

JASMINe Probe

User's guide

v1.1

JASMINe Probe: User's guide

JASMINe Team

Publication date \$Id: jasmine-probe_guide.xml 9781 2012-02-07 12:45:56Z danesa \$

Copyright © 2011 Bull SAS



This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.0/deed.en> [<http://creativecommons.org/licenses/by/2.0/>] or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Table of Contents

1. Introduction	1
2. How to use JASMINe Probe	2
2.1. Where to install JASMINe Probe	2
2.2. Installing JASMINe Probe	2
2.3. Running JASMINe Probe in <i>standalone</i> mode	3
3. Definitions	4
3.1. Main Concepts	4
3.2. Targets	4
3.3. Indicators	4
3.4. Outputs	5
3.5. Probes	5
4. JASMINe Probe artifacts creation	6
4.1. Target Definition	6
4.2. Indicator definition	6
4.2.1. Create an indicator	6
4.2.2. Define a jmx indicator	7
4.2.3. Define a df indicator	9
4.2.4. Define a lewys indicator	9
4.2.5. Define an aggregate (or merge) indicator	9
4.2.6. Define a correlate indicator	10
4.2.7. Define a derived indicator	10
4.2.8. Define a slope indicator	11
4.2.9. Define a constant indicator	11
4.3. Output definition	11
4.3.1. Create an output	11
4.3.2. Define a console output	12
4.3.3. Define a file output	12
4.3.4. Define a mule output	12
4.4. Probe definition	12
4.5. Configuration file for JASMINe Probe artifacts	13
5. JASMINe probe artifacts management	14
5.1. Target management	14
5.1.1. List targets	14
5.1.2. Remove targets	14
5.2. Indicator management	14
5.2.1. List indicator types	14
5.2.2. Show properties for and indicator type	14
5.2.3. List indicators	14
5.2.4. Remove indicators	14
5.2.5. Change an indicator	14
5.3. Output management	15
5.3.1. List output types	15
5.3.2. Show properties for and output type	15
5.3.3. List outputs	15
5.3.4. Remove outputs	15
5.3.5. Change an output	15
5.4. Probe management	15
5.4.1. List probes	15
5.4.2. Remove probes	15
5.4.3. Change a probe	15
6. Running probes	16
6.1. Interfaces for indicator values	16
6.1.1. JasmineIndicatorValue	16
6.1.2. JasmineSingleResult	17
6.2. Indicator values	18

6.2.1. Jmx indicator values	18
6.2.2. Df indicator values	18
6.2.3. Aggregate and merge indicator values	19
6.2.4. Correlate indicator values	19
6.2.5. Derived indicator values	20
6.2.6. Slope indicator values	20
6.3. Output formats for indicator values	20
6.3.1. Console output format	21
6.3.2. File output format	22
6.3.3. Mule output format	23
7. Advanced Configuration	24
7.1. JASMINe Probe's platform configuration	24
7.2. JASMINe Probe configuration	24
8. Tools	26
8.1. Felix Shell Commands	26
8.1.1. Target management	26
8.1.2. Indicator management	26
8.1.3. Output management	27
8.1.4. Probe management	27
8.1.5. Configuration management	27
8.2. JProbe Client	27
8.3. Probe Manager	27
A. Probe configuration schema	28
B. Fragments naming grammar	30

List of Tables

4.1. Specific properties to define for a jmx indicator	7
--	---

Chapter 1. Introduction

JASMINe Probe is a Java application developed within the OW2 JASMINe project [<http://wiki.jasmine.ow2.org/xwiki/bin/view/Main/WebHome>]. It's aimed to monitor Java Middleware *targets* by running *probes*.

The different kind of targets are Java EE Servers, ESB, Java EE Applications, the Java Virtual Machines, and even the OS.

The role of a probe is to periodically collect monitoring data, represented by *indicator* values, in order to make real time, or delayed, target supervision. Indicator values are published by *outputs* towards different type of consumers, like for example the JASMINe Monitoring [<http://wiki.jasmine.ow2.org/xwiki/bin/view/Main/Monitoring>] application.

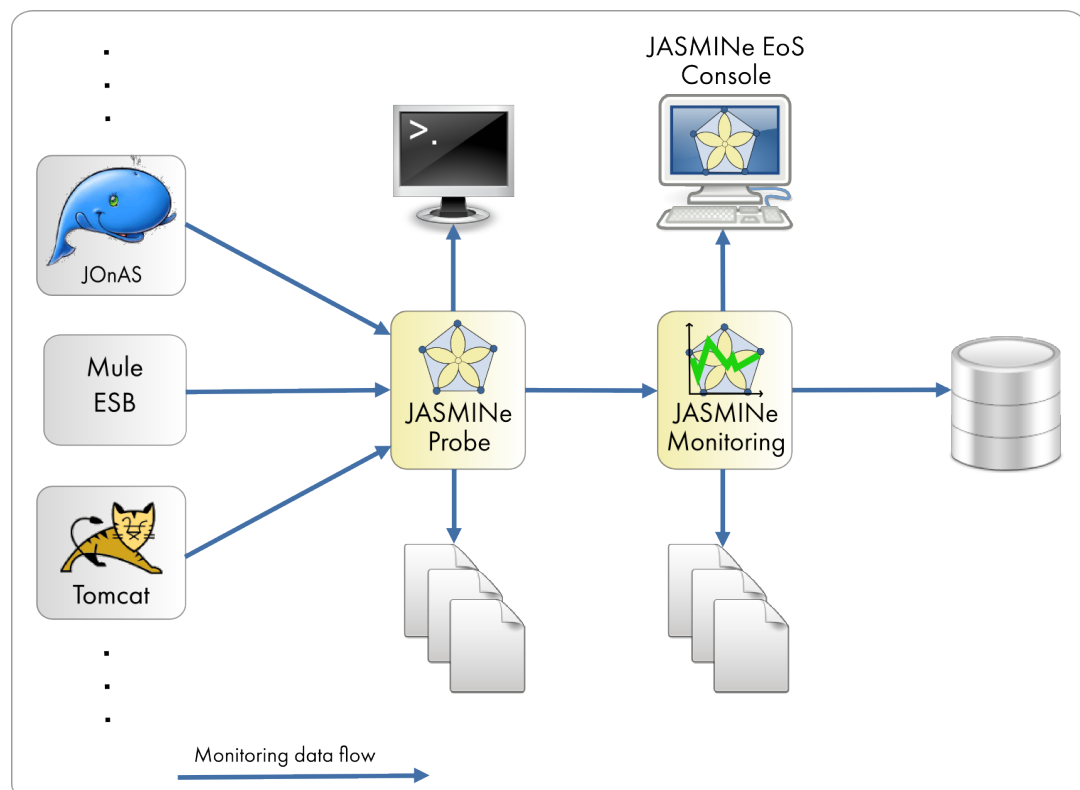
Several techniques are used to collect data. One of them is based on JMX, the Java Management standard, which allows to collect monitoring data from the Management Beans (MBeans) exposed by the targets. There are plans to support other standards, like SNMP, in the future. Besides that, some system monitoring data can be directly obtained by reading system files in an OS dependent implementation.

