Project no. IST-033576

XtreemOS

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

AEM Prototype
Advanced features
D.3.3.7

Due date of deliverable: March 30th,2010
Actual submission date: March 30th,2010

Start date of project: June 1st 2006
Type: Deliverable
WP number:3.3

Name of responsible:Toni Cortes
Editor & editor's address:Ramon Nou
Barcelona Supercomputing Center

Version 1.0/ Last edited by Ramon Nou/ Date 10/03/22

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:

Revision history:

Version	Date	Authors	Institution	Sections Affected / Comments
0.01	10/02/04	Ramon Nou	BSC	Initial Draft
	10/02/09	John Mehnert-Spahn	UDUS	Checkpointing
	10/02/15	Primož Hadalin	XLAB	MonMng AuditMng
0.50	10/02/17	Ramon Nou	BSC	Several adds
0.80	10/02/24	Jacobo Giralt	BSC	Vnodes & monitoring
0.99	10/02/25	Toni Cortes	BSC	General Overview
0.995	10/02/25	Matej Artač	XLAB	Reservation management
0.996	10/03/18	Marjan Šterk	XLAB	Reservation management (co-allocation)
1.0	10/03/22	Ramon Nou	BSC	Finishing touches

Reviewers
Noé Gallego (TID) , Franz Hauck (ULM)

Tasks related to this deliverable

Task No.	Task description	Partners involved °
3.3.5	GRID checkpointing and migration	UDUS*,INRIA
3.3.7	Advanced services to control jobs	BSC*
3.3.8	Co-allocation and resource negotiation	XLAB*, BSC
3.3.9	Advanced Job Monitoring	BSC*, XLAB
3.3.10	Fault tolerant services	BSC*, ULM
3.3.11	Interactive jobs	INRIA*, BSC, XLAB
3.3.12	Auditing	XLAB*, BSC

* Task leader

Executive Summary

AEM (Application Execution Manager, the instance responsible for Job/Resource management in XtreemOS) is evolving and adding more and more features. In this document, we present the current prototype of AEM as in M46. Specifically, we will present the improvements in the following components: Checkpointing, Job monitoring, Monitoring Manager, Auditing Manager, and Reservations.

Furthermore, we will also present improvements in the global functionality of the AEM such as new schedulers that are data conscious or mechanisms to make these services fault tolerant implementing them on top of virtual nodes.

Regarding interfaces, we have proposed new XOS-specific JSDL extensions for Checkpointer, SchedFS, Certificates and Sched_Hints. In addition, we have improved the integration of bash and procps (ps, kill , wait) to support jobs. Several wrappers for C-XATI on monitoring and reservations to make the SAGA support easier. Basic commands to support interactive jobs. C-XATI and XATI clients are now pure clients to make mobile devices implementation easier.

In this document we describe all these features in detail from all points of view: functionality, use cases, architecture, implementation, and user manual.

Table of contents

1 Introduction

AEM, Application Execution Management, provides several capabilities related to the job execution and the resource management inside XtreemOS.

The prototype includes features achieved by month 46 of the project:

- **Checkpointing**: Support for two different checkpointers: LinuxSSI and BLCR.
- **Job monitoring**: On the monitoring side, we provide the advanced monitoring infrastructure. User metrics, callbacks, system metrics, different levels of information are integrated in the prototype.
- MonMng - **Monitoring Manager**, the service that collects events and metrics from the services, developed partially for WP3.3 and WP3.5. In WP3.3 it provides the mechanism to collect information from your advanced job monitor, aggregate them and send them to anyone who subscribes to them.
- AuditingMng - the **Auditing Manager** service, also developed partially for WP3.5 as well. In WP3.3, it collects information from MonMng and provides history storage and interfaces for generating reports used in billing, resource utility analysis, etc.
- **Reservations** : AEM includes reservations. We can ask for advanced reservations of resources, and use them in several jobs and processes. It uses JobDirectory for distribution of the core service.
- **Virtual Nodes Support**, we added several layers to AEM and DIXI to support Virtual Nodes. This is still under development.
- **Schedulers**: Implemented several schedulers, one of them is using XtreemFS location information to schedule jobs near the used files. An extension to the JSDL is proposed for this.
- Support for new XOS-specific JSDL extensions: Checkpointer, SchedFS, Certificates and Sched_Hints
- **User and client side**: Integration of bash and procps (ps, kill , wait) to support jobs. Several wrappers for C-XATI on monitoring and reservations to make the SAGA support easier. Basic commands to support interactive jobs. C-XATI and XATI clients are now pure clients to make mobile devices implementation easier.
- Several bug fixes and performance improvements.

2 Checkpointing

2.1 Overview

The grid checkpointing architecture [XGCA], see Figure 1, is able to checkpoint and restart jobs consisting of one or more job units. Prototypes of coordinated and uncoordinated checkpointing have

been improved. A so-called channel flushing component has been developed and integrated taking care of in-transit messages during a coordinated checkpoint to avoid lost and orphan messages at restart time. Channel flushing can be used even if a job is executed on nodes equipped with different heterogeneous checkpointing packages[CMCHE].

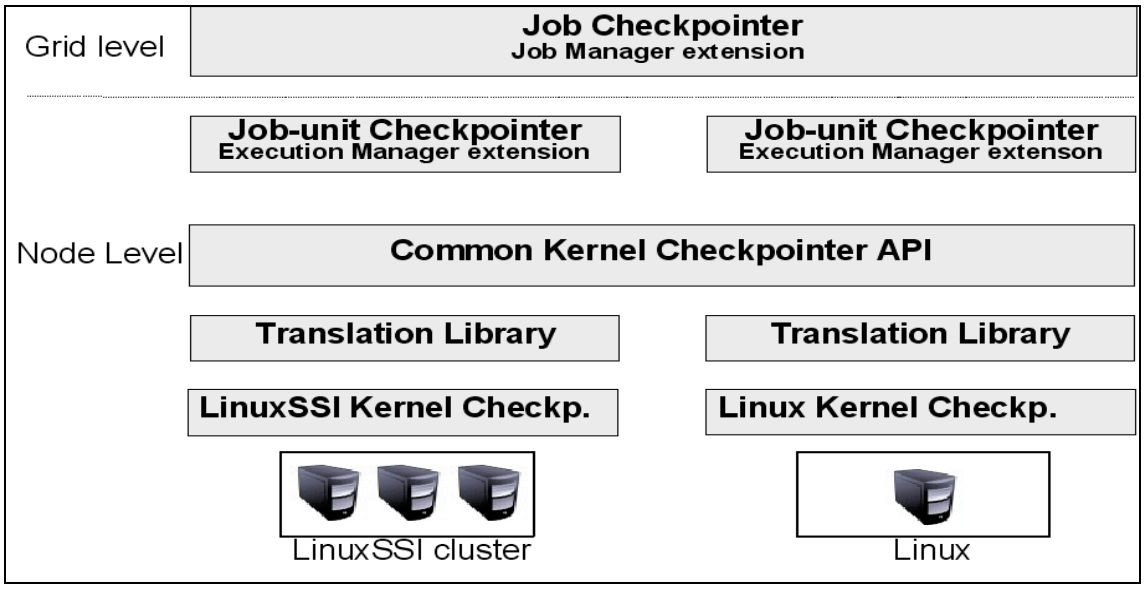


Figure 1

2.2 Use Cases

XtreemOS job-checkpointing and restart can be used to realise job fault tolerance.

