Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Job Monitoring Description
D.3.3.5

Due date of deliverable: November 30th, 2008
Actual submission date: November 28th, 2008

Start date of project: June 1st 2006

Type: Deliverable
WP number:3.3
Task number (optional):

Name of responsible: Toni Cortes
Editor & editor's address: Julita Corbalan
Barcelona Supercomputing Center

Version 3.0/ Last edited by Ramon Nou/ Date 08/11/25

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Keyword List: Application Execution Management, Job Monitoring,

Revision history:

Version	Date	Authors	Institution	Sections Affected / Comments
0.9	08/10/10	Julita Corbalan	BSC	Initial Draft
1.0	08/10/23	Ramon Nou	BSC	First Draft
2.0	08/11/20	Ramon Nou	BSC	Final Draft (with reviewers comments)
3.0	08/11/25	Sandrine L'Hermitte / Ramon Nou	INRIA / BSC	Final check (layout…)

Reviewers
Jeff Napper (VUA), Franz J. Hauck (ULM)

Tasks related to this deliverable

Task No.	Task description	Partners involved [°]
3.3.9	Advanced Job monitoring	BSC

[°] This task list may not be equivalent to the list of partners contributing as authors to the deliverable

Executive Summary

In the D3.3.3-D3.3.4 we have shown how we could see information of the running jobs via xps command, in this deliverable we present the advanced monitoring of AEM.

This document describes services and API related with monitoring. Job monitoring is one of the challenges of XtreemOS because we are able to provide information correlating all the scopes involved in the execution of a job: from the job, like a whole, to the specific process running in a node. One of the goals of WP3.3 is to provide a powerful infrastructure to support the monitoring of jobs at different levels of detail and granularity, but also to offer an extensible mechanism that allows future XtreemOS developers and users to include as much information as required.
In the document we present the detailed API to get information, add information and manage monitoring. We also present basic metrics provided by AEM, and the callback system to automatically execute functions based on specific events related with metrics values (such as metric EQUAL_TO value X or status CHANGE_TO RUNNING).

The document shows the different I/O interfaces (using XML) with users of the monitoring interface and the description of the interface.

The basic set of methods is implemented in M30 as well as the XML output. In the next months we will work in the implementation of the methods described here and working forward to the integration of this interface with the SAGA one. This monitoring interface will be related to the accounting that will take place on XtreemOS.

Table of Contents

1. INTRODUCTION 5
2. AEM MONITORIZATION OVERVIEW 5
2.1 LEVELS OF MONITORING 6
2.2 TYPE OF INFORMATION 6
2.3 INPUT AND OUTPUT XML DESCRIPTION 7
2.3.1 MetricDesc 7
2.3.2 Obtaining monitoring information 8
3. XATI FUNCTIONS RELATED WITH JOB MONITORING 9
3.1 OBTAINING JOBID'S 9
3.2 OBTAINING JOB INFORMATION 9
3.3 EXTENDING THE JOB MONITORING INFORMATION 10
3.4 MANAGING THE JOB MONITORING 10
4. SYSTEM PROVIDED METRICS 11
5. XATI COMMANDS RELATED WITH JOB MONITORING 13
6. CHANGES IN THE ARCHITECTURE 13
7. CONCLUSION 13
8. REFERENCES 13

1. Introduction

The AEM, Application Execution Management [D3.3.3-4], component is responsible to provide mechanism to execute jobs. In this deliverable we will introduce the monitorization using this component. AEM will monitor the whole life of jobs, from job creation to job finalization. Each state can have associated a certain amount of information but specially when running. This information can be from the interest of the job itself or of other jobs executed by the same user (for instance *xps* command). The aim of the monitoring provided by AEM is:

- To automatically provide typical information associated to jobs such as execution time,
- To provide a mechanism to limit the type and the granularity of information collected,
- To provide a mechanism to easily add new information to the generated by the system,
- To provide a mechanism to be notified when certain monitoring event occurs (callbacks),

The idea is to provide a basis that provides enough information to be used by typical users without deep knowledge on our system, but as powerful as to allow to include any new metric and then to make a monitoring system easily extensible.