JASMINe Probe is modular and extensible. It is based on OSGi. The underlying OSGi framework and the modular architecture of JASMINe Probe allow to easily add extensions that support new *targets*, *indicator*, or *output* types.

JASMINe Probe can be used alone or in interaction with JASMINe Monitoring.

In standalone mode, the monitoring data collected by probes can be browsed directly on the console terminal or/and saved into files.

Used in conjunction with JASMINe Monitoring, the indicator values are published into the JASMINe Monitoring's message bus. This allows to use JASMINe EoS, the JASMINe Monitoring Web Console for live or delayed supervision. Moreover, it allows to detect abnormal behavior on targets, and possibly to execute repairing actions (see JASMINe Monitoring documentation).



Chapter 2. How to use JASMINe Probe

2.1. Where to install JASMINe Probe

JASMINe Probe must be installed and run locally, on the monitored target's host, in case the monitored target is the OS.

Otherwise, when monitoring Java targets, it can be installed on a distant host. Remote access to the monitored targets is based on RMI (RMI based JMX connectors).

2.2. Installing JASMINe Probe

The JASMINe Probe project provides several packages allowing to install and use the JASMINe Probe application.

- A package *jasmine-probe-standalone.zip* contains a pre-configured OSGi based JOnAS [<http://wiki.jonas.ow2.org/xwiki/bin/view/Main/WebHome>] platform.
- *jprobe-client.jar* represents a client application providing commands for the creation and management of JASMINe Probe artifacts.

Unpacking the package results to a pre-configured JOnAS platform having the JASMINe Probe application ready to be deployed.

```
$ cd ...
$ unzip jasmine-probe-standalone.zip
$ cd jasmine-probe-standalone
$ pwd
/home/.../jasmine-probe-standalone <----- jasmine-probe-standalone_dir
```

Set the following environment variables:

```
$ export JPROBE_ROOT=jasmine-probe-standalone_dir
$ export PATH=$JPROBE_ROOT:$PATH
```

The *jasmine-probe-standalone_dir* installation directory contains several files and directories including:

- *jasmine-probe.sh* and *jasmine-probe.bat* commands allowing to start/stop the platform.
- *conf* directory containing configuration files for JOnAS and a *probe-config.xml* configuration file containing some JASMINe Probe artifacts definitions.
- *jprobe-client.jar* client application allowing to dynamically create and manage JASMINe Probe artifacts.
- *deploy* directory containing a deployment plan for JASMINe Probe modules' deployment.
- *repositories*, a local maven repository containing the JASMINe Probe modules.

2.3. Running JASMINe Probe in *standalone* mode

Then start the platform. The JASMINe Probe application is automatically deployed due to the deployment plan under the *deploy* directory.

```
$ jasmine-probe.sh start
```

The traces show that a JOnAS server, named 'jasmine-probe' is started and JASMINe Probe modules deployed. Moreover, some probes are created corresponding to definitions provided in the *probe-config.xml* configuration file located under the *conf* directory.

Now JASMINe Probe is ready for user to:

- create new probes,
- start/stop probes execution,
- manage created probes (list, change or remove),
- save probes definitions to a XML file,
- load probes definitions from a XML file.

These operations can be done using Apache Felix Shell Commands or executing the *jprobe-client.jar* application. See Chapter 8, *Tools* for details.

You may stop the platform:

```
$ jasmine-probe.sh stop
```

Chapter 3. Definitions

3.1. Main Concepts

The main concepts used in JASMINe Probe are: *target*, *indicator*, *output* and *probe*.

The role of a *probe* is to supervise a *target* by periodically collecting monitoring data.

The role of an *indicator* is to specify the data to be collected. In JASMINe Probe, the monitoring data is represented by *indicator* values.

Finally, the role of an *output* is to specify what to do with the collected data (this depends on how the collected data is intended to be used).

In order to run *probes*, one have to define *indicators*, *outputs* and possibly *targets*, then use these artifacts to define the *probes*.

3.2. Targets

A target is identified by a *name* chosen by the user and specified in the target's definition.

Target names must be unique. They are used in probe definitions, but may also be used in indicator definitions (only for some indicator types).

Currently, only one target type is supported in JASMINe Probe: the *jmx* target type. When the monitored target is the OS, there is no need to define explicitly a target artifact. When monitoring a JVM, a *jmx* target must be defined.

Besides *name* and *type*, the *jmx* target definition includes other properties, like the *JMX URL* allowing to establish connection with the monitored target (see Section 4.1, "Target Definition").

3.3. Indicators

An *indicator* specifies the monitoring data to be collected by probes.

It is identified by a *name* chose by the user and specified in its definition.

Indicator names must be unique. They are used in probe definitions (each probe definition must specify at least one indicator). When running a probe, the indicator names are used to identify the values produced by the probe and published by outputs.

Several types of indicators are supported in JASMINe Probe. The complete list is provided in Chapter 4, *JASMINe Probe artifacts creation*. New indicator types may be developed and thus extend JASMINe Probe's capability to monitor targets. Moreover, all the indicator types are optional (see ???).

Besides *name* and *type*, an indicator definition includes other properties that depend on its type.

We can distinguish two indicator categories: *raw* and *processing* indicators.

- **Raw** indicators specify data to be collected from targets. A raw indicator's values are obtained by reading some resource values (MBean attributes, system resources, etc.)
- **Processing** indicators are aimed to produce new monitoring values by processing some *source* indicator values.

Several kind of processing operations are supported by the different processing indicator types, like *aggregate* or *correlate*.

A processing indicator takes as input the values corresponding to one or more source indicators, that may be raw indicators or even other processing indicators.

Depending on its type, a processing indicator transforms or combines the input values. The resulting values may also combine data providing from different collect iterations. For example, $\text{delta}(v) = \text{current}(v) - \text{previous}(v)$, where v is a source indicator's value.

In some cases, constant values are necessary to perform processing operations. For this reason, JASMINe Probe provides a particular type of indicators that have a predefined constant value, and that can be used as source indicators within processing indicators definitions.

3.4. Outputs

An output is identified by a *name* chosen by the user and specified in its definition.

Output names must be unique. They are used in probe definitions (each probe definition must specify at least one output).

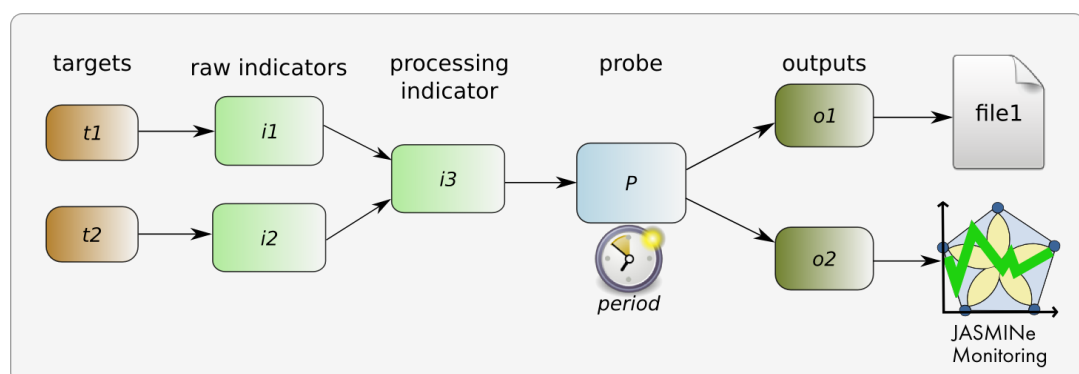
Several output types are provided (*console*, *file*, etc.), and new ones can be easily developed in order to extend JASMINe Probe.