2.3 Architecture

The job checkpointer (extension of job manager) realises job checkpoint and restart. It delegates job unit checkpoint and restart commands to job unit checkpointer (extension of execution manger). The job unit checkpointer in turn calls an underlying kernel checkpointer via a so-called common kernel checkpointer API. The API is implemented by a kernel checkpointer-bound translation library. More information about the architecture can be found here [XGCA]

2.4 Implementation

The CRJobMng and CRExecMng java classes equal the job and job unit checkpointer. Coordinated checkpoints can be triggered from the console, while uncoordinated checkpoints get triggered by the application. Therefore the application must be linked against the so-called job unit checkpoint API (JUCAPI) library.
SSI callbacks have been integrated allowing executing user-defined actions immediately before and after a checkpoint as well as after a restart. Furthermore, SSI incremental checkpointing has been integrated allowing to reduces checkpointing overhead (duration and disk space) if a program modifies only small parts of its address space after a checkpoint.
The channel flushing protocol is executed in the context of pre- and post checkpoint and restart callbacks, implemented for BLCR [BLCR] and SSI.

3 Job Monitoring

3.1 Overview

In any operating system we have a set of operations, commands and even programming interfaces that provides the user with the ability to know what is happening on the system. XtreemOS provides them through the job monitoring interface.

3.2 Use Cases

Job Monitoring provides:

- Typical information associated to jobs such as execution time
- Mechanism to limit the type and the granularity of information collected
- Mechanism to easily add new information to the generated by the system (user metrics)
- Mechanism to be notified when certain monitoring events fire (callbacks)

3.3 Architecture

The architecture of job monitoring is distributed among the **JobMng** and **ExecMng** services inside AEM. The user interfaces is available through JobMng,, though the information is stored in both depending on its granularity: Job related metrics are stored on the JobMng and process related ones in the ExecMng. This way storage is as close as possible to the source of the data, thus reducing the cost of setting values. This is represented in the following figure (Figure 2).

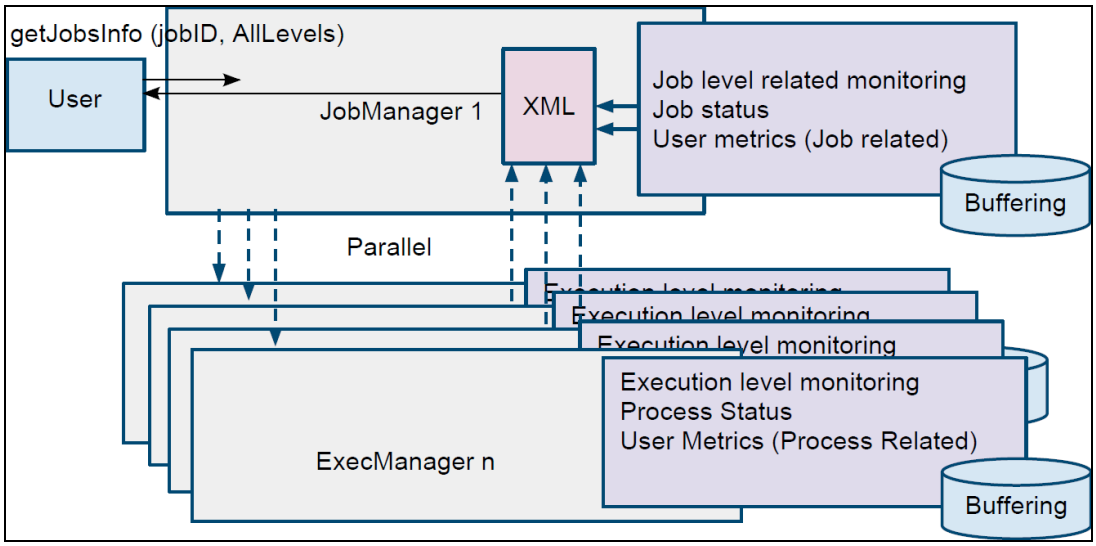


Figure 2

3.4 Implementation

The methods implemented and exported to the user are the next ones:

getJobsInfo(jobIds, flags, infoLevel, metrics)
Get XML with requested information and detail level.

getJobMetrics(jobId)
Returns the list of available metrics for a specific job.

setMetricValue(jobId, metricName, resourceID, pid, value)
Sets the value of a Metric.

setMonitorBuffering(jobId, metricName, resourceID, pid, flags)
Switches on and off buffering for the specified metric.

addJobMetric(jobId, MetricsDesc)
Adds a new user defined metric to the job.

removeJobMetric(jobId, metricName)
Removes a user defined metric from the job.

addMonitoringCallback(jobId, metricEvent)
Adds a monitoring callback expecting an event to fire.

These operations are all accessed through **JobMng**, though this service may contact others in order to gather all the information requested. In order for the user to limit this effect and thus the associated cost, the infoLevel parameter allows three values: JOB, PROCESS and KERNEL. The first one restricts the information source to the JobMng, the second one to the ExecMng and the latter may even allow more costly operations on each processing node like those that need interaction with kernel tracing modules.

Some metrics allow buffering; this means that the system might store multiple values for them avoiding that the user misses any value after subsequent updates. This also potentially reduces the frequency in which the user requests monitoring information. Combined with callbacks, the information request can be programmed to be effective on system demand, when the desired buffers are close to be full.

The monitoring infrastructure is extensible by the user through the *addJobMetric* method. Once added, values can be assigned to it invoking *setMetricValue*. Permission to do so is certificate based; system metrics can not be set by the user.

The format exchanged in all those operations is XML based, though helper classes and tools are provided in both XATI and XATICA interfaces to ease its management. The XML format representing the general monitoring output looks as shown in Figure 3. It is a list of jobInfo elements; each of those represents

hierarchically the state of a job. JobInfo elements have a jobId attribute and a set of metrics associated to it. Then for each process inside the job, there is a jobUnit element. This element has a resourceId attribute which represents the actual node where the process is being run, and a set of associated metrics [D335].

```
<jobInfoList>

        <jobInfo jobId='10d32332-d243-475f-93df-92cf81702216'>
                <metric>
                        <name>jobID</name>
                        <type>string</type>
                        <value timestamp='1255623770000'>10d32332-d243-475f-93df-92cf81702216
                        </value>
                </metric>
                <metric>
                        <name>userDN</name>
                        <type>string</type>
                        <value timestamp='1255623770000'>8903e091-efc4-4c97-9053-d784ccc21b06
                        </value>
                </metric>
                <metric>
                        <name>VO</name>
                        <type>string</type>
                        <value timestamp='1255623770001'>fcc1d2eb-b534-44d3-8daf-3b94c57b5035
                        </value>
                </metric>
                <metric>
                        <name>jobStatus</name>
                        <type>string</type>
                        <value timestamp='1255623770001'>Running</value>
                </metric>
                <metric>
                        <name>submitTime</name>
                        <type>string</type>
                        <value timestamp='1255623770001'>Thu Oct 15 18:22:44 CEST 2009</value>
                </metric>
                <jobUnitInfo resourceId=':://84.88.50.161:60000(84.88.50.161)'>
                        <metric>
                                <name>runNode</name>
                                <type>string</type>
                                <value timestamp='1255623770002'>84.88.50.161:60000</value>
                        </metric>
                        <procInfo pid='19601'>
                                <metric>
                                        <name>PID</name>
                                        <type>int</type>
                                        <value timestamp='1255623770005'>19601</value>
                                </metric>
                                <metric>
                                        <name>processStatus</name>
                                        <type>string</type>
                                        <value timestamp='1255623770005'>S</value>
                                </metric>
                                <metric>
                                        <name>userTime</name>
                                        <type>time</type>
                                        <value timestamp='1255623770006'>00:00.00</value>
                                </metric>
                                <metric>
                                        <name>systemTime</name>
                                        <type>time</type>
                                        <value timestamp='1255623770006'>00:00.00</value>
                                </metric>
                        </procInfo>
                        <procInfo pid='19602'>
                <!-- Output shortened for example purposes -->
                        </procInfo>
                </jobUnitInfo>
        </jobInfo>
```

The *JobInfoList* class (Figure 4) is the XATI helper to parse monitoring information; it's constructed with the output from *getJobsInfo* and optionally a path to the XSD Schema to validate against it. That schema is provided in an XtreemOS system as well. *JobInfoList* methods reflect the hierarchy explained before

with the XML sample and return a *MetricValue* object. This one is formed by a set of time and value pairs if the metric is buffered, or just one pair if not.

