

policy contains policy files for running the MonALISA service.

util contains different simple programs, their source and scripts for getting useful information:

ShowReceivedValues a short program example that interrogates a farm database and gets results from it. It receives as argument a configuration file, like the `Service/TEST/ml.properties`.

ShowStoreConfig a short program example that parses a farm configuration file (as `Service/TEST/ml.properties`) received as argument and shows the database tables structure used.

SimpleClient a program example that finds the MonALISA farm services registered in the reggie services from the locators given in the `locators.conf` file and shows their attributes.

SimpleDBShell a set of useful scripts for getting information from a mysql farm database. For using these scripts you must first edit the `mysql_console.sh` script, set the variables from here and delete the following lines:

```
echo "Please edit mysql_console.sh first" > /dev/stderr
exit
```

All the other scripts use ”mysql_console.sh”. Use them for getting information from the farm database tables.

Chapter 2

MonALISA Configuration Guide

There are three configuration files that the user can modify for specifying farm service environment and characteristics: a global configuration file (`$MonaLisa_HOME/Service/CMD/ml_env`), and the others used by MonALISA itself (`$MonaLisa_HOME/Service/<YOUR_FARM_DIRECTORY>/ml.properties` and `$MonaLisa_HOME/Service/<YOUR_FARM_DIRECTORY>/<YOUR_FARM_CONF_FILE>.conf`).

2.1 Global configuration file

The file for the global configuration is `$MonaLisa_HOME/Service/CMD/ml_env`. The variables that the user has to set or can set are:

MONALISA_USER the name of the user that is running the service. It will not start from other account or from the root account.

JAVA_HOME the path to your current JDK.

SHOULD_UPDATE whether or not MonALISA should check for updates when it is started. If this parameter is "true" when MonALISA is started, first it will check for updates and after that it will start. If set to "false" it will not check for updates. This parameter is also used to check for autoupdates when it is running. Please see Section 2.7 from this user guide.

MonaLisa_HOME path to your MonALISA installation directory. Environment variables can also be used. (e.g `${HOME}/MonaLisa`)

FARM_HOME path to a directory where reside your farm specific files. It's better to place this directory in the Service directory. (e.g. You can use the variable `MonaLisa_HOME` defined above. `${MonaLisa_HOME}/Service/MyTest`. MonALISA comes with a simple example in `${MonaLisa_HOME}/Service/myFarm`.

FARM_CONF_FILE the file used at the startup of the services to define the clusters, nodes and the monitor modules to be used. It should be in the `${FARM_HOME}` directory. (e.g **`FARM_CONF_FILE="${FARM_HOME}/mytest.conf"`**).

FARM_NAME the name for your farm. (e.g **`FARM_HOME="MyTest"`**). We would like to ask the users to use short names to describe the SITE on which they are running MonALISA.

JAVA_OPTS is an optional parameter to pass parameters directly to the Java Virtual Machine (e.g **`JAVA_OPTS="-Xmx=128m"`**).

2.2 The MonALISA properties

The file `$MonaLisa_HOME/Service/<YOUR_FARM_DIRECTORY>/ml.properties` is specific for your farm configuration.

You can specify here:

- what lookup services to use (**`lia.Monitor.LUSs`**);
- the jini groups that your service should join (**`lia.Monitor.group`**);
- the location of the farm server (**`MonaLisa.LAT, MonaLisa.LONG, MonaLisa.Country`**);
- Web Services settings (**`lia.Monitor.startWSDL=true`** starts the MonALISA web service, **`lia.Monitor.wsdl_port`**);
- database configuration (**`lia.Monitor.keep_history`** how long to keep data in farm database, parameters to configurate database tables, etc.);
- parameters for logging (**`.level`** - the logging level - defaults to **`INFO`**, etc.)

You will find explanations before every field for setting it correctly).

2.3 The MonALISA Farm Configuration and Measured Parameters

The MonALISA service is using a very simple configuration file to generate the site configuration and the modules to be used for collecting monitoring information. By using the administrative interface with SSL connection the user may dynamically change the configuration and modules used to collect data.

