OpenROCS v2.0 - Full Documentation

Josep Sanz, sanz@ice.cat

January 27, 2012

Contents

1 The OpenROCS objective

OpenROCS is the *Open Robotic Observatory Control System* that allow you to control and manage a telescope without human actions. The name is inherited from the first release of the software that implements a design that only uses the needed features for the real case of control a telescope.

The experience adquired by some years demonstrates that an abstract and independent model is more configurable. When appear new requirements or needs, as install a new instrument, sensor or change the workflow of the entire system, then a system that allow quickly define the new interfaces to the new hardware or allow to change the decision tree easily is more efficient and robust.

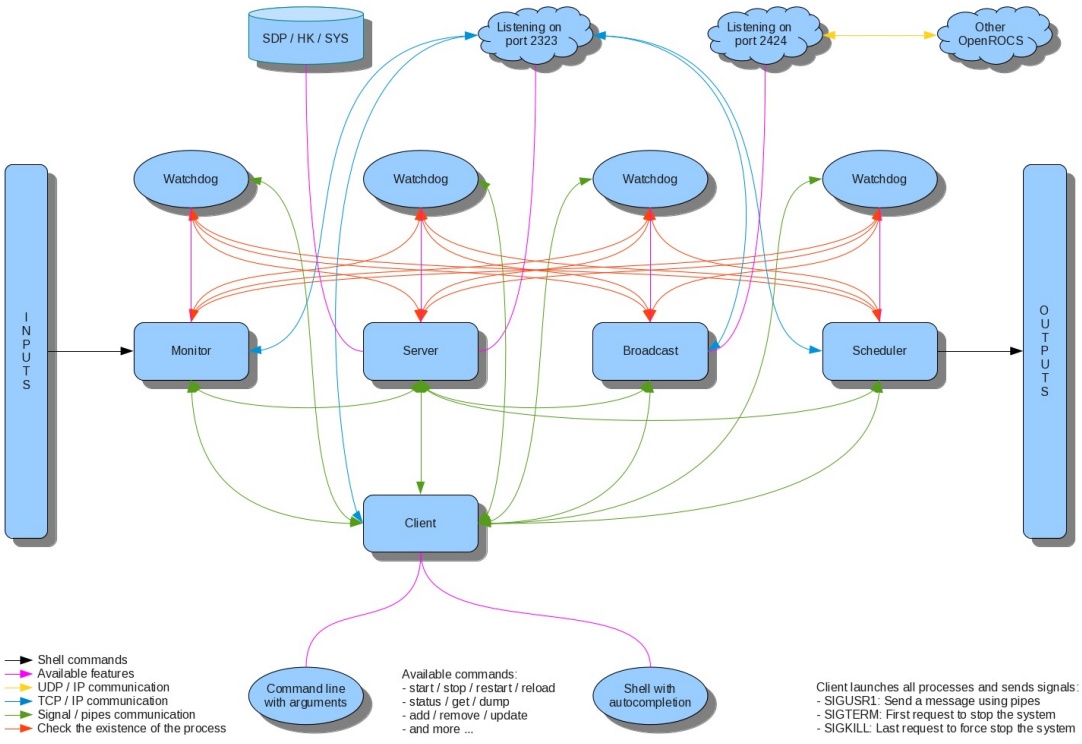
To achieve the objective, we design an abstract model that define all needed features as storage data, monitor tasks and the scheduler actions that executes commands without enclose with an specific hardware or software.

As conclusion, you can understand that OpenROCS v2.0 allow to control "hardware from and to hardware". You can use shell commands as inputs of the system, configure using the xml, the specific logic to take decisions and then, execute the shell commands to actuate in the desired hardward.

2 Architecture overview

To understand the basis used by OpenROCS, you need see the full diagram that contain all relations between processes, pipes, signals, network communications and the inputs/outputs.

OpenROCS v2.0 Block Diagram



The system is organized as:

- 4 main services:
  - Server: that attend the requests using a typical TCP/IP socket and provides a remote storage to retrieve data using total or incremental timestamps.
  - Broadcast: syncronize multiples OpenROCS distributed on a network using broadcast to discover they.

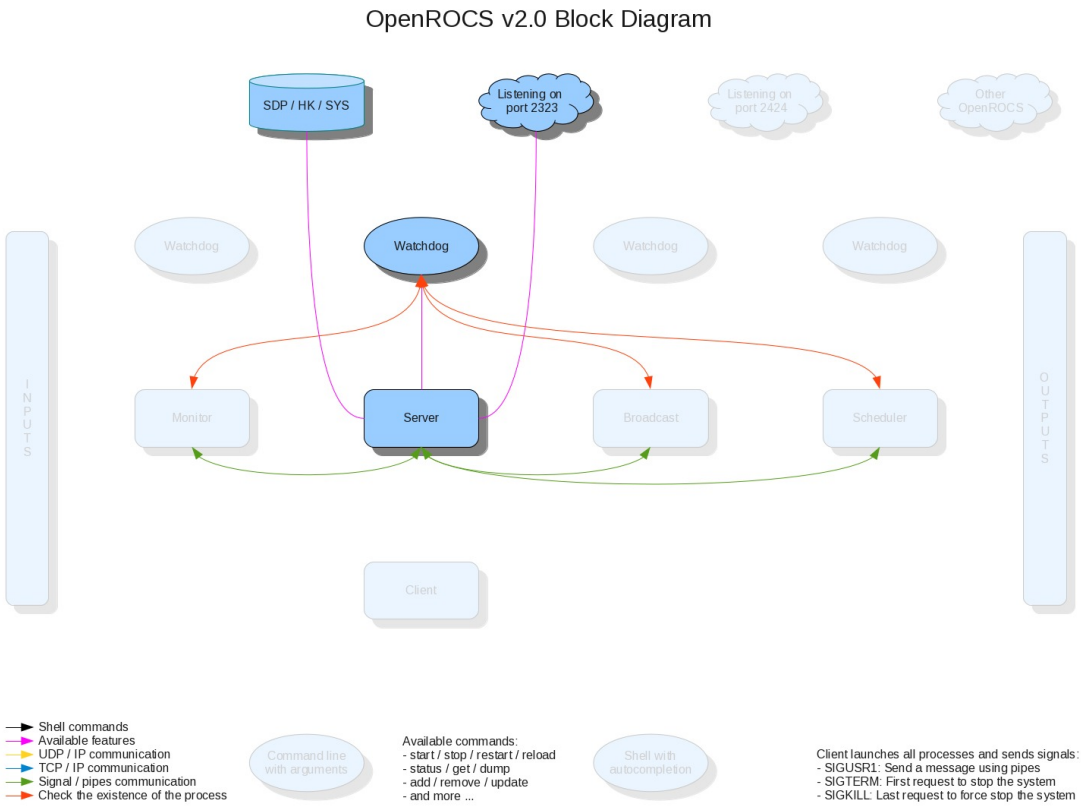
- Monitor: the main task of this service is maintain updated the SDP and HK to be used by the scheduler.
- Scheduler: allow to execute actions that control the entire telescope when the SDP or HK changes.