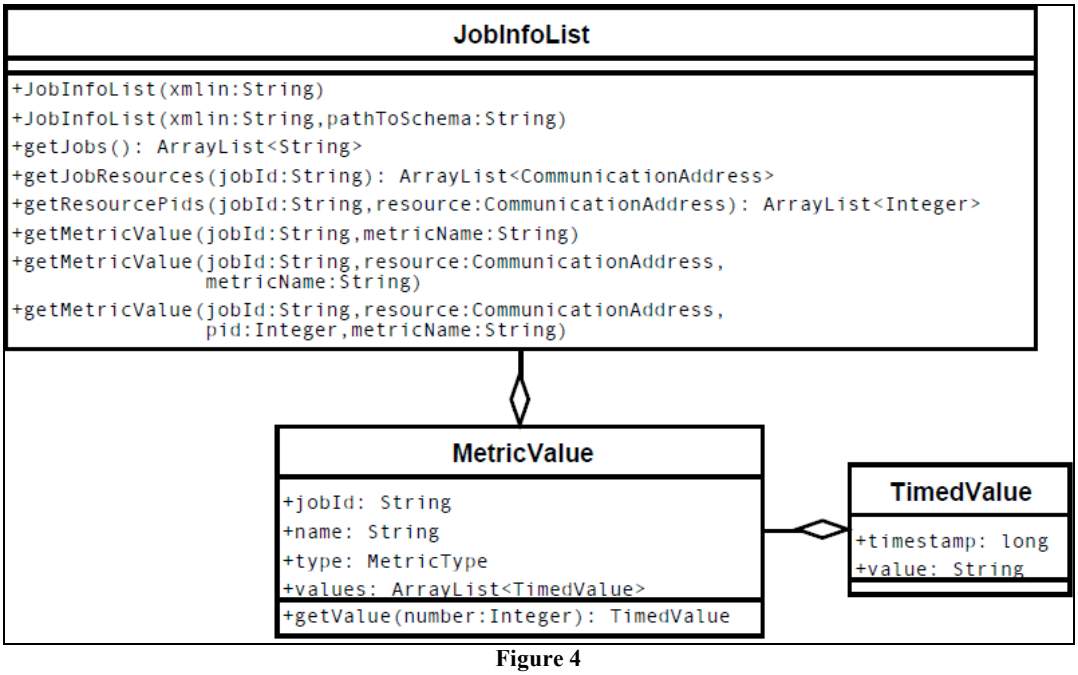


Figure 4

4 Monitoring Manager

4.1 Overview

The Monitoring manager provides monitoring of basically anything that can be measured in XtreemOS [D335]. It collects monitoring data from various sources and stores it for a period of time.
Interested parties define monitoring rules which describe what to monitor. When conditions of the monitoring rule are met, a notification is issued.
A particular monitoring rule is indentified by monitoring rule name to which interested parties subscribe to receive notifications.

4.2 Use Cases

Monitoring Manager provides:

- saving events
- saving metrics
- setting monitoring rules
- subscribing to monitoring notifications
- defining aggregated metrics

4.3 Architecture

As shown on figure, Figure 5, various services send events and metrics to the monitoring manager. Event is an occurrence with no value, whereas metric holds a value, value type and value unit. Both event and metric also specify measurement context or domains: VO, user group, user, job, resource. One or more of these domains must be provided for better presentation of the measurement.

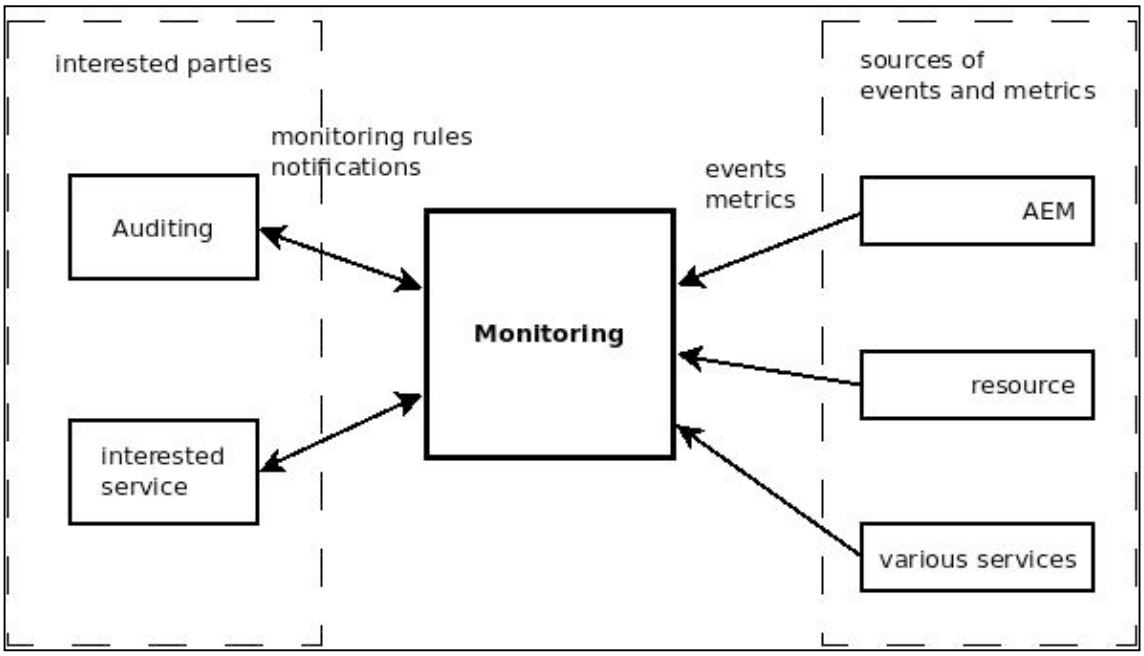
Examples of events are:

- job started
- job ended

- job failed
- rca request
- VO created
- user certificate not valid

Examples of metrics:

- cpu utilization - 80 %
- free memory - 1395772 KB
- free disk - 100.5 GB
- jobs running - 5



4.4 Implementation

Methods exposed to the user:

void addNotification(String monitoringRule, String monitoringRuleName)
Method adds monitoring rule and sets monitoring rule name to which interested parties can subscribe and get notified.

String subscribe(String monitoringRuleName, ICallback)
Method used for subscribing to specified monitoring rule to get notified by callback when monitoring rule conditions are met. Id of the subscription is returned which is used for possible later cancelation.

void unsubscribe(String subscriptionId)
Used for unsubscribing from monitoring rule notifications.

MetricValue query(String monitoringRule)
Performs query that is based on monitoring rule.

void saveEvent(Event event)
Method used by various services to send event to monitoring manager.

void saveDoubleMetric(DoubleMetric doubleMetric)
Method used by various services to send metric of type double to monitoring.

void saveFloatMetric(FloatMetric floatMetric)
Method used by various services to send metric of type float to monitoring.

void saveIntegerMetric(IntegerMetric integerMetric)
Method used by various services to send metric of type integer to monitoring.

void saveStringMetric(StringMetric stringMetric)
Method used by various services to send metric of type string to monitoring.