By default, jobs have associated a set of metrics provided by AEM, we will refer to these metrics as system metrics. A metric is a certain type of information with a name, a type, a description (human readable text), and a value (or list of values in case the metric includes the *buffering* feature, see Section 2.2). Any information associated to a job must be defined with a metric, from the jobID to the userTime of each of its processes. Apart from system metrics, the AEM monitoring interface allows jobs to add additional metrics (and its associated values) to provide the extensibility required in the service. We will refer in this document to these metrics as user metrics. User metrics could be, for instance, a string that represents the function name currently executed by a job, or performance metrics provided by dynamically loaded libraries that reads some of the performance counters of the architecture. Since user metrics are dynamically added to specific jobs (by *addJobMetric*), not all the jobs are able to return the same set of metrics. The AEM monitoring interface provides the required functionality to collect the set of valid metrics for a specific job.

The AEM monitoring service will be used by different types of jobs that will require different amounts of information and, potentially, by other XtreemOS services. Having this scenario in mind, the monitoring service (and interface) will support the definition of three *levels* of monitoring that will provide metrics with different granularity. The three levels are: JOB, PROCESS AND KERNEL. These levels provide, respectively, information associated with the scope of the job, the *job+processes*, and *job+processes+advanced* kernel metrics (see Section 2 for more information). We call them *levels* because each one includes additional cost to access the information: includes the previous level plus more information using additional AEM components. Additionally, to define the *level* of the information to be returned, the interface allows jobs to filter the set of metrics to be returned. We offer different predefined set of metrics to make easy the work of jobs. For example, jobs can use the flag BASIC that returns minimal information such as jobID, status or some time metrics such as submission time or start time, or the flags JOB_DEFINITION that includes the jsdl that describes the job including all the dynamic modifications performed by the user during the job life. These flags to select set of metrics can be combined (with logical OR's) to create by default super-sets of metrics, see Section 3 for more details.

To provide the full AEM monitoring functionality, we provide an automatic mechanism to notify jobs when a certain event associated with a metric occurs. By default there is no notification associated to metrics and the jobs themselves must use the specific interface to provide a function that will be executed when the event occurs. Events can be situations where values associated to metrics reach a certain value (EQUAL_TO, GREATHER_THAN, etc), or can be specific situations such as the case that the buffer associated to a metric (with the *buffering* feature activated) is full. The idea behind this functionality is to be able to extend the monitoring service provided in the project as required by jobs or by other XtreemOS components.

We will present the API (provided through **XATI** / **C-XATI** interfaces) to be used by users and tools, and the different Input / Output formats (mainly XML) to use it.

2. AEM Monitorization Overview

We can see how monitorization relates to AEM and XtreemOS architecture in Figure 1.