- A client to manage the services: allow to control the services (start/stop/restart/reload) and provides two modes of use (command line with arguments or shell with autocompletion).

3 Modules design

All services started by OpenROCS use some portions of code that share features as the watchdog, the process communication and the process management. This features are grouped to do a robust system that checks all time the state of the entire system. In other words: the server verifies that the other involved processes are running as expected, the monitor does the same things and the scheduler, too does the same things. With this design, the redundant watchdogs controls that all system are working as expected. If some part of the system fails, it's more reasonable stop all instead of continue running with a part that fails.

3.1 The Server service



This service has the follow objectives:

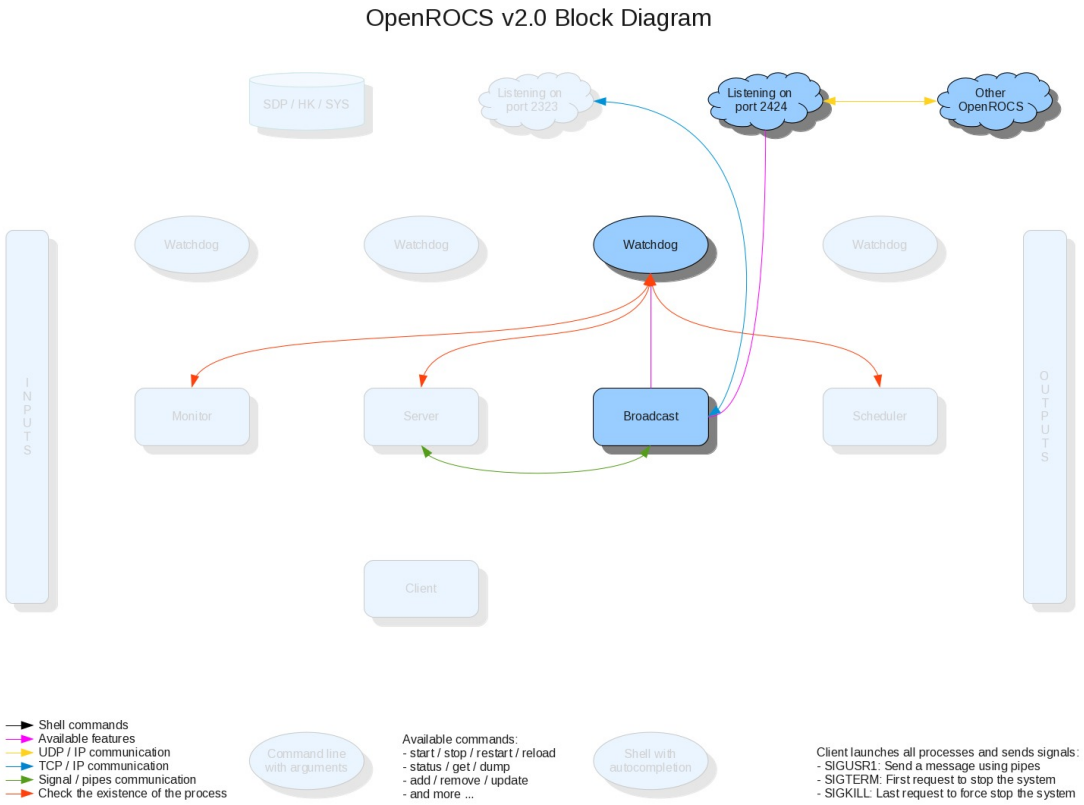
- Execute a TCP/IP server on port 2323 to handle requests.
- Provide a remote storage.
- Provide a language to manage stacks, variables and their values.

The main idea of this service is centralize all information of the entire system and allow the *clients* of this service to set, update and retrieve the data stored in the stacks. Exists a lot of commands that allow clients retrieve all variables of a stack or only retrieve the updated variables from a timestamp allowing the system to

do more efficient and prevent the full bandwidth ocupation. The commands that the server can understand are explained in the *client tool* section.

Too, this service allow to communicate directly with the processes using signals and pipes. From the commands supported by the server, is possible to pause and continue the execution of some service, useful to prevent concurrence problems when, for example, a scheduler that needs to update the same variable that a monitor, allowing the scheduler to: stop the monitor, modify the variable with an exclusive access, and when finish the task, start the monitor.