Monitoring manager uses internal database (HSQLDB) to store metrics and events for a brief period of time. To permanently store monitoring data, Auditing Manager is used. Hibernate is used on top of the database for mapping between Java objects and database tables. Event, DoubleMetric, FloatMetric, IntegerMetric and StringMetric classes are currently supported to be mapped.

A user of monitoring manager can get notified when certain events occur or various metrics' values are in certain range. For example one might want to be notified when job on certain VO is started or when CPU on certain resource reaches 90 percent of utilization.

To describe conditions when a user wants to be notified, monitoring rules are used. Monitoring rules parser was generated using CUP LALR parser generator. Monitoring rules have a form as described below:

@[{*domains*}]event_name | metric *operator* reference_value

where *domains* represents one or more domains: VO, user group, user, job and resource. *Operator* is one of the following logical operators:

- < - less
- <= - less or equal
- == - equal
- > - more
- >= - more or equal

Here is an example of monitoring rule to notify when CPU utilization reaches 90 percent on a specific resource under specific VO:

@{vo=[fd3fd3-df4s-34r3-fd34],resource=[IP=[192.168.0.10]]}cpu_user>=90

When a monitoring rule is defined, a name to monitoring rule is also assigned. Because of publish/subscribe principle several interested users can subscribe to specific monitoring rule to get notified. Subscribers provide callback which gets triggered when conditions of the monitoring rule are met.

Each of classes used by monitoring to represent metric or event has a string representation. String to represent an event class has the following form:

event_name;VO;user_group;user;job;resource

Metric classes (DoubleMetric, FloatMetric, IntegerMetric, StringMetric) have the following form:

metric_name;value;value_unit;VO;user_group;user;job;resource

These string representations are used when referencing event and metric classes from XATICA.

5 Auditing Manager

5.1 Overview

Main purpose of the Auditing manager is to permanently store monitoring data received from the Monitoring manager. Data archived in a history database can be later analyzed and used for generating reports.

Users of Auditing manager can query history database using Hibernate query language. Because Hibernate is used, many relational databases can used as history database.

5.2 Use Cases

Auditing manager provides:

- defining archiving rules
- permanently storing monitoring data to history database
- history database querying
- support for various databases

5.3 Architecture

Auditing manager (Figure 6) defines several archiving rules at startup and begins to archive monitoring data. New archiving rules can be added at run-time.

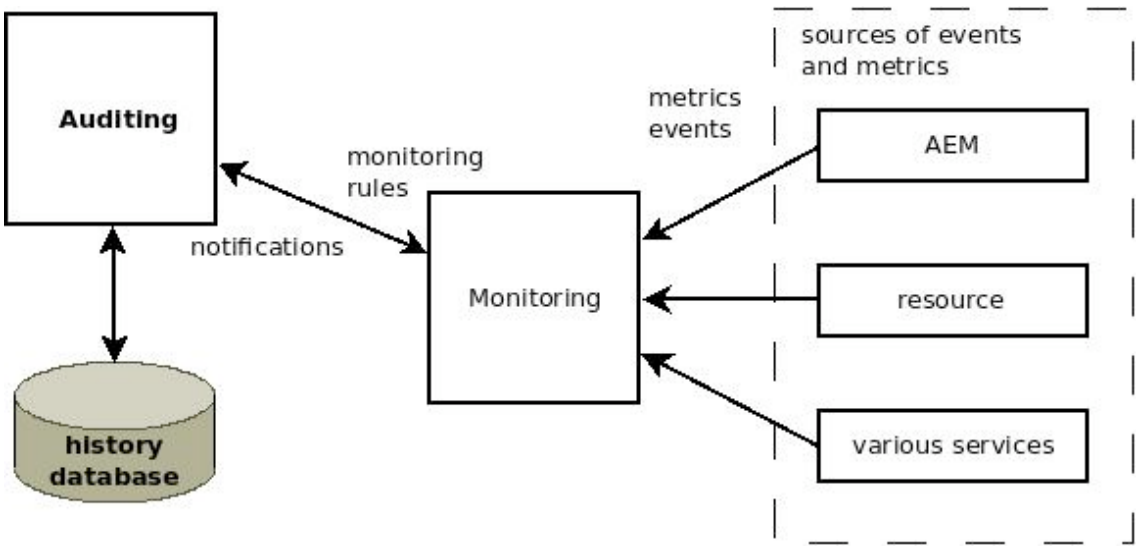


Figure 6

5.4 Implementation

String addArchiveRule(String monitoringRule)
Method for adding archive. Auditing will archive the data described in the monitoring rule. Archive rule ID is returned which is used for later cancelation.

void cancelArchiveRule(String archiveRuleId)
Cancels archive rule and stops archiving data.

ArrayList query(String query)
Queries history database and returns a list of results. Hibernate query language (HQL) is used as query string.

Auditing manager relies on Monitoring manager to collect monitoring data which is then permanently stored into history database. Auditing manager reads monitoring rules from configuration file which are then added to Monitoring manager. By subscribing to these monitoring rules, Auditing manager gets notified and received data is stored into history database.

Auditing Manager uses Hibernate on top of the database to map event and metrics classes to database tables. HSQLDB database is used to store the data, although theoretically any relational database can be used. For testing purposes MySQL database was also used and successfully replaced HSQLDB database. Because of Hibernate only configuration file had to be reconfigured in order to change the database.

The user queries history database using Hibernate Query Language (HQL), although other query languages can be used depending on Hibernate configuration. In order for the user to query the database one must know the structure of the database or the classes that get mapped into the database tables. Flexibility of the queries allows the user to retrieve single record, a list of records or performs aggregating operations like average, sum, count, etc.

6 Reservation Manager

The Reservation Manager was already included in the last prototype. However we updated and included several wrappers to make easier (seamless) the integration with XOSAGA.
The architecture mimics JobMng and ExecMng with ReservationMng and AllocMng respectively. We can reserve resources (CPU/MEM/etc…) in a shared or exclusive way.

6.1 Overview

The reservation management consists of the core-level service Reservation Manager and node-based Allocation Managers. We developed this subsystem in the previous release already, mimicking the architecture of JobMng and ExecMng, respectively. The services enable the user to reserve the resources, such as CPU and memory, on each node, consisting of a complex set of time-table slots. The user can request individual time-table slots to exclusively reserve a particular resource, or let it be shared with other users. Each time-table is uniquely identified with a reservation ID, which, in turn, can be associated with a particular job. In fact, with the previous release, any job that would run in a VO has to have a reservation first.
The basic functionality of the reservation manager system remained unchanged in this release. There were certain changes, however, that provided secondary functionality. The first important change involved the scalability, because the core-level Reservation Manager now leverages SRDS for reservation directory (similar to JobMng's use of the Job Directory). Further, we provided several class wrappers which make a seamless integration with the XOSAGA. Finally, we extended the standard JSDL to express the job lifetime requirements and the requested co-allocation.

6.2 Use Cases

The new use cases include:

i) Creating a reservation that enables periodic or time-limited execution of the job, e.g., every Friday from 11pm to 10pm.
ii) Reserving a set of nodes that, in the sum, follow the resource requirements (e.g., needing 3-5 nodes that, together, have 5 GB of RAM available and 8 CPU cores).