An output type defines a format and a destination for the indicator values obtained in each collect iteration.

3.5. Probes

In order to define a probe, users have to specify:

- period (it exists a default period)
- indicators (at least one indicator name)
- outputs (at least one output name)
- targets (optional; this depends on the indicators type and definition properties)



Chapter 4. JASMINE

Probe artifacts creation

This chapter describes the different target, indicator and output types currently supported by JASMINE Probe.

It presents parameters necessary to create the different JASMINE Probe artifacts: targets, indicators, outputs and probes.

4.1. Target Definition

In order to define a target, users have to specify:

- **name:** a user defined name that uniquely identifies the target
- **type:** only *jmx* supported
- **properties:** defines parameters necessary to establish connection with the target
 - **url:** JMX URL (required)
 - **user:** only required when the jmx server is secured
 - **password:** only required when the jmx server is secured

Examples:

```
<!-- the standalone jasmine-probe server -->
<target name="agent0" type="jmx">
  <property key="url" value="service:jmx:rmi:///jndi/rmi://localhost:4099/
jrmpconnector_jasmine-probe"/>
</target>
<!-- the jasmine-monitoring server -->
<target name="jasmine" type="jmx">
  <property key="url" value="service:jmx:rmi:///jndi/rmi://localhost:1199/
jrmpconnector_jasmine-monitoring"/>
</target>
<!-- a jonas server with secured jmx -->
<target name="sec_jonas" type="jmx">
  <property key="url" value="service:jmx:rmi:///jndi/rmi://localhost:1099/
jrmpconnector_jonas"/>
  <property key="user" value="jonas"/>
  <property key="password" value="jonas"/>
</target>
```

4.2. Indicator definition

4.2.1. Create an indicator

In order to define an indicator, users have to specify:

- **name:** a user defined name that uniquely identifies the indicator.
- **type:** one of the available types depending on the current JASMINE Probe configuration (see ???). A management operation exists allowing to list the available types. The default configuration makes available all the indicator types described in the section below.

- **scale**: optional numeric value that defaults to 1; defines a scale for an indicator having a number as value.
- **properties**: depend on the type.

4.2.1.1. Indicator types

Several raw indicator types are provided by JASMINe Probe.

Raw indicator types are characterized by the technique used to collect the monitoring data:

- **jmx**: use JMX to get monitoring data from MBeans registered in the target's MBean server.
- **df**: allows to get data about file system disk availability (in p.c.); based on the Unix *df* command
- **lewys**: allows to get different kind of system resource parameters (disk, memory, cpu, network); its based on lewys commands (see ..?..)

Processing indicators are characterized by the *processing operation* and the *source indicators*:

- **aggregate**: aggregates a source indicator's values
- **correlate**: correlates several source indicators' value
- **derived**: get a new value from a given source indicator's value ; several derived operations exist, as for example *delta*, that implements $\text{delta}(v) = \text{current}(v) - \text{previous}(v)$, where *v* is the source indicator's value
- **slope**: calculates slope value from two source values ($\text{delta}(s1) / \text{delta}(s2)$ where *s1* and *s2* are the two source indicators)

Constant indicators are characterized by their value and type.

The list of the available indicator types, and the description of properties to be provided when defining an indicator of a given type, can be obtained using JASMINe Probe management operations (see Chapter 5, *JASMINe probe artifacts management*).

4.2.2. Define a jmx indicator

Table 4.1. Specific properties to define for a jmx indicator

Property name	Property value
mbean	an OBJECT_NAME or an OBJECT_NAME pattern conform to the MBean ObjectName Syntax. Required. Allows to determine the MBean(s) to be polled
attr	list of <i>attribute</i> , or <i>attribute fragment</i> names
target	default target's name. Optional - if not specified, a target must be specified by a probe using this indicator

An attribute fragment is a part of an attribute's value, when the attribute has a complex type. See below for details.

The **attr** property determines the attributes to be read from the MBeans. If not defined, all the attributed are read.

Examples:

```

<indicator name="procnb" type="jmx">
  <property key="mbean" value="java.lang:type=OperatingSystem"/>
  <property key="attr" value="AvailableProcessors"/>
</indicator>

<indicator name="gc_collection_count" type="jmx">
  <property key="mbean" value="java.lang:type=GarbageCollector,*"/>
  <property key="attr" value="CollectionCount"/>
</indicator>

<indicator name="threading" type="jmx">
  <property key="mbean" value="java.lang:type=Threading"/>
  <property key="attr" value="TotalStartedThreadCount,ThreadCount,PeakThreadCount"/>
</indicator>

<indicator name="memoryHeap" type="jmx">
  <property key="mbean" value="java.lang:type=Memory"/>
  <property key="attr" value="HeapMemoryUsage"/>
</indicator>

```

4.2.2.1. Attribute fragments

In the case of an attribute having a complex value, the user may want to poll not the whole value, but only a part of it (a fragment), or several fragments.

To specify an attribute fragment, the user must provide a *fragment name* based on the naming convention presented in Appendix B, *Fragments naming grammar*.

The fragment name contains a <base name> given by the attribute name, and a sequence of <fragment elements> corresponding to the path from the base down to the part of attribute user wants to be polled.

The *fragment value* is the part of the attribute's value corresponding to the path expressed by the fragment name.

Let's consider for example the JVM MBean having OBJECT_NAME *java.lang:type=Memory*. The attribute named *HeapMemoryUsage* has a complex type (*javax.management.openmbean.CompositeData*). The possible key values are: *committed*, *used*, *init*, *max*. If the user wants to poll only the value corresponding to the *use* key, he specifies the fragment named *HeapMemoryUsage.used* as **attr** property value, instead of using the "*HeapMemoryUsage*" attribute name.

Here is the indicator definition, and some more attribute fragments examples:

```

<indicator name="memoryHeapUsed" type="jmx">
  <property key="mbean" value="java.lang:type=Memory"/>
  <property key="attr" value="HeapMemoryUsage.used"/>
</indicator>

<indicator name="inputArg0" type="jmx">
  <property key="mbean" value="java.lang:type=Runtime"/>
  <property key="attr" value="InputArguments[0]"/>
</indicator>

<indicator name="propFileEncoding" type="jmx">
  <property key="mbean" value="java.lang:type=Runtime"/>
  <property key="attr" value="SystemProperties[file.encoding]"/>
</indicator>

```

4.2.2.2. Jmx indicator values

Let's consider a probe that has in its definition the above **jmx** indicator named *memoryHeapUsed*. When running the probe, at each polling iteration, the indicator value is obtained by getting from the "*HeapMemoryUsage*" attribute's composite data, the value corresponding to the "*used*" key.