3.2 The Broadcast service

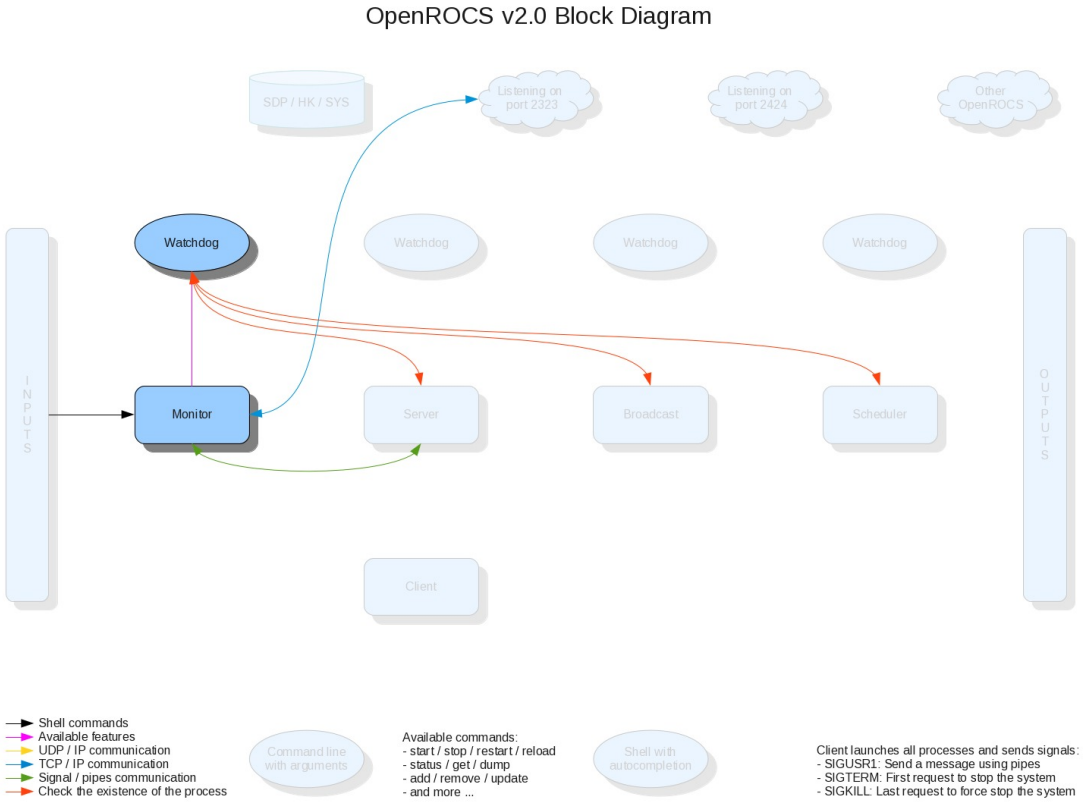


This service has the follow objectives:

- Execute a UDP/IP server on port 2424 to handle requests.
- Send the discovery packets to the broadcast.
- Update the stacks located on server with the new information

The idea of this service is to allow OpenROCS to discovery another instances of OpenROCS running in the same network and interchange the stacks periodicaly. This is useful when need to use variables defined by another OpenROCS instances.

3.3 The Monitor service

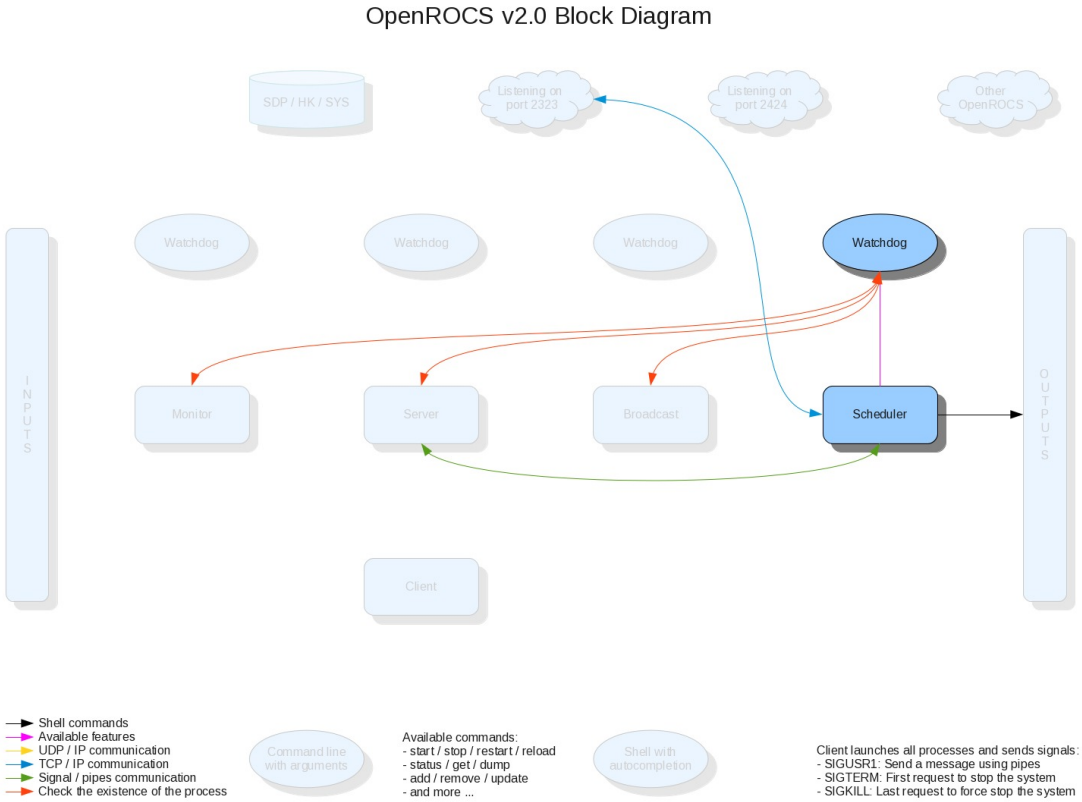


This service has the follow objectives:

- Execute periodicaly the configured tasks and depending of the output (stdout or stderr), add, remove or update with the desired value the different stacks (SDP/HK/SYS).
- Configure some tasks executed between an interval or frecuency: the difference between interval and frequency is that interval is the time between the end of the task and the begin of the same task and frequency execute "more" periodicaly the same task.
- Control of the tasks using timeouts and programing actions when "ontimeout" is reached.
- Multiple evaluation control of the output generated by the commands. To do this feature, we use the choose/when/otherwise language structure.
- Communication with the Server service to set data (the expected result of this service)

The idea of this service is allow the system to execute periodicaly tasks reporting the outputs as updates in the variables stacks. It's useful, for example, to monitorize the status of the weather stations, the PDU's outlets, daemon status or other requirements. When the system startup, all schedulers are stop until all monitors executes, as minimun, one time all tasks. This is to prevent an erroneous schedulers executions (you must understand that monitors must initialize the entire system and set the first value in all needed variables to a correct schedulers executions). The initial status of the monitors is *init* until all tasks of each monitor are executed, as minimum, one time.

3.4 The Scheduler service

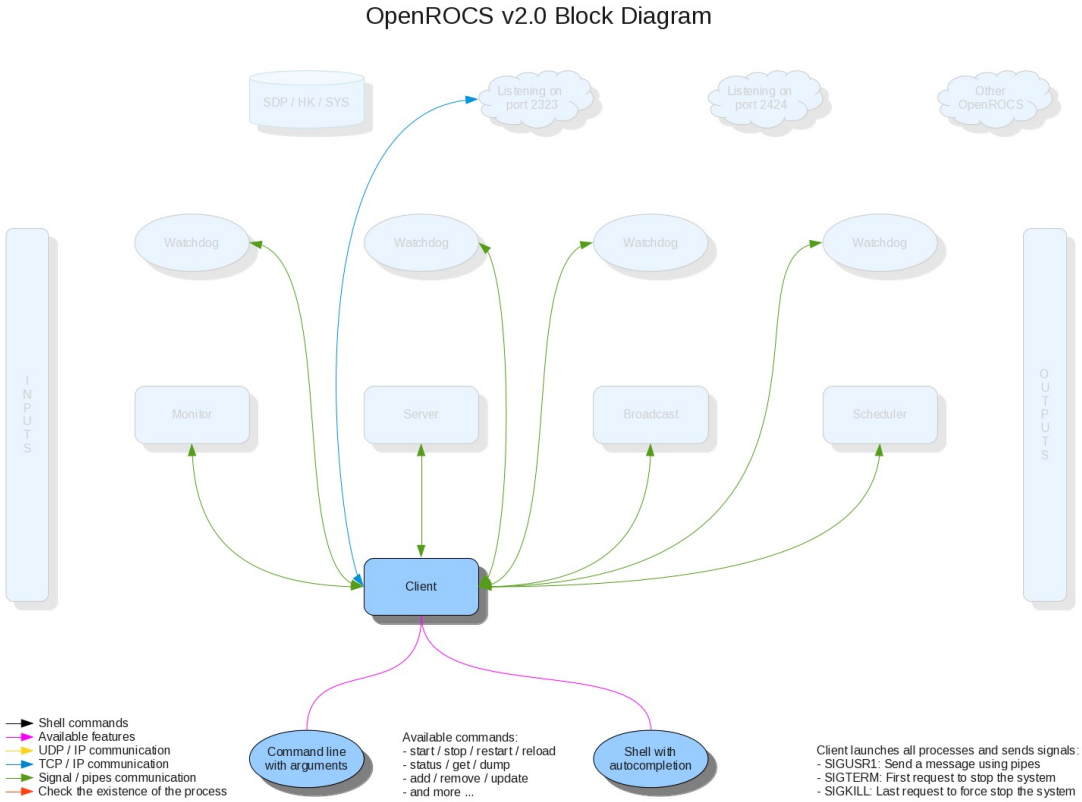


This service has the follow objectives:

- Offer a monitor service that can be trigger when detect changes in some variables of the SDP/HK/SYS (remember that monitor service is executed periodicaly without external trigger).
- This service can define more that one scheduler execution flow and their variables (the hash concept).
- Gets and sets data from and to the Server service. To be more efficient with the getting data action, we use the incremental feature, that only reports the stacks and variables that has been added, updated or removed using a timestamp as control.

The idea of this service is allow the system to take decisions depending of the values of the variables of the stacks. Each scheduler must defines the *hash variables* that is the list of variables that can trigger the execution of the scheduler. As is explained in the *monitor service*, the scheduler must waits until all monitors sets the needed variables to prevent erroneous executions. For this reason, the initial status of schedulers are *stop* waiting that all monitors executes all tasks as minimum one time.

3.5 The Client tool



The Client tool was designed to allow the actors of the telescope modify the running scheduler, adding, updating and/or remove data from the SDP/HK/SYS. This client can work in 2 modes:

- Command line with arguments: allow to execute the OpenROCS command with arguments that are executed in synchronous (blocking mode).
- Interactive shell with autocompletion: allow the human interaction using a typical shell with autocompletion (like bash, for example).

The two modes, has the same commands and features and the idea of provide these two methods of client work is for example, integrate with another software or shell scripts and provide an easy interaction tool.