It is possible to use the build modules (for snmp, local or remote `/proc` file...) or external modules. We provide several modules which allow exchanging information with other monitoring tools. These modules are really very simple and the user can also develop its own modules.

Below we will present a few simple examples in how to make the configuration for a farm. This file is the **`.conf`** file from your `Service/<FARM_DIRECTORY>` directory.

2.3.1 Monitoring a Farm using snmp

The configuration file should look like this:

The first line (**`*Master`**) defines a Functional Unit (or Cluster). The second line (**`>citgrid3.cacr.caltech.edu citgrid3`**) adds a node in this Functional Unit class and optionally an alias. The lines:

```
monProcLoad%30
monProcIO%30
monProcStat%30
```

define three monitoring modules to be used on the node "citgrid3". These measurements are done periodically, every 30s. The **`monProc*`** modules are using the local `/proc` files to collect information about the cpu, load and IO. In this case this is a master node for a cluster, were in fact MonALISA service is running and simple modules using the `/proc` files used to collect data.

The line:

```
*ABPing{monABPing, citgrid3.cacr.caltech.edu, " "}
```

defines a Functional unit named "ABPing" which is using an internal module **`monABPing`**. This module is used to perform simple network measurements using small UDP packages. It requires as the first parameter the full name of the system corresponding the real IP on which the ABping server is running (as part of the MonALISA service). The second parameter is not used. These ABPing measurements are used to provide information about the quality of connectivity among different centers as well as for dynamically computing optimal trees for connectivity (minimum spanning tree, minimum path for any node to all the others...)

Example 2.3.1: Farm configuration with SNMP

```
*Master
>citgrid3.cacr.caltech.edu citgrid3
monProcLoad%30
monProcStat%30
monProcIO%30

*ABPing{monABPing, citgrid3.cacr.caltech.edu, " "}

*PN_CIT
>c0-0
snmp_Load%30
snmp_IO%30
snmp_CPU%30
>c0-1
snmp_Load%30
snmp_IO%30
snmp_CPU%30
>c0-2
snmp_Load%30
snmp_IO%30
snmp_CPU%30
>c0-3
snmp_Load%30
snmp_IO%30
snmp_CPU%30
```

`*PN_CIT`

defines a new cluster name. This is for a set of processing nodes used by the site. The string ”PN” in the name is necessary if the user wants to automatically use filters to generate global views for all this processing units.

Then it has a list of nodes in the cluster and for each node a list of modules to be used for getting monitoring information from the nodes. For each module a repetition time is defined (%30). This means that each such module is executed once every 30s. Defining the repeating time is optional and the default value is 30s.

2.3.2 Monitoring a Farm using Ganglia gmon module

The configuration file should look like this:

The first line (**`*Master`**) defines a Functional Unit (or Cluster). The Second line (**`>ramen gateway`**) adds a node in this Functional Unit class. In this case ramen is a computer name and optionally the user may add an alias (gateway to this name).

The lines:

```
monProcLoad%30
monProcIO%30
monProcStat%30
```

define three monitoring modules to be used on the node ”ramen”. These measurements are done every 30s. The **`monProc*`** modules are using the local `/proc` files to collect information about the cpu, load and IO.

The line:

```
*PN_popcrn {IGanglia, popcrn01.fnal.gov, 8649}%30
```

Example 2.3.2: Farm configuration with Ganglia gmon

```
*Master
>ramen gateway
monProcLoad%30
monProcIO%30
monProcStat%30

*PN_popcrn {IGanglia, popcrn01.fnal.gov, 8649}

*ABPing{monABPing, ramen.fnal.gov, " "}

*Internet
>tier2.cacr.caltech.edu
monPing%50
```

defines a cluster named "PN_popcrn" for which the entire information is provided by the IGanglia module. This module is using telnet to get an XML based output from the Ganglia gmon. The telnet request will be sent to node **`popcrn01.fnal.gov`** on port 8649.

All the nodes which report to ganglia will be part of this cluster unit and for all of them the parameters selected in the IGanglia module will be recorded. This measurement will de done every 30s.