What about the value obtained for another indicator, named *memoryHeap*, having the **attr** property set to "HeapMemoryUsage" ? In this case, the indicator value is composed of several values, one for each key. We call these values **indicator results**.

A **jmx** indicator's value is composed of more than one indicator results in the following situations:

- several MBeans correspond to the **mbean** property (see indicator named *gc_collection_count* in the above examples)
- more than one elements are defined by the **attr** property (see indicator named *threading*)
- one of the attributes or fragments defined in the **attr** property has a complex value (see indicator named *memoryHeap*)

Each result is identified by a name that is generated using the fragments naming convention presented in Appendix B, *Fragments naming grammar*. Each result's value has a simple type (a Number, String or Boolean).

4.2.3. Define a df indicator

The following property may specify one or more device names to check:

- **disk**: device path names relative to */dev*

When the **disk** property is not defined, the available space, in percentage, for all the disks is to be obtained.

A **df** indicator's value is composed of more than one indicator results if several devices are monitored.

Each result is identified by a name that is equal to the device name. Each result's value is the used space on the device in percentage.

Examples:

```
<indicator name="df-sda1" type="df">
  <property key="disk" value="sda1"/>
</indicator>
```

4.2.4. Define a lewys indicator

Two properties must be defined for a lewys indicator:

- **cmd**: a lewys command corresponding to the monitored resource type ; the possible values are *disk*, *memory*, *network*, *cpu* and *kernel*
- **resources**: a list of resource names (see the *probe-config.xml* configuration file for details)

Examples:

```
<indicator name="lewys-disk" type="lewys">
  <property key="cmd" value="disk"/>
  <property key="resources" value="sda1 reads issued,sda1 reads merged,sda1 read
sectors,sda1 read time in ms"/>
</indicator>
```

4.2.5. Define an aggregate (or merge) indicator

The **aggregate** and **merge** indicators definition requires:

- **op**: the aggregate operation ; the possible values are *sum*, *average*, *max* and *min*
- **source**: source indicator name. The source indicator value must only have number results.
- **merge**: not implemented yet, its aim is to filter out some non relevant values from the aggregation.

The value of an **aggregate** or **merge** indicator contains one result. Its value is obtained by aggregation of the source indicator's results.

Examples:

```
<indicator name="gc_collection_count" type="jmx">
  <property key="mbean" value="java.lang:type=GarbageCollector,*"/>
  <property key="attr" value="CollectionCount"/>
</indicator>

<indicator name="total_gc_collection_count" type="aggregate">
  <property key="op" value="sum"/>
  <source>gc_collection_count</source>
</indicator>
```

4.2.6. Define a correlate indicator

The **correlate** indicator definition requires:

- **op**: the correlate operation ; the possible values are *add*, *sub*, *mul*, *div*, *percent*
- **sources**: source indicator names. Source indicator values must be numbers.

Current limitation: all the source indicators values must contain only one result.

Examples:

- **type**: one of the followings are accepted *long*, *int*, *float*, *double*
- **value**: the constant value

```
<indicator name="memoryHeapUsed" type="jmx">
  <property key="mbean" value="java.lang:type=Memory"/>
  <property key="attr" value="HeapMemoryUsage.used"/>
</indicator>

<indicator name="memoryHeapCommitted" type="jmx">
  <property key="mbean" value="java.lang:type=Memory"/>
  <property key="attr" value="HeapMemoryUsage.committed"/>
</indicator>

<indicator name="memoryHeapUsedPercent" type="correlate">
  <property key="op" value="percent"/>
  <source>memoryHeapUsed</source>
  <source>memoryHeapCommitted</source>
</indicator>
```

4.2.7. Define a derived indicator

The **derived** indicator definition requires:

- **op**: the derived operation ; the possible values are *prev*, *delta*, *rate*, having the following meanings:
 - **prev**: at each iteration, return the results corresponding to the previous iteration,

- **delta**: at each iteration and each result, return the value corresponding to the current iteration, minus the value corresponding to the previous iteration,
- **rate**: at each iteration and for each result, return the $\text{delta}(v)/\text{delta}(t)$,
- **sources**: source indicator name. The source indicator value may be composed of several results, but the result values must only be numbers.

The value of an **derived** indicator is composed of as many results as the source indicator's value. Each result in the derived indicator value is obtained by applying the derived operation on one result in the source indicator value.

Examples:

```
<indicator name="totalStartedThreadCount" type="jmx">
  <property key="mbean" value="java.lang:type=Threading" />
  <property key="attr" value="TotalStartedThreadCount" />
</indicator>

<indicator name="deltaStartedThreadCount" type="derived">
  <property key="op" value="delta" />
  <source>totalStartedThreadCount</source>
</indicator>
```

4.2.8. Define a slope indicator

Two sources must be defined for a slope indicator, both corresponding to a source indicator. The first one for the *numerator*, and the second one for the *denominator*. The slope value corresponds to $(\text{delta}(s1) / \text{delta}(s2))$.

Current limitation: both source indicators must have values composed of only one result.

4.2.9. Define a constant indicator

The following properties must be specified:

- **type**: one of the followings are accepted *long*, *int*, *float*, *double*
- **value**: the constant value

Example:

```
<indicator name="1K" type="constant">
  <property key="type" value="int" />
  <property key="value" value="1000000" />
</indicator>
```

4.3. Output definition

4.3.1. Create an output

In order to define an output, users have to specify:

- **name**: a user defined name that uniquely identifies the output.
- **type**: one of the available types depending on the current JASMINE Probe configuration (see ???). A management operation exists allowing to list the available types. The default configuration makes available all the output types described in the sections below.

- **default**: if *true*, this output will be part of the default output list for probes with no explicitly defined output. Default value is *false*
- **properties** that depend on the type.

4.3.1.1. Output types

- **console**: outputs the indicator values to the console terminal,
- **file**: saves the indicator values into a file,
- **mule**: the indicator values are published into a JASMINe Monitor server's event bus (currently based on Mule).

The list of the available output types, and the description of properties to be provided when defining an output of a given type, can be obtained using JASMINe Probe management operations (see Chapter 5, *JASMINe probe artifacts management*).

4.3.2. Define a console output

Users only have to provide a name to define a **console** output. No other parameters are required. The name can then be used in probes' definition to specify that indicator values are to be printed on the console terminal.

Examples:

```
<output name="stdio" type="console">
</output>
```

4.3.3. Define a file output

One property named **path** has to be defined for a **file** output. Its value gives the relative name, or a complete path for a file in which indicator values have to be saved.

Examples:

```
<output name="log1" type="file">
  <property key="path" value="/tmp/log1.csv"/>
</output>
```

4.3.4. Define a mule output

One property named **url** has to be defined for a **mule** output. Its value gives the address of the JASMINe Monitor server's event bus.