6.3 Architecture

The architecture of the reservation management remained mostly unchanged from the previous release [D336]. The new element, shown also on the Figure 7, is the Reservation Directory which connects to the SRDS [D3213]. With this addition, the Reservation Manager can run on multiple core nodes without running into reservation information duplication.

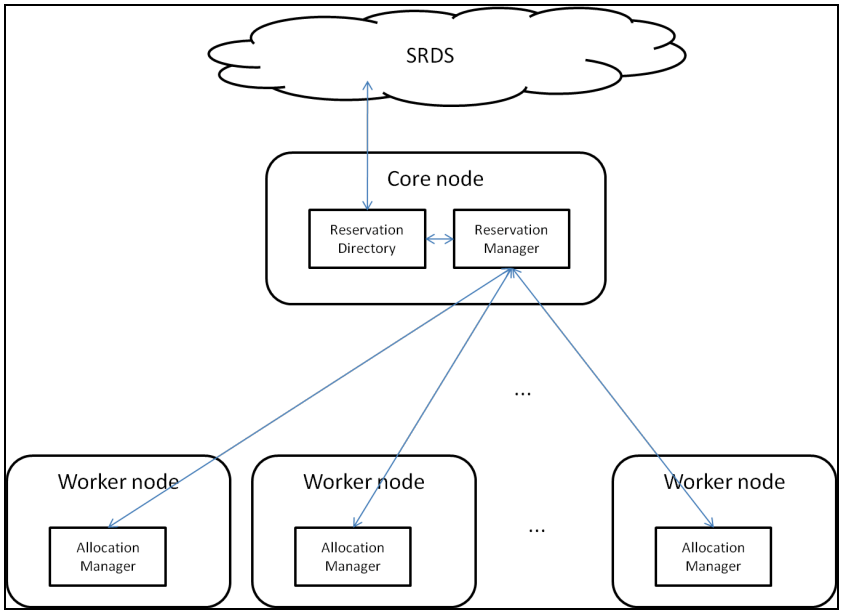


Figure 7

6.4 Implementation

The changes explained in the previous subsection involved intrinsic changes without any impact on the API. There are, however, also changes in terms of the JSDL extensions that provide the ability to express co-allocation requirements or the job life-time requirements. The following syntax expresses the requested lifetime of the job:

```
<LifeTime>
    <StartTime> datetime </StartTime>
    <ExecutionTime> time </ExecutionTime>
    <Constraints>
          <Constraint>
                <DayOfWeek> dayofweek </DayOfWeek>
                <TimeInterval>
                      <Start> time </Start>
                      <End> time </End>
                </TimeInterval>
          </Constraint>
    </Constraints>
</LifeTime>
```

The meaning and cardinality of the tags are as follows:

- **LifeTime** – a complex structure expressing the life time requirements and constraints. Zero or one can occur in a document. If it is missing, the system can select any values.
- **StartTime** – expresses the time at or after which the job should start. Zero or Exactly one should occur in the tag. If it is omitted, default is "anytime".
- **ExecutionTime** – the time in seconds expressing the expected execution time.
- **Constraints** – a complex tag expressing additional temporal constrants. Zero or one occurence of the tag is possible.
    - **Constraint** – a complex tag expressing additional constraints. They are composed of the days of week when the job is allowed to run, and the time intervals within those

days when the job is allowed to run. There should be one or more occurrences of the tag.

- **DayOfWeek** – a numerical representation (0-6, 0 representing a Monday, and 6 representing a Sunday) of the day of the week when the job is allowed to run. Zero or more occurrences of the tag are expected. Zero occurrences mean the job can run on *any* day of the week. Multiple occurrences mean that the job can run on *each* day of week listed in the tag.

- **TimeInterval** – composed of **Start** tag and an **End** tag, which represent the start and the end time of the interval, respectively. Within this interval, the job is allowed to run, but only on the days of week specified by the **DayOfWeek** tag on the same level as the **TimeInterval**. Zero or more occurrences of the tag are expected. If zero tags occur, the system assumes that the job can run on the whole day. If multiple tags occur, the job can run on all of the provided intervals.

To enable the co-allocation of multiple nodes that, in sum, provide the required quantities of resources, JSDL tags **TotalResourceCount**, **TotalCPUCount**, **TotalPhysicalMemory**, **TotalVirtualMemory**, and **TotalDiskSpace** are used for this purpose, thus no JSDL extension is required. simply include the number of resource nodes that will share the job within the `Resources` tag of the JSDL:

```
<CoAllocatedNodes>
        <Exact> nodecount </Exact>
</CoAllocatedNodes>
```

The `CoAllocatedNodes` tag should appear at most once within the JSDL. The `Exact` tag nested within should appear exactly once and contain an integer of the requested co-allocated resource nodes.

7 Virtual Nodes

7.1 Overview

Virtual Nodes is a component that is currently in phase of integration into AEM. With it, fault tolerance is added to system in the form of active replication. It was described on deliverable [D325]. Application Execution Management state is distributed among many services and nodes, but only some parts of that state are so critical that its loss could severely damage the Grid behaviour. This is the case with the JobMng; without any form of fault tolerance, a problem in one node running this service would lead to a situation where all the jobs created on it are no longer accessible for any type of management operation.

The other service candidate for replication is the ReservationMng; analogue to the JobMng, some reservations could remain unmanageable after a crash on it.

From the fault tolerance perspective, resource-node services are not worth being replicated. They store data that is exclusively relevant to the node they run on. If the node fails, and thus the job does as well, this data is of no use for any other node. Thus, replicating its data would not represent any added value.

7.2 Use Cases

After configuring Virtual Nodes for AEM it should be possible that:

iii) A core node forming part of a virtual node might fail, and then all requests targeting it will be redirected to the other replicas in the virtual node. This only applies for core services.

iv) The Virtual Organization administrator should be able to create, modify and destroy Virtual Node replicas through XATI/XATICA and a new service: VNodeMng.

7.3 Architecture

The integration of Virtual Nodes with AEM has several parts. On one hand, AEM architecture is a set of services running on top of DIXI [D3217], this means it follows an asynchronous communication model. The Virtual Nodes library was designed with synchronous models in mind and so are its implementations, like the ones described in [D3214] for Java RMI.

For that reason, both DIXI and Virtual Nodes needed some redesign in order to make the integration possible. In the DIXI side most of the changes targeted the identification of a request flow. Those were the added requisites:

i) Addition of a tracker field to DIXI Service Messages that links all the messages exchanged by involved services during a user request.
ii) Enforce the use of callId filed to locally identify requests and replies outside the service stage in Service Messages.
iii) Enforce a consistent use of contexts and callbacks in intra-service invocations.

AEM is distributed in several services, meaning that its state is also distributed. An analysis was made in order to determine which parts of the state need to be replicated in order to guarantee a certain level of fault tolerance. Preliminary conclusions were shown in D3.2.14. [D3214]

One of the objectives of the integration, given the advanced phase of the project, is keeping it as decoupled and optional as possible. A new layer is added between the services and the DIXI's message bus that links the AEM and Virtual Nodes logic. This layer, that actually supersedes the message bus when enabled, is composed by two entities that are responsible for the administration and client parts respectively. On top of that there is a VNode stage per replica in the XOSD. Between client and server parts there is group communication to all the replicas. This is shown in the following figure (Figure 8).