The Ganglia module is located in the `Service/usr_code/GangliaMod`. The user may edit the file and customize it. This module is NOT in the MonaLISA jar files and for using it the user MUST add the path to this module to the MonaLISA loader. This can be done in `ml.propreties` by adding this line:

```
lia.Monitor.CLASSURLs=file:${MonaLisa_HOME}/Service/usr_code/GangliaMod/
```

The line:

```
*ABPing{monABPing, ramen.fnal.gov, " "}
```

defines a Functional unit named "ABPing" which is using an internal module **`monABPing`**. This module is used to perform simple network measurements using small UDP packages. The first parameter must be the full name of the system which corresponds to the real IP on which the ABping server is running. The second parameter is not used.

The next lines:

```
*Internet
 >tier2.cacr.caltech.edu caltech monPing%50
```

define a new functional unit (Internet) having one node

```
tier2.cacr.caltech.edu
```

with the alias caltech for which a ping measurement is done by the monPing module every 50s.

2.3.3 Monitoring a Farm using Ganglia Multicast module

For getting copies of the monitoring data sent by the nodes running the ganglia demons (using a multicast port) it is necessary that the system on which MonaLISA is running to be in muticast range for these messages.

Adding such a line:

```
*PN_cit{monMcastGanglia, tier2, "GangliaMcastAddress=239.2.11.71; GangliaMcastPort=8649"}
```

in the configuration file, will use the Ganglia multicast module to listen to all the monitoring data and then to select certain values which will be recorded into MonALISA. The service system will automatically create a configuration for all the nodes which report data in this way.

The **PN_cit** is the name of the cluster of processing nodes. Is is important for the cluster name of processing nodes to contain the "PN" string. This is used by farm filters to report global views for the farms.

The **tier2** is the name of the system corresponding to the real IP address on which this MonALISA service is running. The second parameter defines the multicast address and port used by Ganglia.

The GangliaMcat module is located in the `Service/usr_code/GangliaMCAST`. The user may edit the file and customize it. This module is NOT in the MonaLISA jar files and to be used, the user MUST add the path to this module to the MonaLISA loader. This can be done in `ml.propreties` by adding this line:

```
lia.Monitor.CLASSURLs=file:${MonaLisa_HOME}/Service/usr_code/GangliaMCAST/
```

2.3.4 Getting Job related information from PBS

To monitor data provided by the PBS, you will need to add these line into the config file:

```
*JOBS
>tier2   PBSjobs{cmsim,ooHits}%30
```

The first line defines the name of the functional unit (JOBS). The second line defines the node (normally the current system) where the PBSJobs module will run. The module has a parameter containing a list o jobs for which information will be provided. The module will run every 30s. The code of the PBSJobs module is in `Service/usr_code/PBS/` and must be added to the MonaLISA class loader in similar way like the Ganglia modules.

2.3.5 Monitoring Applications

MonALISA monitor external applications using ApMon API. In order to configure MonALISA to listen on UDP port 8884 for incoming datagrams (XDR encoded, using ApMon) you should add the following line in your config file:

```
^monXDRUDP{ListenPort=8884}%30
```

The Clusters, Nodes and Parameters are dynamically created in MonALISA's configuration tree every time a new one is received. It is possible, also, to dynamically remove "unused" Clusters/Nodes/Parameters, if there are no datagrams to match them for a period of time. The timeouts are in seconds:

```
^monXDRUDP{ParamTimeout=10800,NodeTimeout=10800,ClusterTimeout=86400,ListenPort=8884}%30
```

In the example above the parameters and the nodes are automatically removed from ML configuration tree if there are no data received for 3 hours (10800 seconds). The Cluster is removed after one day (24 hours - 86400 seconds).

For further informations how to send data into MonALISA please see ApMon API documentation.