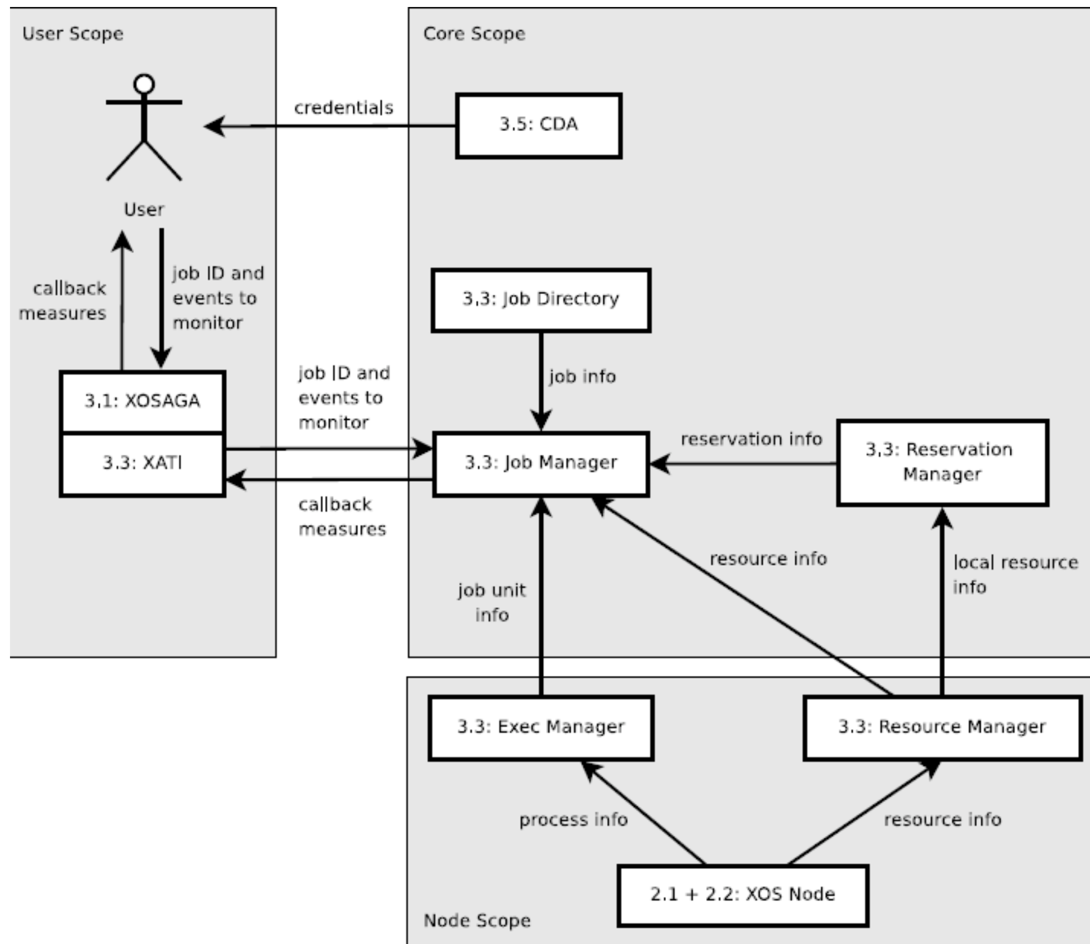

Figure 1 - AEM Monitorization Capability (from D3.1.7)

Monitorization is accessible through Job Manager who gets information from Exec Manager and Resource Manager if needed. AEM Monitoring provides the capability to receive more or less detailed information to decrease overload of the system. Access to the monitorization methods is granted to the user owning the job, but will be granted to other users or tools using Virtual Organization rights like special users or VO admin / creators.

2.1 Levels of monitoring

These levels of monitoring define the granularity of information that will be returned when calling the getJobInfo XATI call. The higher the level we use, the higher is the overhead. We proposed, based on requirements of D3.3.1 [D3.3.1] to have three levels of detail:

- **JOB**: It returns information related to the job as a whole. This information will be collected contacting with the JobManager, minimizing communications between XOSD Daemons.
- **PROCESS**: It returns information of the job and the different processes that compound the job. This information will be collected contacting the JobManager and ExecManagers, having more computational cost than BASIC level.
- **KERNEL**: It activates additional monitoring included with kernel distributions such as performance counters libraries (PAPI (S. Browne, 2000)). This level will depend on the availability of these advanced monitoring features in local systems. It only refers to monitoring systems included in the local systems, in any case this information could be included using the *setMetric* and *setMetricValue* calls.

2.2 Type of Information

Associated to a job and its processes there is a high amount of metrics, from quite static information such as the job definition to very dynamic information such as metrics related with performance counters. The basic function to get job information is the getJobInfo XATI call. This function receives a flag that allows selecting the kind of information we are interested in. These flags can be combined with logical OR's. We have grouped the different types of information in the following groups:

- **BASIC**. This information will always be returned and it includes:
  - Job identification: jobID, UserDN, VO
  - Status: GRID_SUBMITTED, LOCAL_SUBMITTED, RUNNING, CHECKPOINTED & SUSPENDED, SUSPENDED, CANCELLED, FINISHED