Available single commands:

- shell: open an interactive shell (default if not command is present)
- help: display this screen
- start: launch the main server process
- restart: restart the main server process
- reload: same as restart but without losing data
- stop: stop the main server process
- status: check the server state
- dump: dump all stacks and their keys and values
- check: run a quick validation test
- history: display the command history (only exists in the interactive shell)

Available commands that need parameters:

- get [none]:return a list with all stacks
- get STACK: return a list with all data in the STACK
- get STACK KEY: return the value of the KEY in the STACK
- get TIMESTAMP: return a list with the modified stacks from TIMESTAMP
- get STACK TIMESTAMP: return a list with the modified data from TIMESTAMP in the STACK
- add/create STACK: create the STACK if not exists
- add/create STACK KEY: add the KEY to the STACK
- add/create STACK KEY=VALUE: add the KEY with their VALUE to the STACK
- update/set STACK KEY=VALUE: update in the STACK the KEY with the new VALUE
- remove/delete STACK: remove the STACK only if exists and is void
- remove/delete STACK KEY: remove the KEY in the STACK if exists
- stop SERVICE: pause the service requested
- status SERVICE: check the status of the service requested
- start SERVICE: continue the service requested

Notes:

- The start and stop commands are only availables from the command line (using arguments or using the interactive shell). If you try to send some of these commands directly to the server, the response will be *permission denied*.
- Other commands as shell, help, dump, check and history, are only availables from the orocs client. If you try to send some of these commands directly to the server, the response will be *unknown command*.

4 System interfaces

The OpenROCS, uses various techniques to communicate the processes (Server, Broadcast, Monitor, Scheduler and Client) and access to the storage service (Server) to set and retrieve SDP/HK data.

4.1 Signals and pipes

This communication technique is used by three ways:

- From the start command and it's used to communicate the client with the created childrens and get the output generated (the console messages).
- By the watchdog subsystem that transfers the entire childrens list and checks the existence of the other implied process.
- By the server to start and stop the other subsystems process.

4.2 TCP/IP sockets

This communication technology is the common feature used by the rest of implied subsystems:

- Server executes a server to handle all requests of port 2323 and respond with the requested response.
- Client set and get data from and to the SDP/HK/SYS and CHLD (childrens launched by OpenROCS as services).
- Monitor sends (and too can retrieve) all tasks results to the storage server (SDP/HK/SYS).
- Scheduler retrieves and send information of SDP/HK/SYS.
- Broadcast uses this channel to update the stacks with the data provided by another OpenROCS.

4.3 UDP/IP sockets

This communication technology is used by the broadcast service:

- Broadcast executes a server to handle all requests of port 2424 and respond with the requested response.
- Too, sends periodicaly packets to the broadcast address to discover other OpenROCS running in the same network.

4.4 Shell commands

In the OpenROCS's objectives, we explain that the design defines an abstract model to be independent of the hardware and the software of telescope. To allow this abstract level, OpenROCS don't implements specific program code to connect, communicate and actuate with the telescope and other elements as environment systems (wheather station, for example). The unique method to communicate with the external system to retrieve inputs and actuate with output is the command line shell.

This concept allow that OpenROCS uses external tools as:

- SNMP to get or set parameters to devices (PDU's, UPS's, Wheather stations and a lot of other hardware that can communicate using this standar and extended protocol) and OpenROCS, uses it using an external command (snmpget and snmpset)
- INDI clients to communicate and controls the telescope.
- Another scripts that offers similars issues as INDI clients to wrap commands, for example, TALON when writes and reads directly to and from the fifos (named pipes).

As you can see, OpenROCS offers useful methods to get and set data with the rest of the system. All commands can use the stdout and stderr to generate the requested output. OpenROCS allow to evaluate the output (using the stdout and stderr channels) using a intuitive language (PHP by default).

5 System configuration

The system uses XML to define all needed variables and data's structures and it's important to use correctly the XML syntax. Some cases, can be needed to use tricks to accomplish the standard XML specification:

- If the value of a node contain the characters "&" or "<", the entire value must be enclosed using a CDATA clausule:

```
1  <eval><![CDATA[$OUTPUT>=-200 && $OUTPUT<=200]]></eval>
```

- Without the CDATA clausule, the parser makes an error when read the node:

```
1  <eval>$OUTPUT>=-200 && $OUTPUT<=200</eval>
```

- This is the XML format that all files must accomplish to be parsed correctly

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <root>
3      *
4      *
5      *
6  </root>
```

5.1 File config.xml

This file is used by all system services and client. Defines the server configuration and stacks, debug flags, shell features, sets some variable values (php.ini and system environment).

5.1.1 Node <server>

It defines the host and port used by the server service and create the stacks at start up.

```
1  <server>
2      <host>127.0.0.1</host>
3      <port>2323</port>
4      <name>myOpenROCS</name>
5      <stacks>
6          <stack>HK</stack>
7          <stack>SDP</stack>
8          <stack>SYS</stack>
9      </stacks>
10 </server>
```

Notes:

- The system creates by default (and mandatory to the internal processes control) only 1 stack (CHLD) to store all information related to the services used by OpenROCS. This structure is a base64 string that contain a serialized array with the follow information: PID, HOSTNAME, NODE and NAME.
- The <name> node allow to set an human name useful when need to use the *broadcast service*.

5.1.2 Node <broadcast>

It configures the broadcast parameters used by OpenROCS to execute this service.

```
1  <broadcast>
2      <enabled>true</enabled>
3      <port>2424</port>
4      <discovery>60</discovery>
5      <synchronize>10</synchronize>
6  </broadcast>
```

Notes:

- You can turn off this service if you set a false value to the <enabled> node .
- The <discovery> node sets the interval used by the broadcast service to send packets to the broadcast address.
- The <synchronize> node set the interval used by the broadcast to retrieve and update the stacks between the discovered OpenROCS and the local OpenROCS.

5.1.3 Node <debug>

It defines the debug flags used by OpenROCS. It's used to add more or less debug data to the log files.

```
<debug>
    <comm>false</comm>
    <signal>false</signal>
    <process>false</process>
    <trace>true</trace>
    <maxlines>1000</maxlines>
    <percent>50</percent>
</debug>
```

Notes:

- The <maxlines> node sets the maximum number of lines that can be stored in each file. If the file contain more lines, then the rotate log moves the old file and creates a new file (as the UNIX rotate logs system). This parameter apply to all log files that system uses.
- The <percent> node sets the maximum allowed timeout reached before that the systems reports a complete trace for debug purposes into the *debug.log* file. Useful to check the network quality.
- The other nodes can enable or disable the log features of each module (recomended only for testing purposes). The debug system writes the outputs to the stdout channel.
- "the UNIX rotate logs system" allow to rename the log file periodicaly to leave it as an old file and create a new file. This technique prevent that exists big files unusable for the daily work.

5.1.4 Node <shell>

It only defines the path where the interactive shell saves and retrieves the history of commands used.