Examples:

```
<output name="eventswitch" type="mule">
  <property key="url" value="tcp://localhost:18564/JProbe"/>
</output>
```

4.4. Probe definition

In order to define a probe, users can specify:

- **id**: the probe name (must be unique ; generated by JASMINe Probe if not specified by the user).

- **period:** defines the execution period. The default value is 10 sec.
- **indicators:** list of indicators. At least one indicator name should be specified.
- **outputs:** list of outputs. At least one output name should be specified.
- **targets:** list of targets. Optional. If defined, currently only one target is allowed. This target has priority over those specified in the jmx indicators' definition.
- **start:** boolean that specifies, if true, to immediately start the probe after creation completes. Optional (by default the probe start is a separate operation).

4.5. Configuration file for JASMINe Probe artifacts

JASMINe Probe artifacts can be dynamically created using the tools presented in see Chapter 8, *Tools*.

An alternative approach is to provide definitions in a configuration file named *probe-config.xml*. Its location is the *conf* directory within the JASMINe Probe installation directory. This file is read at JASMINe Probe start-up, and the defined artifacts are created. Probes can even be started in case their definition specifies *status="started"*.

Once JASMINe Probe started, management operations can be invoked to create new artifacts and to manage the existing ones.

All the JASMINe probe artifacts existing at a moment of time may be saved in the JASMINe probe configuration file. Moreover, a *probe-config.xml* file can be explicitly loaded in order to create the necessary JASMINe Probe artifacts.

Here is a *probe-config.xml* snippet containing definitions for JASMINe Probe artifacts depicted in Chapter 3, *Definitions* section.

```
<target name="t1" type="jmx"/>
  <property key="url" value="service:jmx:rmi:///jndi/rmi://host1:1099/
jrmpconnector_jonas"/>
</target>
<target name="t2" type="jmx"/>
  <property key="url" value="service:jmx:rmi:///jndi/rmi://host2:1099/
jrmpconnector_jonas"/>
</target>
<indicator name="i1" type="jmx"/>
  <property key="mbean" value="java.lang:type=Memory"/>
  <property key="attr" value="HeapMemoryUsage.used"/>
  <property key="target" value="t1"/>
</indicator>
<indicator name="i2" type="jmx"/>
  <property key="mbean" value="java.lang:type=Memory"/>
  <property key="attr" value="HeapMemoryUsage.used"/>
  <property key="target" value="t2"/>
</indicator>
<indicator name="i3" type="correlate"/>
  <property key="op" value="sum"/>
  <property key="sources" value="i1,i2"/>
</indicator>
<output name="o1" type="file"/>
  <property key="path" value="/tmp/f1"/>
</output>
<output name="o2" type="mule">
  <property key="url" value="tcp://localhost:18564/JProbe"/>
</output>
<probe id="p" period="5" status="started">
  <indicator>i3</indicator>
  <output>o1</output>
  <output>o2</output>
</probe>
```

Chapter 5. JASMINe probe artifacts management

This chapter presents the operations allowing to manage the created targets, indicators, outputs and probes.

5.1. Target management

5.1.1. List targets

Shows all the defined targets and their definitions.

5.1.2. Remove targets

Used to remove target definitions. This operation fails if a target to be removed is used by a running probe.

5.2. Indicator management

5.2.1. List indicator types

Show the available indicator types.

5.2.2. Show properties for and indicator type

Allows to show the specific properties that need to be specified for a given indicator type, when creating an indicator of that type.

5.2.3. List indicators

Show all the defined indicators and their definitions. Its possible to list the indicators of a given type only.

5.2.4. Remove indicators

Remove indicators definitions. This operation fails if a indicator to be removed is used by a running probe.

5.2.5. Change an indicator

Change an indicator's definition (to redefine the indicator's type and/or properties). The parameters that are not redefined keep their initial values.

If the indicator to be changed is used by a running probe, the probe is stopped, and execution restarted after the change completed.

5.3. Output management

5.3.1. List output types

Show the available output types.

5.3.2. Show properties for and output type

Allows to show the specific properties that need to be specified for a given output type, when creating an output of that type.

5.3.3. List outputs

Show all the defined outputs and their definitions. Its possible to list the outputs of a given type only.

5.3.4. Remove outputs

Remove outputs definitions. This operation fails if a output to be removed is used by a running probe.

5.3.5. Change an output

Change an output's definition. If the output to be changed is used by a running probe, the probe is stopped, and execution restarted after the change completed.

5.4. Probe management

5.4.1. List probes

Show all the defined probes and their definitions.

5.4.2. Remove probes

Remove probes definitions.

5.4.3. Change a probe

Change a probe's definition. If the probe is running, the probe is stopped, and execution restarted after the change completed. The operation allows to redefine any of the probe's parameters. The parameters that are not redefined keep their initial values. For example, the user may define a new target for the probe and keep the rest of the definition unchanged.

Chapter 6. Running probes

JASMINe Probe provides operations to *start* and *stop* a probe's execution any number of times.

Once started, the probe's status change from STOPPED to RUNNING if no error occurs, or to FAILED otherwise.

Reasons that may lead to an error state may be a bad JASMINe probe artifact definition, or some problem related to polling the specified resources. For example, when using a jmx indicator, the jmx target may be unreachable. In this case, definitions must be checked, and verify urls and/or resource names.

Note that *list probes* operation allows to see the probes' status.

A RUNNING probe periodically collects the values corresponding to all the indicators specified in its definition. All the obtained values are passed to the outputs specified in the probe's definition. Each output publishes the values in a specific format on a given support (console, file, bus, etc.). See Section 6.3, "Output formats for indicator values".

At each polling iteration, several values are collected because one or more of the following reasons:

- Several indicators are specified by the probe's definition.
- Several resources are polled for an indicator. For example, if a jmx indicator is used, several MBeans or several attributes or attribute fragments may be specified in the indicator's definition.
- A polled resource may have a complex value. In this case, the indicator value is decomposed in fragments having simple values (Number, String, Boolean), also called simple results.

The JASMINe Probe API defines data types (*JasmineIndicatorValue* and *JasmineSimpleResult* classes) allowing to construct a data structure for each indicator value obtained at each iteration.

Each indicator value is identified by the *indicator name*. Moreover, each simple result that is composing an indicator value is identified by a *result name* that is generated by JASMINe Probe based on the fragments naming convention (see Appendix B, *Fragments naming grammar*). In most of the cases, the indicator name + the result names are sufficient to uniquely distinguish the obtained results. In some particular case, other metadata associated to the results make possible the distinction.