2.4 Database support configuration

The configuration options relevant to the storage are set in the `FARMNAME/ml.properties` file:

```
lia.Monitor.use_emysqldb=true|false
```

this will unpack the embedded mysql (if any)

```
lia.Monitor.use_epgsqldb=true|false
```

for the embedded postgresql

If none of these options is enabled then the following options are relevant for the database server selection:

```
lia.Monitor.jdbcDriverString=
       com.mckoi.JDBCDriver      or
       com.mysql.jdbc.Driver     or
       org.postgresql.Driver
```

McKoi is the default database if nothing else is available, but we don't recommend using it for storing large data structures. If you have a standalone database server you should disable the embedded databases and specify the mysql or postgresql driver here accordingly. The following parameters are the connection parameters for the JDBC driver.

```
lia.Monitor.ServerName=IP_ADDRESS
lia.Monitor.DatabasePort=TCP_PORT
lia.Monitor.DatabaseName=DB_NAME
lia.Monitor.UserName=DB_USERNAME
lia.Monitor.Pass=DB_PASSWORD
```

The actual database structure is determined by the following options:

```
lia.Monitor.Store.TransparentStoreFast.web_writes=N
```

this option specify the number of tables that are used. For each X=0..N-1 you should have:

```
lia.Monitor.Store.TransparentStoreFast.writer_X.total_time=SECONDS
lia.Monitor.Store.TransparentStoreFast.writer_X.table_name=UNIQUE_NAME
lia.Monitor.Store.TransparentStoreFast.writer_X.writemode=MODE
lia.Monitor.Store.TransparentStoreFast.writer_X.samples=SAMPLES
lia.Monitor.Store.TransparentStoreFast.writer_X.descr=UNIQUE_STRING
```

SECONDS specify the time period for which the data is stored in the database. Data older than now()-SECONDS will be automatically deleted.

The "table_name" and "descr" should be unique among the other options of the same kind. "table_name" must be a valid database table name (no spaces and so on), "descr" can be any string you like.

You can store data in either averaged or raw modes. When using and averaged mode the SAMPLES value determine the number of values that are kept for the specified interval. For example if you want to store a single value each minute for an year you should specify SECONDS=31536000 and SAMPLES=SECONDS/60=525600. This is applied separately for each parameter that you store, so such a database can become rather large.

MODE has these possible values:

0: averaged mode the table structure will be

```
rectime | farm | cluster | node | function | mval   | mmin   | mmax
long    | text | text    | text | text     | double | double | double
```

1: raw mode same structure as 0

2: raw mode for storing abstract Object values seldom used

3: averaged mode, data is only kept in memory to control the maximum size of the in-memory buffer use:

```
lia.Monitor.Store.TransparentStoreFast.writer_X.countLimit
```

if set to -1 then only the time limit given by SECONDS is relevant

4: raw mode, in memory, same as 3 but without data averaging

5, 6 : averaged / raw modes for an optimized table structure each `farm/cluster/node/function` combination is given an unique ID, stored in `monitor_ids` table, and the database structure is now:

`rectime | id | mval | mmin | mmax`

this option is the best for large data but with always-changing parameter names (for example netflow data aquisition)

7,8 : averaged / raw modes for another ID-related structure for each unique ID a separate table is kept with the data from that series only, the table name will be `UNIQUE_NAME_id` and the structure is

`rectime | mval | mmin | mmax`

this option is the best one when the data series are constant in time, it works well with up to 10000 table names (10000 unique ids if you have a single table writer, 5000 unique ids if you define 2 separate writers and so on).

> IMPORTANT
>
> Modes 7 and 8 only work with PostgreSQL because of some stored procedures needed to improve response times.

For a large data repository we would recommend using PostgreSQL with something like:

```
lia.Monitor.Store.TransparentStoreFast.web_writes = 2

lia.Monitor.Store.TransparentStoreFast.writer_0.total_time=31536000
lia.Monitor.Store.TransparentStoreFast.writer_0.samples=525600
lia.Monitor.Store.TransparentStoreFast.writer_0.table_name=monitor_1y_1min
lia.Monitor.Store.TransparentStoreFast.writer_0.descr=1y 1min
lia.Monitor.Store.TransparentStoreFast.writer_0.writemode=7

lia.Monitor.Store.TransparentStoreFast.writer_1.total_time=31536000
lia.Monitor.Store.TransparentStoreFast.writer_1.samples=5256
lia.Monitor.Store.TransparentStoreFast.writer_1.table_name=monitor_1y_100min
lia.Monitor.Store.TransparentStoreFast.writer_1.descr=1y 100min
lia.Monitor.Store.TransparentStoreFast.writer_1.writemode=7
```

We define 2 separate writers with different averaging intervals (1min and 100min) so the repository can use the proper one in different situations. For example when plotting a 1-hour chart it will choose the 1min table, but if you plot a 6-months chart it will choose the 100min one, reducing the number of operations needed to plot that data. A single writer would either limit the data resolution or response speed, more than 2 writers add much overhead and supplemental disk usage without much benefit.