```
<shell>
    <history>$HOME/.openrocs_history</history>
    <maxlines>1000</maxlines>
</shell>
```

Notes:

- The <maxlines> node sets the maximum number of entries in the history. If the number of entries is higher that the specified value, then saves the last <maxlines> commands.

5.1.5 Node <timeout>

This node defines the timeouts used internally by OpenROCS in the pollings process.

```
<timeout>
    <comm>1000000</comm>
    <childs>1000000</childs>
    <pipes>3000000</pipes>
    <wait>5000000</wait>
    <server>5000000</server>
    <semaphore>5000000</semaphore>
</timeout>
```

Important notice:

- It is highly recommended to leave these values because they should be consistent among all.

5.1.6 Node <polling>

This node defines the sleeps times used internally by OpenROCS in the pollings process.

```
<polling>
    <comm>1000</comm>
    <childs>1000</childs>
    <pipes>1000</pipes>
    <wait>100000</wait>
    <server>100000</server>
    <monitor>100000</monitor>
    <scheduler>100000</scheduler>
    <broadcast>100000</broadcast>
</polling>
```

Important notice:

- It is highly recommended to leave these values because they should be consistent among all.

5.1.7 Node <retries>

This node defines the retries used internally by OpenROCS when the pollings process fails.

```
<retries>
    <comm>5</comm>
    <childs>5</childs>
    <pipes>5</pipes>
</retries>
```

Important notice:

- It is highly recommended to leave these values because they should be consistent among all.

5.1.8 Node <ini_set>

This node is for internal use only. The user don't need to modify these values because are configured by the system administrator according to the computer profile.

```
<ini_set>
    <session.bug_compat_42>On</session.bug_compat_42>
    <register_globals>Off</register_globals>
    <memory_limit>128M</memory_limit>
    <max_execution_time>0</max_execution_time>
    <date.timezone>UTC</date.timezone>
    <default_charset>UTF-8</default_charset>
</ini_set>
```

Important notice:

- It is highly recommended to leave these values because they should be consistent among all.

5.1.9 Node <putenv>

It allows to setup some environment variables. This node is important because the PATH or LANG used by the shell determines the correct execution of the commands programmed in the monitor and scheudler subsystem.

```
<putenv>
    <PATH>/bin:/usr/bin:/usr/local/bin</PATH>
    <LANG>en_US.UTF-8</LANG>
</putenv>
```

Important notice:

- It is highly recommended to leave these values because they should be consistent among all.

5.2 File variables.xml

This file allows to define useful variables used internally in the XML files (monitor.xml and scheduler.xml). The tree allows multiple levels and the resultant variable is the concatenation of all node names with the "understand" character (_). For example, to use the HOME IP, you must use the variable $IP_HOME.

```
<variables>
    <IP>
        <SERVER>192.168.0.10</SERVER>
        <SENSORS>192.168.0.11</SENSORS>
        <TELESCOPE>192.168.0.12</TELESCOPE>
        <DOME>192.168.0.13</DOME>
        <CAMERA>192.168.0.14</CAMERA>
        <POWER>192.168.0.15</POWER>
    </IP>
    <INTERVAL>10</INTERVAL>
    <PDU>
        <OID>.1.3.6.1.4.1.318.1.1.12.3.3.1.1.4</OID>
        <ON>1</ON>
        <OFF>2</OFF>
    </PDU>
</variables>
```

5.3 File monitor.xml

This file defines the tasks executed by the monitor. The number of maximum tasks depends of the memory and cpu capacity of the computer that executes the monitor. By definition, you can understand that the maximum number of tasks is unlimited. The follow example shows the supported scheme.

```
<monitor>
    <name>mymonitor1</name>
    <task>
        <interval>10</interval>
        *
        *   Process nodes
        *
    </task>
    <task>
        <frequency>10</frequency>
        *
        *   Process nodes
        *
    </task>
    <task>
        <interval>10</interval>
        *
        *   Process nodes
        *
    </task>
</monitor>
<monitor>
    <name>mymonitor2</name>
    <task>
        <frequency>10</frequency>
        *
        *   Process nodes
        *
    </task>
    <task>
        <interval>10</interval>
        *
        *   Process nodes
        *
    </task>
    <task>
        <frequency>10</frequency>
        *
        *   Process nodes
        *
    </task>
</monitor>
```

Notes:

- <interval> or <frequency>: The trigger used to executed each task:

- <interval>: defines the time elapsed between the end of the last task and the begin of the same task. For example: if a shell command consumes 1 second and defines the interval as 10 seconds, then the real execution frequency is 11 seconds.
- <frequency>: Allows to force a precise frequency of execution. It's independent of the time consumed by the shell command. For a safe use of this feature, it's recomendable, the use of a <timeout> tag to prevent an overlap of the execution.

- You can group tasks using the <monitor> node. Each <monitor> node is executed in a separated process that can be suspended and resumed.
- You can specify the <name> node in each <monitor> node. If the node <name> is not present, it uses a number to identify it.
- The contents of each <task> must be conform the process parser specification explained in the *Process nodes* section.

5.4 File scheduler.xml

This file defines the tasks executed by the scheduler. Note that the scheduler service have the same options that monitor service and the unique difference is that the scheduler uses as trigger the hash of some defined variables.