6.1. Interfaces for indicator values

6.1.1. JasmineIndicatorValue

```
public class JasmineIndicatorValue implements Serializable {
```

```

/**
 * Indicator Name.
 */
private String name;

/**
 * Target name
 */
private String target;

/**
 * Additional Metadata, depending on the indicator type.
 */
private Map metadata;

/**
 * Collection of results
 * We have a Collection because indicators may have complex values.
 */
private Collection<JasmineSingleResult> values;
}

```

6.1.2. JasmineSingleResult

```

public abstract class JasmineSingleResult implements Serializable {

    /**
     * Result Name.
     */
    private String name;

    /**
     * Timestamp of the measure in millis
     */
    private long timestamp;

    /**
     * List of properties associated to this value.
     * Their name depend on the Collector type.
     */
    private Map properties;

    public abstract Object getValue();

    public abstract void setValue(Object value);
}

```

Several sub-classes are provided corresponding to the supported result types: *JasmineSingleNumberResult*, *JasmineSingleStringResult*, *JasmineSingleBooleanResult* and *JasmineSingleObjectNameResult*.

```

public class JasmineSingleNumberResult extends JasmineSingleResult {
    /**
     * Numeric value of this result
     *
     * The abstract class Number is the superclass of classes
     * BigDecimal, BigInteger,
     * Byte, Double, Float,
     * Integer, Long, and Short.
     */
    private Number value;

    public Number getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = (Number) value;
    }
}

```

6.2. Indicator values

6.2.1. Jmx indicator values

A *JasmineIndicatorValue* instance corresponding to a **jmx** indicator is initialized with the following values:

- **name** = the name provided by the indicator's definition ;
- **target** = the target name provided by the indicator's definition, except if the probe's definition specifies a target. In this latter case the probe's definition take precedence, and provides the target name.
- the **metadata** is composed by the following properties:
 - **domain** = the management domain name to which belongs the target jmx server,
 - **server** = the target server name, when the monitored target in a Java EE Server like JOnAS, or a WEB Server like Jetty or Tomcat. Otherwise, the default value is "unknown",
 - **url** = the jmx url from the target definition.

A *JasmineSingleResult* instance is initialized with the following values:

- **name** = an attribute name if it has a simple type, an attribute fragment name otherwise,
- **value** = the attribute's value if its simple, or the fragment's value otherwise,
- **timestamp** = the measurement time,
- the **properties** is composed by:
 - **mbean** = the OBJECT_NAME corresponding to the MBean containing the read attribute.

6.2.2. Df indicator values

A *JasmineIndicatorValue* instance corresponding to a **df** indicator is initialised with the following values:

- **name** = the name provided by the indicator's definition,
- **target** = the host name,
- the **metadata** is composed by the following properties:
 - **domain** = the management domain name to which belongs the target jmx server,
 - **server** = the target server name, when the monitored target in a Java EE Server like JOnAS, or a WEB Server like Jetty or Tomcat. Otherwise, the default value is "unknown".

A *JasmineSingleResult* instance is initilaised with the following values:

- **name** = a device name,
- **value** = free space on that device in percentage,
- **timestamp** = the measurement time,
- **properties** not used.

6.2.3. Aggregate and merge indicator values

A *JasmineIndicatorValue* instance corresponding to a **aggregate** or **merge** indicator is initialised with the following values:

- **name** = the name provided by the indicator's definition,
- **target** = the **target** in the source *JasmineIndicatorValue*,
- the **metadata** is composed by the same elements that compose the **metadata** in the source *JasmineIndicatorValue*.

A *JasmineSingleResult* instance is initialised with the following values:

- **name** = empty String,
- **value** = the values obtained by the aggregation of the values corresponding to the source indicator's results,
- **timestamp** = the biggest **timestamp** from the source indicator's results,
- **properties** contains all the elements composing the **properties** from all the source *JasmineSingleResults*.

6.2.4. Correlate indicator values

The value of a **correlate** indicator is composed of one result. Its value is obtained by applying the correlate operation to the source result values.

A *JasmineIndicatorValue* instance corresponding to a **correlate** indicator is initialised with the following values:

- **name** = the name provided by the indicator's definition
- **target** =
 - if all the source indicators have the same target, use the **target** in any of source *JasmineIndicatorValues*.
 - **misc** otherwise
- the **metadata**: look for "domain" and "server" keys in the source *JasmineIndicatorValues*. If all the values found for "domain" are the same, a "domain" element having the found value is inserted in **metadata**. Otherwise, a "domain" element having **misc** value is inserted. A similar treatment is done for the "server" metadata.

A *JasmineSingleResult* instance is initialised with the following values:

- **name** = empty String
- **value** = the values obtained by applying the correlate operation over the values from each source's *JasmineSingleResult* unique element.
- **timestamp** = the biggest **timestamp** from the source indicators' results
- **properties**: look for "mbean" property name in the source *JasmineSingleResults*. If all the found values are the same, a "mbean" element having the found value is inserted in **properties**. Otherwise, a "mbean" element having **misc** value is inserted.

6.2.5. Derived indicator values

The value of an **derived** indicator is composed of as many results as the source indicator's value. Each result in the derived indicator value is obtained by applying the derived operation on one result in the source indicator value.

A *JasmineIndicatorValue* instance corresponding to a **derived** indicator is initialised with the following values:

- **name** = the name provided by the indicator's definition
- **target** = the **target** in the source *JasmineIndicatorValue*
- the **metadata** is composed by the same elements that compose the **metadata** in the source *JasmineIndicatorValue*

A *JasmineSingleResult* instance is initialised with the following values:

- **name** = empty String
- **value** = the values obtained by transforming the source indicator's result using the specified operation
- **timestamp** = the **timestamp** from the source *JasmineSingleResult*
- **properties** contains all the elements composing the **properties** from all the source *JasmineSingleResults*

6.2.6. Slope indicator values

The result value is obtained as follows: $\text{delta}(\text{vnumerator}) / \text{delta}(\text{vdenominator})$, where *vnumerator* is the value of the numerator's result and *vdenominator* is the value of the denominator's result.

6.3. Output formats for indicator values

At each iteration, the probe collects the values corresponding to each indicator specified in its definition.

For each indicator, one or more results are published by the outputs. They correspond to *JasmineSingleResult* instances referenced by the *JasmineIndicatorValue* instance (by its *values* attribute).

Each result is published in a format that depends on the output type.

The examples below presents the indicator values for a running probe *p2indic* having two indicators, *used-memory* which is a *jmx* indicator, and a *df* indicator named *dfi*.

```
<probe id="p2indic" period="20">
  <indicator>used-memory</indicator>
  <indicator>dfi</indicator>
  <output>stdio</output>
  <output>log</output>
  <output>jm</output>
</probe>
<indicator name="used-memory" type="jmx"/>
  <property key="mbean" value="java.lang:type=Memory"/>
  <property key="attr" value="HeapMemoryUsage.used"/>
  <property key="target" value="agent0"/>
</indicator>
```

```

<indicator name="dfi" type="df"/>
<output name="stdio" type="console"/>
<output name="log" type="file"/>
  <property key="path" value="/tmp/fl"/>
</output>
<output name="jm" type="mule">
  <property key="url" value="tcp://localhost:18564/JProbe"/>
</output>
<target name="agent0" type="jmx"/>
  <property key="url" value="service:jmx:rmi:///jndi/rmi://localhost:4099/
jrmpconnector_jasmine-probe"/>
</target>

```

6.3.1. Console output format

For each result a line is printed to the console terminal. It contains elements separated by the ";" character.

```
probeName;time;date;targetName;name;resultValue;props
```