Whatever storage type you use there is a memory buffer that is used in parallel with the disk storage (if any). Its size depends on the maximum JVM memory (-Xmx parameter) and is dinamically adjusted so that it doesn't use all of the available memory. When making a history query this is the first source of data, if more data is needed then a separate database query is executed to retrieve the remaining interval. In a repository you can see the current buffer status in `http://...../info.jsp`, look for something like:

```
Data cache: values: 252275/262144 (max 262144), time frame: 2:13:29, served
requests: 16490
```

this tells you the number of values in the buffer, what period of time it holds and how many requests were served from this buffer.

2.5 How to setup the configuration files for your site

- Go to `"MonaLisa"/Service` directory and create a directory for your site (e.g `MySite`). You may copy the configuration files from one of the available site directory (e.g.: those from the `"MonaLisa"/Service/TEST` directory). You must include the following files in you new Farm (`ml.properties`, `db.conf.embedded` and `my_test.conf`)

- Edit the configuration file (`my_site.conf`) to reflect the environment you want to monitor.
- Edit `ml.properties` if you would like to change the Lookup Discovery Services that will be used or if you would like to use another DB System.
- You may add a `myIcon.gif` file with an icon of your organization in `"MonaLisa"/Service/ml_dl`.

The only script used to start/stop/restart "MonaLisa" is `ML_SER` from this directory. After you have done what is described in Section 2.3 section you can start using MonALISA:

```
Service/CMD/ML_SER start
```

2.6 How to start a Monitoring Service from init.d

Please set correctly `MonaLisa_HOME` and `MONALISA_USER` variables from **`${MonaLisa_HOME}/Service/CMD/MLD`**.

For 'Rehat like'

```
#cp ${MonaLisa_HOME}/Service/CMD/MLD /etc/init.d
#chkconfig --add MLD
#chkconfig --level 345 MLD on
```

For Debian

```
#cp ${MonaLisa_HOME}/Service/CMD/MLD /etc/init.d
#update-rc.d MLD start 80 3 4 5 .
#update-rc.d MLD stop 86 3 4 5 .
```

2.7 How to start a Monitoring Service with Autoupdate

This allows to automatically update your Monitoring Service. The cron script will periodically check for updates using a list of URLs. When a new version is published the system will check its digital signature and then will download the new distribution as a set of signed jar files. When this operation is completed the MonALISA service will restart automatically. The dependecies and the configurations related with the service are done in a very similar way like the Web Start technology.

This functionality makes it very easy to maintain and run a MonALISA service. We recomnend to use it!

In this case you should add `"MonaLisa"/Service/CMD/CHECK_UPDATE` to the user's crontab that runs MonALISA. To edit your crontab use: **`$crontab -e`**

Add the following line:

```
*/20 * * * * /<path_to_your_MonaLisa>/Service/CMD/CHECK_UPDATE
```

This would check for update every twenty minutes. It is resonable value that this value should be >= twenty minutes. To check for update every 30 minutes add the following line instead of the one above.

```
*/30 * * * * /<path_to_your_MonaLisa>/Service/CMD/CHECK_UPDATE
```

To disable autoupdate you cand edit the `ml_env` file in `"MonaLisa"/Service/CMD` and set **`SHOULD_UPDATE="false"`**>. It is no need to remove the script `CHECK_UPDATE` from your crontab.

Launch **`"MonaLisa"/Service/CMD/ML_SER start`**. MonALISA should check for updates now.

2.8 Running MonALISA behind a firewall

The MonALISA service has no problem now running behind a firewall because of the new proxy services that connect clients with services. If the proxy service cannot create the TCP connection with the farm service because of a firewall, then the farm starts the connection with the proxy.