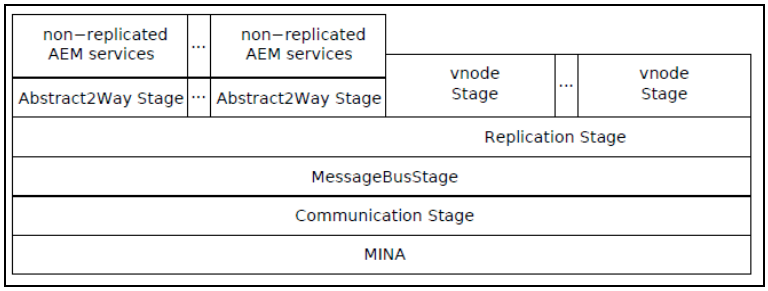


Figure 8

Finally a new service is also added to the XOSD, the VNodeMng, which acts as a gateway for the Virtual Organization administrator to configure Virtual Nodes. As previously stated, that actions are actually performed at the new layer on the message bus, but the user interface is exported through this service to XATI and XATICA.

7.4 Implementation

The changes involve service data structures like the Context and ServiceMessage ones and the addition of a new stage between the Message Bus and the replicated services. DIXI templates for service stubs code generation have been also modified to handle these new fields.

The addresses used for communication (*CommunicationAddress* class, Figure 9) have been generalized in order to support different kinds of it: The normal one, which was always used until now, and a new one (*VNodeAddress*), which packages a set of addresses (*JustCA*) representing all the replicas in a virtual node and a special Id.

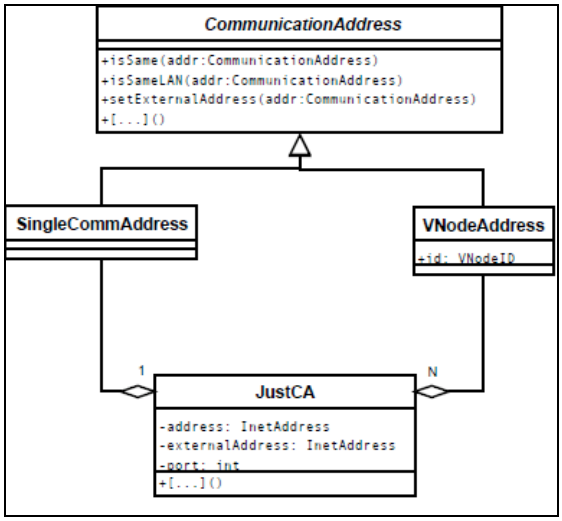


Figure 9

Job Management requests were already distributed among core nodes on a job basis depending on the node used at creation. If a job creation request arrives at a node that has Virtual Nodes enabled, the create operation will be forwarded to all its replicas and the address that represents them will be uploaded to the Job Directory. With these new type of addresses inserted into the Job Directory, the redirection functionality is extended to the use of Virtual Nodes in a transparent way to the user.

8 Schedulers

Schedulers select where a job or process should be run from the list of resources reserved by the user. We included 4 schedulers in the prototype.

8.1 Overview

The schedulers are available in SchedulerSvc and in SchedFS. Currently SchedFS is under test.
We have 3 main schedulers, Random, Round Robin and Least used. SchedFS is used only when the JSDL have information about the files used.

8.2 Use Cases

Schedulers provides:

- A set of schedulers to be selected by the administrator.
- Scheduling Hints for jobs (ONE_PER_NODE).
- Scheduling Hints to collaborate with XtreemFS node location service.

8.3 Architecture

SchedFS scheduler analyzes the JSDL for files information. If the JSDL file extension is found it tries to gather (using XtreemFS Vivaldi [D345]) files location. Then the location information is added to the JSDL and resources meeting the criteria are found using SRDS.
SchedFS tries to find nearby resources to execute the processes near the files. To do so it asks for resources inside a square centered in a coordinate and with a specified side_length. To do this ResMng modifies inflight the JSDL adding the Vivaldi Type and sends to GetResource at SRDS. This way the interface remains unchanged. However we think a new interface could benefit the advanced user, seeking and reserving resources manually.
SchedFS also stores node coordinates into SRDS, because XtreemFS cannot speak directly with SRDS.

The rest of schedulers are implemented on SchedulerSvc (separated from JobMng) and filter /select a suitable resource from the list of resources reserved.

Finally scheduling hints are implemented

8.4 Implementation

The implementation is done in SchedFS and SchedulerSvc.
SchedFS is implemented via a Cron event to adquire XtreemFS client coordinates (via Vivaldi_coordinates file), and a set of JNI code to adquire the coordinates of the OSD containing a file. In the case of SchedulerSvc the class is removed from jobMng to avoid problems with Virtual Nodes. It provides a filtering / reordering / selection of a list of nodes (normally received from a reservation). The cost of Random and RoundRobin schedulers are O(1).

SchedFS should be activated via SchedFS.conf:

```
#Properties File for the client application
#Thu Nov 05 16:05:37 CET 2009
minutes=1
VivaldiPath=/tmp/coordinates
useADS=false
xtreemFSDIR=192.168.0.1
mountCommand=/usr/bin/mount.xtreemfs
umountCommand=/usr/bin/umount.xtreemfs
```

ResMng.conf has an additional parameter specifying the maximum side_length to check:

```
side_length=1
useSchedFS=true
```

A sample JSDL can be the next one:

```
<?xml version="1.0" encoding="UTF-8"?>
<JobDefinition
        xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
        xmlns:jsdl-posix="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jsdl-sdl="http://schemas.qqf.org/jsdl/2009/11/jsdl-sdl"
        >

    <JobDescription>
        <JobIdentification>
            <JobName>ls</JobName>
            <Description>Xtreemos JSDL automatically generated</Description>
            <JobProject>XtreemOS</JobProject>
        </JobIdentification>
        <Application>
        <POSIXApplication>
            <ApplicationName>uname</ApplicationName>
            <Executable>/bin/true</Executable>
            <Output>/tmp/out.txt</Output>
        </POSIXApplication>
        </Application>
        <Resources>
            <TotalResourceCount>
            <Exact>1.0</Exact>
            </TotalResourceCount>
        </Resources>
        <SchedulingHint>
          <FileSystem type="XtreemFS">
            <Volume id="demo">
              <File>hello/readme.txt</File>
              <File>hello/readme3</File>
            </Volume>
          </FileSystem>
```

```
            </SchedulingHint>
        </JobDescription>
</JobDefinition>
```

SchedulerSvc.conf contains the selected scheduler:

```
#Properties File for the client application
#Tue Nov 03 15:31:46 CET 2009
scheduler=RoundRobin
```

9 User and Client side modifications

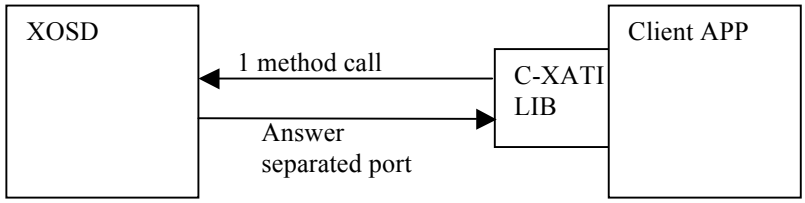
In this prototype we included important changes in the client side. The changes increase performance and reduce mobile device technical issues.

9.1 Overview

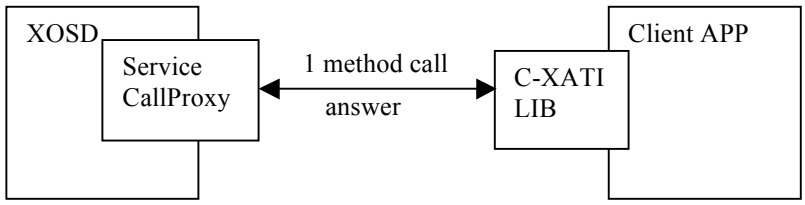
C-XATI clients used a server – client paradigm. Every petition opens up a server on the client side expecting a response. This produced several negative effects, as the impossibility of use clients over a NAT (mobile devices) and a decoupling between petition and response that was a bug source. Also issuing two xcommands at once, converted in a reissue of the command to change output port (potentially closed by a firewall).