- **probeName**: the name provided by the probe definition, or generated by JASMINe Probe if no name provided
- **time**: **timestamp** in *JasmineSingleResult*
- **date**: **time** having "yyyy/MM/dd HH:mm:ss" format
- **targetName**: **target** in *JasmineIndicatorValue*. Note that when no explicit target artifact is required, the targetName is replaced by the host name.
- **name**:
 - **indicatorName** in the following cases:

Raw indicators:

the indicator defines one value to poll and the value has a simple type

Processing indicators:

the indicator produces one simple type value

- **indicatorName.resultName** when several simple values are obtained for the indicator

Where **indicatorName** is provided by **name** in *JasmineIndicatorValue*, and **resultName** is provided by **name** in *JasmineSingleResult*

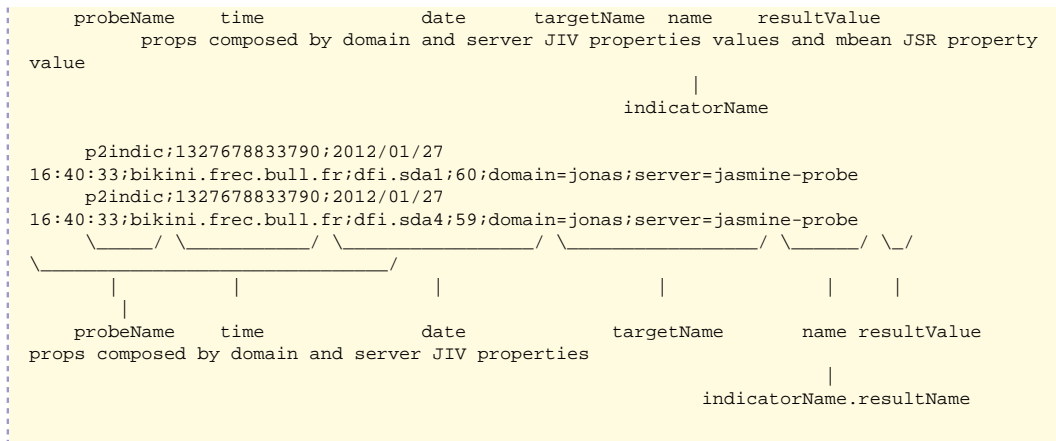
- **resultValue**: **value** in *JasmineSingleResult*
- **props**: a sequence of *propName=propValue* elements corresponding to **metadata** in *JasmineIndicatorValue* and *properties* in *JasmineSingleResult*.

Example below presents values published for *p2indic* probe. This probe has two indicators, a jmx indicator named *used-memory*, and a df indicator named *dfi*.

```

p2indic;1327676884394;2012/01/27 16:08:04;agent0;used-
memory;79207464;domain=jonas;server=jasmine-probe;url=service:jmx:rmi:///jndi/rmi://
localhost:4099/jrmpconnector_jasmine-probe;mbean=java.lang:type=Memory

```



The elements composing the *props* can be used to get information about the target and the resource being monitored. Moreover, in some situation it allows to distinguish between different values having the same name. For example, suppose that the *p2indic* probe contains only one jmx indicator named *used-memory*. Suppose that the probe specifies two targets named *t1* and *t2*. The two values are published having the same *name* but different *targetNames*, and possible different server properties.

Another typical jmx indicator example is an indicator defining a *mbean* property having as value a pattern that corresponds to several MBeans in the target server. In this case, several values are published having the same *name* but different *mbean* property.

6.3.2. File output format

For each result a line is printed to the file. It contains elements separated by the ";" character.

```
time;date;targetName;props;resultValue
```

- **time:** **timestamp** in *JasmineSingleResult*
- **date:** **time** having "yyyy/MM/dd HH:mm:ss" format
- **targetName:** **target** in *JasmineIndicatorValue*. Note that when no explicit target artifact is required, the targetName is replaced by the host name.
- **props:** a sequence of property values from **metadata** in *JasmineIndicatorValue* (the ones corresponding to **domain** and **server**) and the **mbean** property value in *JasmineSingleResult*'s **properties**, combined with the indicator name and result name (**mbean:indicatorName** or **mbean:indicatorName.resultName**).
- **resultValue:** **value** in *JasmineSingleResult*

Example

```

    1327676884394;2012/01/27 16:08:04;agent0;jonas;jasmine-
    probe;java.lang:type=Memory:used-memory;79207464
    \_____/ \_____/ \_____/ \_____/ \_____/ \_____/ \_____/
    |         |         |         |         |         |         |
    time      date      targetName  props
    resultValue
    \_____/ \_____/ \_____/ \_____/ \_____/ \_____/ \_____/
    \_____/ \_____/ \_____/ \_____/ \_____/ \_____/ \_____/
  
```

```
domain;server;mbean:indicatorName
```

6.3.3. Mule output format

For each result 3 lines are published to the JASMINe Monitoring event bus. The first line only contains a format version number. The second contains a header that describes the elements of the third line. The header elements and the data elements from the third line are separated by the ";" character. Here are the header and data lines format:

```
cmdid;time;domain;server;sname;mbean;name  
probeName;date;domainName;serverName;targetName;mbeanName;value
```

Chapter 7. Advanced Configuration

Installing JASMINe Probe (unzip the provided archive) creates an installation directory containing a pre-configured JOnAS server and the JASMINe Probe's application modules.

The sections below describe the JOnAS configuration parameters that could be necessary to change in a production environment. It also details how to change the JASMINe Probe application's configuration.

7.1. JASMINe Probe's platform configuration

JASMINe Probe application runs within OSGi based JOnAS application server. Complete documentation on JOnAS server configuration can be found in http://jonas.ow2.org/JONAS_5_3_0_M5/doc/doc-en/html/configuration_guide.html.

The configuration provided for JASMINe Probe corresponds to a 'micro' server: only a few services are activated, like *registry*, *security* and *jmx* mandatory services, and a deployment service called *depmonitor*.

The *jmx* service provides an MBean server that has registered MBeans for JVM and JOnAS management. In addition, a JASMINe Probe registered MBean having OBJECT_NAME *jasmine:dest=probe-manager* exposes operations for JASMINe Probe artifacts management. This MBean is used by the JASMINe Probe's JMX Client Section 8.2, "JProbe Client".

In order to allow remote access to the MBean server, a JMX RMI connector is available at the platform start-up. By default, the connector's URL is *service:jmx:rmi:///jndi/rmi://localhost:4098/jrmpconnector_jasmine-probe*. The connector's address changes if one of the following JOnAS configuration parameters are modified:

- The JOnAS server name. By default it's set to *jasmine-probe* in the *jasmine-probe.sh* (or *jasmine-probe.bat*) script (-n option for jonas start)
- The JASMINe Probe's host name can be used instead of *localhost* to allow execution of JMX Client on a remote host.
- RMI port number is set to 4098 by default (see *carol.jrmp.url* in *carol.properties* file within *conf* directory).

Remote management may require security. The *jmx* service can be secured, for example, with password authentication. See *jonas.properties* file within *conf* directory for details.