2.9 Writing new Monitoring Modules

New Monitoring modules can be easily developed. These modules may use SNMP requests or can simply run any script (locally or on a remote system) to collected the requested values. The mechanism to run these modules under independent threads, to perform the interaction with the operating system or to control a snmp session are inherited from a basic monitoring class. The user basically should only provide the mechanism to collect the values, to parse the output and to generate a result object. It is also required to provide the names of the parameters that are collected by this module.

Creating a new module means writing a class that extends the `lia.Monitor.monitor.cmdExec` class and implements `lia.Monitor.monitor.MonitoringModule` interface.

This interface has the following structure:

```
package lia.Monitor.monitor;

public interface MonitoringModule extends lia.util.DynamicThreadPoll.SchJobInt {

public  MonModuleInfo init( MNode node, String args ) ;
    public String[] ResTypes() ;
    public String   getOsName();
    public Object   doProcess() throws Exception ;

    public MNode getNode();

    public String getClusterName();
    public String getFarmName();

    public boolean isRepetitive() ;

    public String getTaskName();
    public MonModuleInfo getInfo();

}
```

The `doProcess` is actuly the function that collects and returns the results. Usually the return type is a Vector of `lia.Monitor.monitor.Result` objects. It can also be a simple `Result` object.

The `init` function initializes the useful information for the module, like the cluster that contains the monitoring nodes, the farm and the command line parameters for this module. This function is the first called when the farm loads the module.

The `isRepetitive` function tells if the module has to collect results only once or repe-titively. The return values is the `isRepetitive` module boolean variable. If true, then the module is called from time to time. The repetitive time is specified in the <farm> `.conf` file. If not there, then the default repetitive call time is 30s.

The rest of functions returns different module information.

Examples to generate new modules can be found in `${MonaLisa_HOME}/Service/usr_code`.

In `usr_code/MDS` is an example of writing the received values into MDS. This is done using a unix pipe to communicate between the dynamically loadable java module and the script performing the update into the LDAP server.

Another simple example which simply print all the values on sysout can be found on `usr_code/SimpleWriter`.

Another example to write the values into UDP sockets is in `usr_code/UDPWriter`.

2.10 Writing new Filters

Filters allow to dynamically create any new type of derived value from the collected values. Es an example it allow to evaluate the integrated traffic over last n minutes, or the number of nodes for which the load is less than x. Filters may also send an email to a list or SMS messages when predefined complex condition occur. These filters are executed in independent threads and allow any client to register for its output. They may be used to help application to react on certain conditions occur, or to help in presenting global values for large computing facilities.

Chapter 3

MonALISA Features

3.1 Remote Control and Configuration of Applications

3.1.1 General description

MonALISA Service is not just a monitoring tool. Starting with this version, it 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.

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

For each controlled application it must exist a corresponding module. Each module can have multiple instances with **different** configuration files.

When a module is loaded you must specify a **unique** file name which will store the specified modules configuration. After that, you must 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 3.1.3 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.

3.1.2 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|false />
    <section name=ConfigurationSection value=Value line=N read=true|false write=true|fals
   ......
      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" write="true"
      <key name="MIMEMagicFile" value="%2Fetc%2Fapache%2Fmagic" line="507" read="true" wr
    </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*>.

3.1.3 Writing New Modules for AppControl

3.1.3.1 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)`.

3.1.3.2 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.

3.1.4 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.

3.1.5 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 3.1.1.

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.

3.2 Web Services for MonALISA

A simple Web Service is integrated with the MonALISA service, as well with the MonALISA Repository. The Web service, "MLWebService", provides an interface for publishing the monitoring data using a WSDL/SOAP technology. In this way, any client can connect and receive selected monitoring data.

3.2.1 Service description

The service offers a single port type with three operations: `getValues`, `getConfiguration` and `getLatestConfiguration`.

- the `getValues` operation interrogates the database and gets all the parameter values for the farm, the cluster, the node and the parameter name specified as arguments. The time when the parameter(s) was registered in the database must be between two moments of time (fromTime and toTime) also specified as operation input. The return type is a complex schema type, an array of results containing all the values taken from the database.