9.2 Architecture

Old architecture was the next:



The implemented architecture is the next:



9.3 Implementation

The implementation modified XATICACommunication.c and the templates autogenerating the code. OpenSSL library has less requeriments as we are only using client side. ServiceCallProxy Service implemented in AEM provides the functionality on XOSD.

10 User guide

Code is accessible through XtreemOS SVN repository at https://scm.gforge.inria.fr/svn/xtreemos/grid/dixi/trunk

10.1 Checkpointing

Checkpoint user interface:

```
xcheckpoint –j jobid
```

Restart user interface (coordinated checkpoint image):

```
xrestart –j jobid –v version
```

Restart user interface (uncoordinated checkpoint image):

```
xrestart –j jobid
```

10.2 Job Monitoring

User tools remain the same, but we included new parameters

xps command has this signature:

```
xps [-a] [-A] [-c userCert] [-j jobid] [-m metric] [-T tag] [-i]
```

- -a selects all the jobs of the current user
- -A ANSI output (color console)
- -c userCertificateFile will use the user certificate provided. We will get *XATICAConfig.conf* as default
- -j jobID will gather information about the job with the specified jobID
- -T TAG. displays the jobs related to the jobID specified with the TAG (One direction only, 5 levels deep)
- -m metric: adds user metrics (or system) to the output. For example, SSO_ipaddr,SSO_ipaddr2
- -i . forces the generation of a ConfigFile, other options are omitted.

We can see a sample output where we can see the job, the resource where it is executing and the information of the related processes:

```
870f5901-0dde-40f6-b9ca-eabdf3c99908 @ 1224589742536 :
        jobId = 870f5901-0dde-40f6-b9ca-eabdf3c99908
        status = LocalSubmited
        submitTime = Tue Oct 21 13:48:58 CEST 2008
        resourceID = XOSDHOST/192.168.0.101:60000
        PID = 20514
        userTime = 00:00.38
        systemTime = 00:00.00
        status = R
        PID = 20526
        userTime = 00:00.34
        systemTime = 00:00.00
        status = R
```

The *xtrace* is similar to *xps*, but it includes one parameter –i <int> where we can put the time between status update on the trace.

```
xtrace [-j jobID] [-c usercertificate] [-i interval] [-o outputtrace] [-f jsdl] [-d depTag]
```

xtrace generates a trace file that can be open through *paraver* visualization tool, it can also accept a JSDL file to launch and get the trace in one shot. JobID or jdsl file is mandatory. –d option acquires information about all dependent jobs with the tag specified.

As an example calling *xtrace* with:

```
xtrace –j <JobID> -i 1 –o testTrace
```

Generates testTrace.prv, testTrace.pcf and testTrace.row *paraver* files for job with jobID and an update interval of 1 second. In this first prototype we will support status metric only.

The result in *Paraver* is shown in Figure 10:

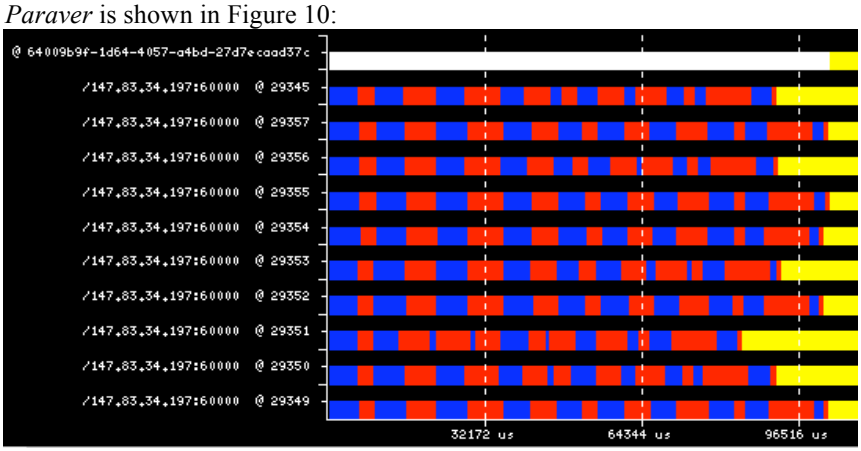


Figure 10

10.3 Monitoring Manager

```
xmon  [-addnotification monRuleString monRuleName] |
      [-cancelnotification monRuleName] |
      [-existsmonrulename monRuleName] |
      [-subscribe monRuleName] |
      [-unsubscribe subscriptionId]
```

- addnotification - adds a new monitoring rule defined by *monRuleString* and associated by *monRuleName*
- cancelnotification - cancels notification associated by *monRuleName*
- existsmonrulename - checks if monitoring rule already exists
- subscribe - subscribes to monitoring rule named *monRuleName*
- unsubscribe - unsubsribes from monitoring rule associated by *subscriptionId*

Sample *MonMng.conf* file

```
trustStore=/etc/xos/truststore/certs/
hibernateConfig=config/monmng/hibernate_config.xml
hibernateMappings=config/monmng/hibernate_mappings.xml
```

10.4 Auditing Manager

```
xaudit  [-addarchiverule monRule]
        [-cancelarchiverule archiveRuleId] |
        [-query queryString]
```

- addarchiverule - adds archive rule defined by *monRule*

- cancelarchiverule - cancels archive rule associated by *archiveRuleId*
- query - performs query with provided *queryString* on the Auditing history database

Sample *AuditingMon.conf* file

```
trustStore=/etc/xos/truststore/certs/
hibernateMappings=config/monmng/hibernate_mappings.xml
hibernateConfig=config/auditingmng/hibernate_config.xml
dbPath=auditingmng/var/historyDB
archivingRules.size=15
archivingRules.0=@cpu_system
archivingRules.1=@cpu_user
archivingRules.2=@mem_total
archivingRules.3=@mem_free
archivingRules.4=@disk_total
archivingRules.5=@disk_free
archivingRules.6=@job_started
archivingRules.7=@job_finished
archivingRules.8=@job_failed
```

10.5 Reservations

10.5.1 Command-line utility

The reservations can be used in various manners. The simplest to use are the automatic reservations which occur whenever a user submits a job containing small processes without any additional parameters. The second type of use is through the AEM API, provided in the XATI or C-XATI libraries. Finally, we a command-line utility `xreservation` which serves as a front-end to the API, helping the user to examine and manipulate time-tables. A special option of the tool –m also provides the compatibility with the MPI library.

```
xreservation [-a <reservationID>] [-m] [-1 <reservationID>] [-n <numresources>] [-f <jsdl>] [-t <durationMinutes>] [-z reservationID] [-Z] [-X] [-e] [-L] [-qf] [-qr reservationID] [-T targetNode] [-M reservationID] [-add] [-time]
```

- -a <reservationID> outputs all the resources of the reservation.
- -m output to MPIRUN special format [File MyResource].
- -1 <reservationID> outputs first resource of the list.
- -n <num resources> specifies number of resources needed.
- -f <jsdl> (use with -n parameter).
- -t <durationMinutes> duration of the reservation.
- -z <ReservationID> removes Reservation.
- -Z removes all user reservation.
- -X exclusive reservation (default Mutual).
- -C <cpu quantity> specifies the quantity of CPU to reserve [1-100], default 1.
- -e create an empty reservation.
- -L list my reservations.
- -qf list free slots.
- -qr <reservationID> print details of the reservation.
- -T <comm.Address> target a node (used by -qf, -add).
- -M <reservationID> modify a reservation (to be used with -add).
- -add add a new slot to an existing reservation (used with -M, -time, -T, optionally -X)
- -time \"dd.mm.yyyy hh:mm:ss - dd.mm.yyyy hh:mm:ss\" define the time interval
-