```
1  <scheduler>
2      <name>myscheduler1</name>
3      <hash>
4          <variable>myvariable1</variable>
5          <variable>myvariable2</variable>
6          <variable>myvariable3</variable>
7      </hash>
8      *
9      *   Process nodes
10     *
11 </scheduler>
12 <scheduler>
13     <name>myscheduler2</name>
14     <hash>
15         <variable>myvariable4</variable>
16         <variable>myvariable5</variable>
17         <variable>myvariable6</variable>
18     </hash>
19     *
20     *   Process nodes
21     *
22 </scheduler>
```

Where:

- The <scheduler> node defines a scheduler task.
- Each <scheduler> node requires the <hash> node that defines the variables used to calculate the hash (used as trigger).
- Each <scheduler> node is executed in a separated process that can be suspended and resumed.
- You can specify the <name> of the <scheduler>. If the node <name> is not present, it uses a number to identify it.
- The contents of each <action> and <scheduler> must be conform the process parser specification explained in the *The process nodes* section.

6 The process nodes

The process nodes allow to specify what commands, evaluations and flow control must use the monitor and scheduler to accomplish the objective. In reality, the process nodes are the real language used by OpenROCS to "program" the required needs. With these nodes, you can setup a "program" that executes commands, parses the outputs and then, executes another codes depending the evaluations. The flow defined using tags as <choose>.

6.1 Node <action>

This node allows to execute defined actions. The actions can be defined in multiple xml files and allow define, for example, functions, procedures and/or macros that can be called from more that one point of the process nodes.

The general specification about actions is:

```
<actions>
    <action>
        <name>myaction1</name>
        *
        *   Process nodes
        *
    </action>
    <action>
        <name>myaction2</name>
        *
        *   Process nodes
        *
    </action>
</actions>
```

To call some action of this file, you must to use the follow notation:

```
<action>action_file.xml[action_name]</action>
```

If you prefer to separate all actions into single files, you can create a single xml file that only contains the process nodes. To execute some single action, only need to specify the xml filename without specify the action_name:

```
<action>action_file.xml</action>
```

6.2 Node <shell>

The tipically command line that can be executed directly into a shell. It uses the default shell of the system (non root user recommended and GNU/Linux commonly uses /bin/bash as default shell for all users of the system).

```
<shell>ping -c 3 -i 1 -W 1 $IP_SERVER | grep received</shell>
```

Notes:

- This node can use variables defined into the xml/variables.xml file.
- The default shell used to executes commands is the /bin/sh. You must known that the shell used is important to prevent errors with the expected environment variables.
- All shell commands are launched in background mode to allow OpenROCS to control the execution of the process and check the elapsed time used by the process to apply the correct rules if the *timeout node* is specified.

6.2.1 Node <timeout>

Defines the maximum time allowed to the command line execution. You can specify the maximum amount of time desired to the execution of the command shell and when the elapsed time arrive, then the execution is aborted (it sends a SIGTERM and SIGKILL to stop the process).

```
<timeout>5</timeout>
```

Notes:

- This tag not requires the <ontimeout> tag. Can works separately.
- This tag launches a routine that kill the process launched and all childrens processes to prevent zombies processes, garbage processes or unused processes.
- More information about *zombies processes* at http://en.wikipedia.org/wiki/Zombie_process

6.2.2 Node <ontimeout>

In conjunction with <timeout> tag, defines the action to do when the timeout time is reached.

```
1  <ontimeout>add HK_INTERNET_SERVER_FAIL</ontimeout>
```

Notes:

- This tag requires the <timeout> tag. If <timeout> tag is not present, the systems report an error.

6.3 Node <php>

Another option to execute PHP code directly and retrieve the STDOUT and STDERR to be used after. See the next example to understand it:

```
1  <php>microtime(true)</php>
```

And a complete example for a periodic task:

```
1  <task>
2      <php>microtime(true)</php>
3      <frequency>1</frequency>
4      <send>update SDP_PHP_MICROTIME=$STDOUT</send>
5  </task>
```

Notes:

- This node can use variables defined into the xml/variables.xml file.

6.4 Node <choose>

Defines a control statement. This feature, allows to specify complexes trees that can implement all needed requirements. The <choose> <when> <otherwise> structure is inspired in the XSLT language, that allow to specify using this simple concept all decisions tree, as the tipical *if then else* used in the more commonly programming languages. The follow example shows the supported scheme.

```
1  <choose>
2      <when>
3          <eval>$STDOUT4>0</eval>
4          <send>remove HK INTERNET_SERVER_FAIL</send>
5          <fromiter>3</fromiter> or <fromsec>30</fromsec>
6      </when>
7      <otherwise>
8          <send>add HK INTERNET_SERVER_FAIL</send>
9      </otherwise>
10 </choose>
```

6.4.1 Node <when>

This tag executes all contents of the tag when the evaluation of the <eval> tag has a true value.

Node <eval>

This tags sets true or false to the <when> node that contain it. The contents of this tag must be a PHP expression and can use all available variables:

- $STDOUT: contain the output string of the last <shell> execution (commonly named standar output pipe or channel).
- $STDERR: contain the error string of the last <shell> execution (commonly named standar error pipe or channel).
- Variables defined in variables.xml file (see the variables.xml in system configuration section).
- Variables availables in the server (SDP/HK/SYS). For example, $HK_INTERNET_SERVER_FAIL.

Notes:

- The STDOUT and STDERR variables are splitted and allows the use of portions of the contents using a number to identify.

  For example:

```
1  <shell>ls -l orocs</shell>
```

  It will generate the next output (if the execution has not problems):

```
1  -rwxrwxr-x 1 sanz sanz 707 sep 11 18:41 orocs
```

  In this case, you can use:

```
1  <eval>'$OUTPUT9'=='orocs'</eval>
```

  The evaluation results is true and the <when> that contain this <eval> tag, will be executed.

Node <fromiter>

This node allow to execute tasks with some delay, using as unit, the iteration of execution. This allow, for example, postpone the execution of some <when> node some iterations. To clarify the idea of this node, see the follow lines:

```
1  <fromiter>3</fromiter>
```

And the result of execution is:

1	2	3	4	5	6	7	...
not	not	run	run	run	run	run	...

Node <everyiter>

Continuing with the concept of delayed executions, see the example for this node is:

```
1  <everyiter>3</everyiter>
```

And the result of execution is:

1	2	3	4	5	6	7	...
run	not	not	run	not	not	run	...

Notes:

- Imagine that you need run every 3 iterations some code, but: do you want to begin at the 2 iteration?. To do this case you need to use the follow nodes:

```
1  <fromiter>2</fromiter>
2  <everyiter>3</everyiter>
```

And the result of execution is:

1	2	3	4	5	6	7	...
not	run	not	not	run	not	not	...

Node <untiliter>

The example for this node is:

```
1  <untiliter>3</untiliter>
```

And the result of execution is:

1	2	3	4	5	6	7	...
run	run	run	not	not	not	not	...

Node <fromsec>

This node uses the same concept that <fromiter>. Instead of use the iterations of the execution as unit to count, here, the system uses the timestamp to control the time elapsed from the first execution and trigger correctly the event to begin the execution. For example, with this tag, if you want to begin the execution when a value is stable during 60 seconds, the node to use is:

```
1  <fromsec>60</fromsec>
```

Node <everysec>

See the <fromsec> and <everyiter> to understand this node.

Node <untilsec>

See the <fromsec> and <untiliter> to understand this node.

6.4.2 Node <otherwise>

The behavior of <otherwise> node is the same that the <when> node and only is executed if all previously <when> evaluates as false.

6.5 Node <send>

This tag defines a direct channel communication with the storage server:

- Can add, update and remove stacks and variables (as documented in *The Client tool* section).
- See the *The Client tool* section to understand all options availables from here.

An example of use for this node is:

```
1  <send>add HK_INTERNET_SERVER_FAIL</send>
2  <send>update SDP_TEMPERATURE_DAVIS_INDOR=$STDOUT1</send>
3  <send>remove HK_INTERNET_SERVER_FAIL</send>
```

6.6 Node <log>

The <log> node allows the user to define in the process nodes, what messages want that appear in the user.log file. This node is useful for personalize the debug process of the xml operator that writes the flow.

An example of use for this node is:

```
1  <log>System running correctly</log>
```

7 Source code organization

The source code uses the follow structure of directories:

- root: This directory contains all software needed to run OpenROCS and the main program used in all requests (the OpenROCS main script)
  - xml: Here, the software try to load the configuration for each service.
    - HOSTNAME: if exists a directory with the same name that the hostname of the computer that is executing OpenROCS, then the directory is used to search the needed xml files and if not found the needed files, as last option, returns to the original xml directory to get the desired file.
  - php: This directory contain all scripts used to implement the OpenROCS system. Too, contain a directory named action that contain specific scripts that executes some actions as start, restart, reload and stop, help and shell, ...
  - log: This directory is used by OpenROCS to store the activity register (logs) used to check the correct work and for debug purposes.

8 Programming language

The language used to code this software is *PHP*, a scripting widespread language with a lot of years of testing and supported by the community. It's similar to Python or Perl, but uses a commonly syntax as *C*.

As an example of the power of PHP, OpenROCS uses features as:

- pcntl_fork and pcntl_signal: allow to OpenROCS to start scripts in background mode and communicates its as a *C* program.
- register_tick_function and unregister_tick_function: allow to OpenROCS to program a periodical task (used as watchdog control system).
- socket_create, socket_bind, socket_listen, socket_select and socket_accept (used to publish a network service).
- register_shutdown_function: it allow to OpenROCS to program a shutdown task (used to close sockets and execute the garbage collector).
- set_error_handler and set_exception_handler: allow to OpenROCS to program the error routines used to prevent malfunctions.

Too, *PHP* provides a lot of features for manage arrays, matrixes and trees, parse strings and xml files, use of variables by reference, the variable variable feature and more things that OpenROCS explodes to accomplish all needed tasks.

You can see more information at

- PHP project site: http://www.php.net/
- PHP documentation site: http://www.php.net/manual/en/

9 Acknowledgments

OpenROCS v2.0 is a software developed by Josep Sanz, but a lot of people has been contributed with experience, know how and ideas with the objective of design a scalable software and daily usable.

I want to thanks the help and the feed back of all team that works with me:

- Francesc Vilardell: the XML configurator, and consequently debugging.
- Ignasi Ribas and Pep Colome: by the main requirements.
- Pere Gil: by the user experience.

10 License

OpenROCS v2.0 is released under the GPL-3.0 license:

```
/*
  ___                   ____   ___   ____ ____          ____    ___
 / _ \ _ __   ___ _ __ |  _ \ / _ \ / ___/ ___|  __   _|___ \  / _ \
| | | | '_ \ / _ \ '_ \| |_) | | | | |   \___ \  \ \ / / __) || | | |
| |_| | |_) |  __/ | | |  _ <| |_| | |___ ___) |  \ V / / __/ | |_| |
 \___/| .__/ \___|_| |_|_| \_\\___/ \____|____/    \_/ |_____(_)___/
      |_|

OpenROCS: Open Robotic Observatory Control System
Copyright (C) 2011 by Institut d'Estudis Espacials de Catalunya (IEEC)
More information in http://www.ieec.cat or ieec@ieec.cat

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
```

OpenROCS v2.0 uses portions of code from the SaltOS project (released under the GPL-3.0 license too) that provides the XML parsers, error management, semaphores management and more other functionalities:

```
/*
  ____        _ _    ___  ____
 / ___|  __ _| | |_ / _ \/ ___|
 \___ \ / _` | | __| | | \___ \
  ___) | (_| | | |_| |_| |___) |
 |____/ \__,_|_|\__|\___/|____/

SaltOS: Framework to develop Rich Internet Applications
Copyright (C) 2011 by Josep Sanz Campderros
More information in http://www.saltos.net or info@saltos.net

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
```