  - Time: submission time, start time of the current state
- **JOB_DEFINITION**. JSDL (Job Submission Description Language) submitted and updated with subsequent variations (for instance when updateJobRequirements is called)
- **RESOURCES_ALLOCATED**: Information about resources allocated to the job (and processes)
  - **ReservationsID.** Details of reservations (resourceID&name)
- **RESOURCES_CONSUMED:** Information about resources consumed by the job (and processes). It will extend the information provided by the previous flag.
- **USER_METRICS.** Information of the metrics provided manually by users with the *setMetrics* XATI function

2.3 Input and Output XML description

2.3.1 MetricDesc

MetricDesc provides the description of a metric in the system. It will work as an input or output parameter.
We include in Table 1 and 2 an XML sample file and the corresponding xsd describing MetricDesc.

Table 1 : Sample XML MetricDesc

```
<?xml version="1.0"?>
<metricDesc
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="MetricDesc.xsd">
<name>jobId</name>
   <description>Global Identifier of the job</description>
   <type>string</type>
   <scope>Job</scope>
   <buffering>false</buffering>
</metricDesc>
```

Table 2 : XSD file for MetricDesc

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="metricDesc" type="MetricDesc"/>
    <xsd:complexType name="MetricDesc">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="type" type="TypeEnum"/>
      <xsd:element name="scope" type="ScopeEnum"/>
      <xsd:element name="buffering" type="xsd:boolean"/>
      <xsd:element name="bufferingSize" type="SizeEnum" minOccurs="0"/>
    </xsd:sequence>
    </xsd:complexType>
  <xsd:simpleType name="SizeEnum" default="SHORT">
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="SHORT"/>
      <xsd:enumeration value="MEDIUM"/>
      <xsd:enumeration value="LONG"/>
      </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="ScopeEnum">
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="Job"/>
      <xsd:enumeration value="Resource"/>
      <xsd:enumeration value="JobUnit"/>
      <xsd:enumeration value="Process"/>
     </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="TypeEnum">
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="int"/>
```

```
      <xsd:enumeration value="double"/>
      <xsd:enumeration value="char"/>
      <xsd:enumeration value="string"/>
      <xsd:enumeration value="time"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

2.3.2 Obtaining monitoring information

Obtaining monitoring information is done by *getJobsInfo* it outputs an XML file with the specified information. An XML output sample can be found on Table 3 and its associated XSD on Table 4, we can see how the buffering output is presented in the XML files in this sample.

Table 3: Sample XML output for getJobsInfo

```
<jobInfoList>
<jobInfo jobID=100000-00000-000000-00000>
<metric>
        <name>submitTime</name>
        <type>time</time>
        <value timestamp=120202020>1/1/09 10:00</value>
        </metric>
</jobInfo>
<jobInfo jobID=200000-00340-003240-00000>
<metric>
        <name>status</name>
        <type>string</time>
        <value timestamp=120202020>GRID SUBMITTED</value>
        <value timestamp=120202030>RUNNING</value>
        <value timestamp=120202040>RUNNING</value>
        <value timestamp=120202050>DONE</value>
</metric>
</jobInfo>
</jobInfoList>
```

Table 4: XSD Schema for getJobsInfo

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="jobInfoList" type="JobInfoList"/>

<xsd:complexType name="JobInfoList">
  <xsd:sequence>
   <xsd:element name="jobInfo" type="JobInfo" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="JobInfo">
<xsd:attribute name="jobID" type="xsd:string"/>
  <xsd:sequence>
    <xsd:element name="metric" type="Metric"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Metric">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="type" type="xsd:string"/>
    <xsd:element name="value" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:simpleContent>
       <xsd:extension base="xsd:string">
        <xsd:attribute name="timestamp" type="xsd:string" />
       </xsd:extension>
```

```
      </xsd:simpleContent>
     </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

3. XATI functions related with job monitoring

This section presents the complete API related with job monitoring. This definition is generic and it must be adapted to the programming language used. A convention used is that all functions have as return value an error code and output values are marked with the tag OUT.