- the `getConfiguration` operation interrogates the database and gets all configurations of all farms that were registered in the database between two limits of time (from time and to time) given as input. The return type is a complex schema type, an array of configurations that were found in the database that matched the constraints .

- the `getLatestConfiguration` operation interrogates the database and returns the last configuration received in the database for a given farm. It receives as input a string, the farm name. It returns a complex schema type that represents the configuration.

3.2.2 Service implementation

The web service application was developed using Apache Axis (See <http://ws.apache.org/axis>).
The interface of the service contains the following functions:

`Result[] getValues (String farmName, String clusterName, String nodeName, String parameterNam`
This function can be called in two ways:

- specifying negative values for times. For example, if the call is

  `getValues( "*", "*", "*", "*",-3600000,0)`

  the service will return all the values registered in the database in the last hour.

- specifying absolute values for time. For example, if the call is

  `getValues ("*", "*", "*", "*", 1060400000000, 1065000000000)`

  the service will return all the values registered in the database with the registration time between the two values specified in milliseconds.

The Result class is a Bean class and has the following description:

```
public class Result {
    private String farmName;      // the farm name that contains the parameters
    private String clusterName;   // the cluster name that contains the parameters
    private String  nodeName ;    // the node name that contains the parameters
    private String[] param_name ; // the parameters names
    private double[] param;       // the parameters values
    private long time;            // the absolute time in milliseconds when this value wa
                                  // registered in the database
    ............                  // get/set functions
}
```

`WSConf[] getConfiguration (long fromTime, long toTime);` The times specified for this function are absolute moments of time in milliseconds.

The WSConf is a Bean class and has the following description:

```
public class WSConf {
    private WSFarm wsFarm; // the farm that had this configuration
    private long confTime; // the time when this configuration was registered in
                           // the database
    ............           // get/set functions
}
```

the java class that describes a farm:

```
public class WSFarm {
    private String farmName;         // the name of the farm
    private WSCluster[] clusterList; // the clusters of this farm
    ..........                       // get/set functions
}
```

the java class that describes a cluster:

```
public class WSCluster {
    private String clusterName; // the name of the cluster
    private WSNode[] nodeList ; // the nodes contained in this cluster
    ..........                  // get/set functions
}
```

the java class that describes a node:

```
public class WSNode {
    private nodeName ;        // the node name
    private String paramList; // the list of parameters for this node
    .........                 // get/set functions
}
```

`WSConf[] getLatestConfiguration (String farm)` returns the latest configurations received in the database for all farms (farm="*") or returns the latest configuration for a specified farm.

3.2.3 Clients Examples for MLWebService

An archive with Java and Perl examples of simple MLWebService clients example can be downloaded from <http://monalisa.cacr.caltech.edu/>. These examples shows you how to interrogate the web service from MonALISA and get monitoring data using the SOAP protocol.

3.2.3.1 MLWebService clients examples presentation

The client examples presented here can interrogate both the MLWebService from the Repository and the MLWebService from the MonALISA service. There are examples for **Java-Axis**, **WSIF** and **Perl**.

3.2.3.2 Examples archive structure

The sources of the clients examples are located in the `WS-Clients` directory. There are special sub-directories in it (`Java-Axis`, `Perl`, `Wsif`), each containing clients developed using different libraries (**Apache Axis**, **Soap:Lite** and **Apache Wsif**). Every client example calls a function of the MLWebService and is located in a directory having the name of the called function of the service. The source of every example is called **Client** (`Client.java` or `Client.pl`). There are special scripts in each directory for automating the installation of used libraries, the compilation and execution of each client:

- for the examples developed in Java (Axis or WSIF) each example contains the following scripts
  - the `generate_classes` script uses the WSDL2Java tool for generating the client used classes;
  - the `compile_classes` script compiles the client classes;
  - the `run_client` script executes the example.
- for the example developed in Perl, there were used special modules (**Soap::Lite** and **Time:HiRes**). This modules are automatically installed using the `install_soap_lite` and `install_time_hires` scripts located in the Perl directory.

For details see the `Readme` files from every example directory.