7.2. JASMINe Probe configuration

The JASMINe Probe installation directory contains the following configuration elements

- The *probe-config.xml* file containing JASMINe Probe artifacts definitions, within the *conf* directory.
- The *deploy* directory containing a deployment plan for JASMINe Probe modules' deployment. Some of these modules implement different type of indicators and outputs. The modules that implement indicators correspond to maven artifacts prefixed by *jprobe-*

collector. The modules that implement outputs correspond to maven artifacts prefixed by *jprobe-outer*. All these modules are optional, user may remove the unnecessary ones, or even replace them by new modules corresponding to particular implementations corresponding to its needs.

- *repositories* containing a local maven repository with modules that were mentioned above.

Note that JASMINe Probe modules could be deployed on an already installed JOnAS server.

Chapter 8. Tools

Different tools are provided by JASMINe Probe and by the JASMINe Monitoring Console for the creation, management and execution of probes.

There are two kind of JASMINe Probe tools: a command line tool and a Java client based on JMX.

As presented in the Introduction (Chapter 1, *Introduction*), JASMINe Probe is based on OSGi and currently uses the Apache Felix OSGi implementation. The command line tool is based on, and extends the shell integrated in the Felix OSGi framework.

The second JASMINe Probe tool is a Java program interacting with to the JASMINe Probe application via JMX. This is possible because the Felix framework contains a JMX service in which JASMINe Probe registers a MBean exposing all the operations that create, manage and run probes.

Another tool is a graphical module belonging to the JASMINe Monitoring's console named JASMINe EoS. This tool is used when running JASMINe Probe application inside the JASMINe Monitoring platform.

8.1. Felix Shell Commands

After starting the JASMINe Probe's OSGi platform with the provided *jasmine-probe* shell command, the Felix prompt `->` permits user to type a Felix command. Get list of all the commands by typing the **help** command. If you need a description of a particular command, type **help commandName**.

```
$ jasmine-probe.sh start -tui
> help
arch
...
indicator-XXX
...
outup-XXX
...
probe-XXX
...
target-XXX
```

Commands allowing to manage JASMINe Probe artifacts are prefixed by *indicator-* *output-probe-* and *target-*.

8.1.1. Target management

After typing the **help** command, you can identify target management commands, the commands starting with *target-*: *target-create*, *target-list* and *target-remove*.

8.1.2. Indicator management

Additionally to create, remove and list operations you may change an indicator definition using the *indicator-change* command. Moreover, the *indicator-types* command allows to list all the indicator types available on the platform.

In order to determine the properties that must be defined for an indicator of a given type, use *indicator-properties* command.

8.1.3. Output management

Similar to indicators management, but use *output-* prefix.

8.1.4. Probe management

Probe management commands' name is prefixed by *probe-*. Additionally to create, remove, list and change operations, commands are provided to start or stop a probe, or all probes.

8.1.5. Configuration management

In order to save all the current artifact definitions, use the *probe-config-save* command. Definitions can be loaded from an updated *probe-config.xml* file using the *probe-config-load* command.

8.2. JProbe Client

Use the *jprobe-client.jar* client application contained in JASMINe Probe's installation directory. To get help, simply type

```
$ java -jar jprobe-client.jar
```

The client connects to the JMX server running within JASMINe Probe platform (the JOnAS server). By default, the address used to establish connection is *service:jmx:rmi:///jndi/rmi://localhost:4098/jrmpconnector_jasmine-probe*. The addressed could be changed if the JOnAS server configuration changed. The following options allow to take into account JOnAS configuration changes:

- -server JOnAS server name
- -host host name
- -port JRMP RMI port number

8.3. Probe Manager

This is a graphical tool allowing to create and manage JASMINe Probe artifacts, and to start and stop probes. In order to use this tool, its necessary to start JASMINe Probe modules inside a JASMINe Monitoring server. This is the default case when starting JASMINe Monitoring version 1.4.

Appendix A. Probe configuration schema

Here is the grammar of the probe configuration XML file:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="org.ow2.jasmine.probe:probe-config"
  xmlns:pec="org.ow2.jasmine.probe:probe-config">

  <xsd:element name="probe-config">
    <!-- Use an inner type to avoid that unmarshall return a JAXBElement -->
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="output" minOccurs="0" maxOccurs="unbounded"
type="pec:output" />
        <xsd:element name="target" minOccurs="0" maxOccurs="unbounded"
type="pec:target" />
        <xsd:element name="indicator" minOccurs="0" maxOccurs="unbounded"
type="pec:indicator" />
        <xsd:element name="probe" minOccurs="0" maxOccurs="unbounded"
type="pec:probe" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="output">
    <xsd:sequence>
      <xsd:element name="property" minOccurs="0" maxOccurs="unbounded"
type="pec:proptype" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:token" use="required"/>
    <xsd:attribute name="type" type="xsd:token" use="required" />
    <xsd:attribute name="default" type="xsd:boolean" default="false" />
  </xsd:complexType>

  <xsd:complexType name="target">
    <xsd:sequence>
      <xsd:element name="property" minOccurs="0" maxOccurs="unbounded"
type="pec:proptype" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:token" use="required"/>
    <xsd:attribute name="type" type="xsd:token" default="jmx" />
  </xsd:complexType>

  <xsd:complexType name="indicator">
    <xsd:sequence>
      <xsd:element name="property" minOccurs="0" maxOccurs="unbounded"
type="pec:proptype" />
      <xsd:element name="source" minOccurs="0" maxOccurs="unbounded"
type="xsd:token" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:token" use="required" />
    <xsd:attribute name="type" type="xsd:token" use="required" />
    <xsd:attribute name="scale" type="xsd:integer" default="1" />
  </xsd:complexType>

  <xsd:complexType name="probe">
    <xsd:sequence>
      <xsd:element name="output" minOccurs="0" maxOccurs="unbounded"
type="xsd:token" />
      <xsd:element name="indicator" minOccurs="1" maxOccurs="unbounded"
type="xsd:token" />
      <xsd:element name="target" minOccurs="0" maxOccurs="unbounded"
type="xsd:token" />
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:token" use="required"/>
    <xsd:attribute name="period" type="xsd:integer" default="30" />
    <xsd:attribute name="status" type="pec:statusType" />
  </xsd:complexType>

  <xsd:complexType name="proptype">
```

```
<xsd:attribute name="key" />
<xsd:attribute name="value" />
</xsd:complexType>

<xsd:simpleType name="statusType">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="stopped"/>
    <xsd:enumeration value="started"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Appendix B. Fragments naming grammar

Here is the grammar of the fragments naming use for the MBean attribute names and indicator value names.

```
<fragment name> ::= <base name> <fragment elements>
<base name> ::= <identifier>
<fragment elements> ::= <fragment element> | <fragment element> <fragment elements>
<fragment element> ::= <key element> | <index expression> | <list index>
<key element> ::= "." <key name>
<key name> ::= <identifier>
<index expression> ::= "[" <indexes> "]"
<indexes> ::= <index> | <index> "," <indexes>
<index> ::= <identifier> | <integer>
<list index> ::= "." <integer>
```