3.1 Obtaining jobId's

```
int getJobId(int resourceID, int pid, X509Certificate userCert, OUT String jobID);
```

Most of the XATI functions receive as first parameter the jobID. The first monitoring function is the one that returns the jobID based on the resource and pid. To have access to this information, the user has to prove to have access rights providing the user certificate. It returns the jobID corresponding with the query if parameters are correct or an error otherwise.
Error codes:

- No jobID associated with the <resource, pid> requested
- Incorrect user certificate

```
int getJobsUser(X509Certificate userCert,OUT list<String> jobIds);
```

This function returns the list of all the jobIDs existing for a specific user identified with the user certificate.
Error codes:

- Incorrect user certificate

3.2 Obtaining job information

```
int getJobMetrics(String jobId, X509Certificate userCert, OUT list<MetricsDesc> metrics);
```

This function returns the list of available metrics for a specific job. It is per job because each job can have its own user defined metrics. If jobID is null, the list of default metrics is returned. Each metric is described by a complex type MetricDesc. MetricDesc has the following fields:

- **Name**. This is the name of the metric as it will appear in the xml returned by the getJobInfo XATI call. It is a string and valid characters are [A-Za-z0-9] and no spaces allowed.
- **Type**. It must be a simple type. Valid types are: int, double, char, string, time. (It can be an enum that represents the type of the metric).
- **Human readable description**. This is a string to make comprehensible the values provided by this metric.
- **Scope**: It can be job, resource, jobUnit, and process. (It can be defined as an enum.).
    - If the scope is **job**, the values of the metric will be associated to a JobID.
    - If the scope is **resource**, the information will be associated to a resourceID (communicationAddress).
    - If the scope is **jobUnit**, the information will be associated to a <jobID,resourceID>.
    - If the scope is **process**, it will be associated to a <jobID, resourceID, PID>
- **Buffering allowed** (yes/no). AEM will provide an advanced mechanism of buffering of sequences of metric values to minimize the communication between AEM and applications performing advanced monitoring. If buffering is set, AEM will allocate a buffer for this metric and subsequent modifications of the metric value will be stored in that buffer.
- **Buffer size**. In case the metric includes the buffering feature the buffer size can be specified. It can be selected from three predefined values: SHORT, MEDIUM, LONG.

Error codes:

- Incorrect jobID
- Incorrect user certificate

int getJobsInfo(list<String> jobIds, int typeOfInfo, int infoLevel, list<String> metrics,
X509Certificate userCert, OUT XMLString jobInfoList);

There is a single XATI function to obtain the information of a job. This function receives a list of jobIDs to be monitored, the level of information to be returned (JOB, PROCESS or KERNEL), and the type of information requested (BASIC, JOB_DEFINITION …), a list of specific metrics that updates the previous selection, and the user certificate. It will return an XML string with the information requested. Section 3 includes a deeper description of values of *infoLevel* and *typeOfMetrics* parameters.
Error codes:

- Incorrect jobID (some of the jobID's)
- Incorrect infoLevel
- Incorrect type of information
- Incorrect list of metrics (some of the metrics do not exist)
- Incorrect user certificate

3.3 Extending the job monitoring information

As we have said in the introduction, one of our main goals is to provide an extensible monitoring mechanism that will make possible to add new information. With this purpose, AEM API includes two functions to define new metrics and to set values for these metrics.

int addJobMetric(String jobId, MetricDesc metric, X509Certificate userCtx);

This function defines a new user metric for the specified jobID.
Error codes:

- Incorrect jobID
- Metric already exists
- Incorrect metric description
- Incorrect user certificate

int updateJobMetric(String jobId, MetricDesc metric, X509Certificate userCtx);

This function updates a previously existing user metric for the specified jobID.
Error codes:

- Incorrect jobID
- Metric does not exist
- Incorrect metric description
- Incorrect user certificate

int setMetricValue(String jobId, String metricName, communicationAddress resourceID, int pid, String value, X509Certificate userCtx);