Chapter 4

Proxy Service

This service intermediates communication between MonALISA Service and its clients. It registers as a Jini client being, in this way, found by clients. It also finds farms in given lookup services and connects with them. Clients send request messages to the known proxy, which forwards them to the specified farm.

4.1 Why this proxy?

This service was introduced because of the following reasons:

- it limits the number of TCP connections to farms. Without this proxy, every client starts its TCP connection with every found farm. With a big number of clients, a farm could be overloaded. But having a number of proxy services, the number of farm clients is much greater
- the number of messages between farms and clients decreases. For example, without this proxy, every client received from every farm the same filter messages, but on its TCP connection. Using the proxy service, this kind of messages are transmitted only between the farms and the proxy service and then spread by it to all known clients interested in those filters.
- the MonALISA service can now run behind a firewall without any problem. If the proxy cannot connect to the found farm, then the farm initiates the TCP connection with the proxy announcing its presence.

4.2 Where are these proxy services? The communication with clients

These proxy services run on different machines and register with known lookup services. The client finds these services and, getting the proxys attributes, makes a decision on which to choose. After choosing one, the communication with farms is intermediated by this one.

If the connection with the chosen proxy has died, the client tries to find another one and initiates a new dialog with farms through the new one.

Chapter 5

MonALISA Repositories

A Repository is actually a client to farm services that collects data from these services, has dedicated procedures to compress old values and to mediate them and stores the results locally into a relational database (MySQL).

These data are used by the Repository to present a WEB synthetic view of the large distributed system. A servlet engine is used to present historical and real time values in a flexible way. The same mechanism is used to offer access to this information from mobile phones using the Wireless Access Protocol (WAP).

5.1 Currently available repositories

WAP Please set this address into your mobile phone:

<http://monalisa.cacr.caltech.edu:8080/wap/index.wml>

It provides real-time information about:

- **US-CMS** production farms
- Wide Area Network Traffic from **CERN** and **PoP in Chicago**.

WEB <http://monalisa.cacr.caltech.edu:8080/index.html>

Provides information about:

- **US-CMS** production farms: Farms Usage, Masters Load, IO Traffic.
- Wide Area Network Traffic from **CERN** and **Chicago**.

Please see the <http://monalisa.cacr.caltech.edu/> for other available repositories.

Chapter 6

Other Related Topics

6.1 How to test SNMP

You can test the values provided by the snmpd demon (based on how it is configured) by using:

legacy ucd-snmp Use the following command:

```
$ snmpwalk [-p port_no] system_name community OID
```

net-snmp Use the following command:

```
$ snmpwalk -v2c -c community system_name[:port_no] OID
```

Example 6.1.1: SNMP

```
snmpwalk -v2c -c public host_name:161 .1.3.6.1.2.1.2.2.1.10
```

In the previous example the query is performed on the host "host_name", using default settings for snmp (transport = UDP; port = 161; community = public). The output should look like this:

```
IF-MIB::ifInOctets.1 = Counter32: 1430
IF-MIB::ifInOctets.2 = Counter32: 966737519
```

For more information in how to configure and use SNMP: http://www.net-snmp.org/ <http://www.net-snmp.org/>MonALISAprovides snmp modules to collect:

- IO traffic from nodes and network elements
- CPU usage
- System Load
- Disk IO traffic
- etc.

Here are the OIDs that must be "exported" by the snmpd daemon in order to allow various dedicated MonALISA snmp modules to collect the data:

snmp_IO Incoming / outgoing network traffic:

```
IN: .1.3.6.1.2.1.2.2.1.10
OUT: .1.3.6.1.2.1.2.2.1.16
```

snmp_Load Load5, Load10 and Load15:

```
.1.3.6.1.4.1.2021.10.1.3
```

snmp_CPU CPU_usr, CPU_nice and CPU_idle:

```
.1.3.6.1.4.1.2021.11
```

snmp_MEM MEM_free, Swap_MEM_Free

```
.1.3.6.1.4.1.2021.4
```

snmp_Disk FreeDSK, UsedDsk:

```
.1.3.6.1.4.1.2021.9
```