This function sets the value for a specific metric. The metricName must be a user metric valid for this jobID. Since the metric can be associated to the job or to one specific process of the job, the function includes two parameters that will be used just in case the scope requires them: the resourceID and the pid. The resourceID (defined by a communicationAddress) is required because the pid is not an identifier in the context of a job executed in more than one node. Since the value could be from an int to a String, we have decided to pass a String and convert using the MetricDesc related to the MetricName.
Error codes:

- Incorrect jobID
- Metric does not exist
- Invalid user certificate
- Incorrect metric value

3.4 Managing the job monitoring

void enableMonitoringBuffering (String jobId, communicationAddress resourceID ,int pid, String metricName, Boolean enable, X509Certificate userCtx);

This function will enable buffering of events of the specified metric, the buffering will be associated to a jobId, user and if applicable resourceID and pid (pid is not an identifier if we are using more than one node).

- Incorrect JobId
- Invalid Metric
- Invalid user certificate

- Metric cannot provide buffering capabilities.

int jobMonitoringControl(String jobId, MonOp op, int Infolevel, X509Certificate userCtx);

Monitoring jobs is a process that can be time consuming depending on the amount of metrics. With this function an application can START, STOP or CHANGE_LEVEL of monitoring. Start or Stop options include activate/deactivate all the callbacks associated with a specific metric, and also, if defined, sets the event associated with all the job metrics.
Error codes:

- Incorrect JobId
- Invalid Parameters
- Invalid user certificate

int addMonitoringCallback(String jobId, metricEvent event, callbackFunction callBack, X509Certificate userCtx);

Defines a function associated with a metric event. Monitoring metrics events can be events generated by the jobMonitoringControl such as START or STOP the monitoring, or it could be also that a metric has reached a certain value. In this way, external monitoring tools can react to certain situations letting the complexity of the action to be taken to the callback function (we will provide the ability to have multiple callbacks per metric).
A metricEvent is a complex type that describes the event that launches the callback. Based on the type of metrics, we can define the different events:

- **Time metrics**: It should be easy to control that a certain time metric reaches a certain values by setting a cron event. However, it is not easy to provide synchronous events, so we will provide events such as GREATHER_THAN
- **Status**: It should be easy to detect status changes and notify the user. We could support events such as status CHANGE_TO.
- **User defined metrics:** User metrics are set using the setMetricValue, so changes in value are automatically detected. We will support EQUAL_TO, GREATHER_THAN, LESS_THAN, CHANGE_TO.
- **Resource consumption**: We will provide support to process and memory accounting. We will support GREATHER_THAN.
- **Metrics with buffering**: In that case, it is interesting to detect situations such as the buffer is full: FULL_BUFFER

Based on this definition, a metricEvent include:

- **Event**: START/STOP/CHANGE_LEVEL/CHANGE_TO/GREATER_THAN, EQUAL_TO, LESS_THAN, FULL_BUFFER.
- **Metric Name** (if CHANGE_TO/GREATER_THAN, EQUAL_TO, LESS_THAN events)
- **Value**: new info level in case CHANGE_LEVEL is selected or value if CHANGE_TO/GREATER_THAN, EQUAL_TO, LESS_THAN are selected.

Callbacks are offered through DIXI infrastructure.

Error codes:

- Invalid user certificate
- Invalid Metric Name

4. System provided metrics

AEM automatically provides some job basic metrics. In this section we describe these metrics that are available for each existing job in the system. Buffering can be enabled or disabled on every metric.

- Scope=Job
  - *Job ID*
    - Name: jobId
    - Description: Global identifier of the job
    - Type: String
    - Buffering:no
  - *User DN*
    - Name: userDN
    - Description: User Distinguish Name of the owner of the job
    - Type:String
    - Buffering:no
  - *Status*
    - Name:status
    - Description: Status of the job (RUNNING, DONE, …)

        - Type:string
        - Buffering:no
    - *Submission time*
        - Name:submitTime
        - Description: Time at which the job was submitted
        - Type:time
        - Buffering:no
    - *Submitted from*
        - Name:submitNode
        - Description: Communication address (IP : Port) where the job was submitted from
        - Multiplicity:one
        - Type:address
        - Buffering:no
    - *JSDL submitted*
        - Name:jsdl
        - Description: Current description of the job in terms of JSDL. It could be the original JSDl submitted or not.
        - Type:String
        - Buffering:no
    - *Exit status*
        - Name:exitCode
        - Description: If the job is finished, exit value provided in jobExit. Undefined value otherwise.
        - Type:int
        - Buffering:no
    - *Reservation ID*
        - Name:String
        - Description: Global identifier for a reservation allocated to that job
        - Type: String
        - Buffering:no
- Scope=Resource
    - *Resource Address*
        - Name:resourceID
        - Description:Identifier
        - Type:address
        - Buffering:no
- Scope=jobUnit
    - *Resource Address*
        - Name:resourceID
        - Description: Communication address (IP : Port) where that part of the job is running
        - Type:address
        - Buffering:no
- Scope=Process
    - *Process Identifier*
        - Name: PID
        - Description: Process identifier local to this resource
        - Type:int
        - Buffering:no
    - *Status*
        - Name: status
        - Description: Process status
        - Type.String
        - Buffering:no
    - *User time*
        - Name.userTime
        - Description: Time spent by this process in user mode
        - Type:time
        - Buffering:no
    - *System time*
        - Name:systemTime
        - Description: Time spent by this process in system mode

- Type:time
- Buffering:no

5. XATI commands related with job monitoring

This section presents the xcommands, command line programs, to be used by most of the users to get job information. We have proposed a set of commands with the same look and feel as Linux commands such as ps. These new commands are:

xps [–j jobID] [-a] [-d level] [-b –jd –ra –rc -um] –c cred_file_path

It prints the information related to either the specified jobID (-j jobID) or all the jobs from the user that executes the command (-a option). The –d option selects the level of information to be shown. The user can specify if he/she wants information BASIC (-b), JOB_DEFINITION (-jd). RESOURCE_ALLOCATED (-ra), RESOURCE_CONSUMED (-RC) and USER_METRICS (-um). The user has also to provide the file path of the user credentials.

xtrace [-j jobID] [-i sec] [-b –jd –ra –rc -um] [-t traceFileName] [-c cred_file_path] [submission]

It collects periodic information about the specified job every sec seconds (default=5 seconds). The user can specify if he/she wants information BASIC, etc. Additionally, one can define a trace file name where information will be also saved; otherwise the standard output will be used. If a submission is provided, the *xtrace* command will submit the job defined by submission, in the same way as the *xsub* command, and will trace the job execution from the beginning.

6. Changes in the architecture

We need to store all these metrics in either *jobManager* or *ExecManager*. This should not be a problem. We also need some way to manage pending events for callbacks. Depending on how we define metrics, we could have a list of pending events per job and/or job_unit (in *ExecMng*).
Places to check pending events will be:

- Each call to setMetricValue
- Based on cron events for time
- Any change in status of countable events such as process creation

We need to include:

- List of pending events
- Validation points
- Callback mechanism, using DIXI callback as base.

We also need to store buffers (circular buffers) so we can store values when buffering is on. We will store them on ExecMng (per job_unit).

7. Conclusion

With this monitoring proposal, applications can collect from jobs so simple information such as the status or can externally add advanced monitoring information. This information can be generated/gathered by the jobs themselves or by third party applications. The mechanism proposed is quite flexible to be used by the users with command line programs, such as xps, or by other XtreemOS components that need to monitor job activity. Finally, the interface proposed includes features that we will be used by SAGA. In the next months we will work implementing the interface described. One of the key points of the implementation is the concept of callback that we expect to solve using DIXI.

8. References

S. Browne, J. D. (2000). A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications , 14* (3).

[D3.3.1] Toni Cortes, *Requirements and specification of XtreemOS services for job execution management, D3.3.1 , 2007*

[D3.3.3-4] Toni Cortes, Julita Corbalan, Gregor Pipan, *Basic Services for application submission,control and checkpointing. Basic services for resource selection, allocation and monitoring D3.3.3-3.3.4, 2007*

[D3.1.7] Thilo Kielmann*, Revised System Architecture, D3.1.7, pending (M30)*