10.5.2 Job life-time requirements

The JSDL with proper extensions accommodates for expressing the time of the reservations. For example, the following JSDL will express a reservation request for a reservation starting on March 15 2010 at noon, which will end at the same day at 1PM:

```
<?xml version="1.0" encoding="UTF-8"?>
<JobDefinition
        xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
        xmlns:jsdl-posix="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jsdl-sdl="http://schemas.qqf.org/jsdl/2009/11/jsdl-sdl"
        xsd:schemaLocation="http://schemas.ggf.org/jsdl/2005/11/jsdl Jsdl_Normative_OGF.xsd">
        >

    <JobDescription>
        <JobIdentification>
            <JobName>ls</JobName>
            <Description>Xtreemos JSDL automatically generated</Description>
            <JobProject>XtreemOS</JobProject>
        </JobIdentification>
        <Application>
        <POSIXApplication>
            <ApplicationName>uname</ApplicationName>
            <Executable>/bin/true</Executable>
            <Output>/tmp/out.txt</Output>
        </POSIXApplication>
        </Application>
        <Resources>
            <TotalResourceCount>
            <Exact>1.0</Exact>
            </TotalResourceCount>
        </Resources>
        <LifeTime>
            <StartTime> 15.3.2010 12:00 </StartTime>
            <ExecutionTime> 3600 </ExecutionTime>
        </LifeTime>
    </JobDescription>
</JobDefinition>
```

In a more complex example, we would like to book the resources for a job that runs on Saturday and Sunday from 11pm to 11:30pm 3 weeks in a row. It will start on March 20, which is the first Saturday after March 15. The changes to the previous example are as follows:

```
        <LifeTime>
            <StartTime> 15.3.2010 12:00 </StartTime>
            <ExecutionTime> 10800 </ExecutionTime>
            <Constraints>
                <Constraint>
                    <DayOfWeek> 5 </DayOfWeek>
                    <DayOfWeek> 6 </DayOfWeek>
                </Constraint>
                <TimeInterval>
                    <Start> 23:00:00 </Start>
                    <End> 23:30:00 </End>
                </TimeInterval>
            </Constraints>
        </LifeTime>
```

10.5.3 Co-allocation

The following example shows a relevant portion of the JSDL which requests retrieval of four resource nodes that enable co-allocation of 4 resources that collectively have at least 6GB of RAM and exactly 10 CPUs. The job also requires fairly uniform resources, thus we require each individual resource to have at least 1 GB RAM and at most 4 CPUs.

Note that the total resource count reserved will always be the lowest possible according to the **TotalResourceCount** tag. If this tag is omitted a single resource will be reserved.

```
<!-- ... -->
<Resources>
        <TotalResourceCount>
                <Exact> 4 </Exact>
        </TotalResourceCount>
        <TotalPhysicalMemory>
                <LowerBoundedRange> 6442450944</LowerBoundedRange>
        </TotalPhysicalMemory>
        <TotalCPUCount>
                <Exact>10</Exact>
        </TotalCPUCount>
        <IndividualPhysicalMemory>
                <LowerBounderRange>1073741824</LowerBoundedRange>
        </IndividualPhysicalMemory>
        <IndividualCPUCount>
                <UpperBoundedRange> 4 </UpperBoundedRange>
        </IndividualCPUCount>
        <CoAllocatedNodes>
                <Exact> 4 </Exact>
        </CoAllocatedNodes>
</Resources>
<!-- ... -->
```

10.6 Virtual Nodes Support

Virtual Nodes integration with AEM is already published on the same repository and project as the rest of the system though its compilation and use is optional. In fact, the so called VNodeLayer is totally decoupled from AEM code. There is an ant compilation task to build and install the appropriate VNodeLayer jar file: install-aem-vnode.

In order to enable it for use, some configuration files need to be changed and some other newly generated.

XOSdConfig.conf needs to add the following line:

```
useRSQ=true
```

A new stage needs to be created to enable the VNodeMng (Virtual Node Manager), a new service that acts as management gateway for service replica administration in the XOSD. An example file is delivered, VNodeMng.stage, which is installed in the disabled directory by default.

A new configuration file for the Virtual Nodes library needs to be defined. An example file is delivered as well, XOSdVNodes.conf, which is installed on /etc/xos/config by default. It should be edited to specify the appropriate IP address of the node. XOSD init script needs to be changed as well to add the following option to the JVM on initialization:

```
-Dvnode.config.file=$VNODES_CONFIG #change to actual path if not using the default one.
```

Prior installation sample files are provided at the Support/config directory in the repository.

10.7 Schedulers

SchedFS should be activated via SchedFS.conf:

```
#Properties File for the client application
#Thu Nov 05 16:05:37 CET 2009
minutes=1
VivaldiPath=/tmp/coordinates
useADS=false
```

```
xtreemFSDIR=192.168.0.1
mountCommand=/usr/bin/mount.xtreemfs
umountCommand=/usr/bin/umount.xtreemfs
```

Parameters:
minutes - Minutes to read Vivaldi_coordinates file and communicate with SRDS.
useADS – For testing, ability to disable SRDS usage.
xtreemFSDIR – Address to the XtreemFS service directory
mount and **umount** – path to the umount and mount commands of XtreemFS.
A sample JSDL can be the next one:

```
<?xml version="1.0" encoding="UTF-8"?>
<JobDefinition
        xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
        xmlns:jsdl-posix="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jsdl-sdl="http://schemas.qqf.org/jsdl/2009/11/jsdl-sdl"
        >

    <JobDescription>
        <JobIdentification>
            <JobName>ls</JobName>
            <Description>Xtreemos JSDL automatically generated</Description>
            <JobProject>XtreemOS</JobProject>
        </JobIdentification>
        <Application>
        <POSIXApplication>
            <ApplicationName>uname</ApplicationName>
            <Executable>/bin/true</Executable>
            <Output>/tmp/out.txt</Output>
        </POSIXApplication>
        </Application>
        <Resources>
            <TotalResourceCount>
            <Exact>1.0</Exact>
            </TotalResourceCount>
        </Resources>
        <SchedulingHint>
          <FileSystem type="XtreemFS">
            <Volume id="demo">
              <File>hello/readme.txt</File>
              <File>hello/readme3</File>
            </Volume>
          </FileSystem>
        </SchedulingHint>
    </JobDescription>
</JobDefinition>
```

SchedulerSvc.conf contains the selected scheduler:

```
#Properties File for the client application
#Tue Nov 03 15:31:46 CET 2009
scheduler=RoundRobin
```

You can select RoundRobin | Random | LRU

10.8 JSDL Extensions implemented

We put several JSDL extensions (not official) to be used inside AEM:

- jsdl-cert to sign jobs

```
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xtreemos.org/jsdl/2010/01/jsdl-cert"
targetNamespace="http://schemas.xtreemos.org/jsdl/2010/01/jsdl-cert" elementFormDefault="qualified">

<!-- Indicates the need for a Joblevel certificate to be created by the AEM - cspann / ULM -->
      <xsd:complexType name="RequireJobCertificate_Type">
            <xsd:attribute name="JobId" type="xsd:boolean" default="true"/>
            <xsd:attribute name="Hostname" type="xsd:boolean" default="true"/>
            <xsd:attribute name="Port" type="xsd:boolean" default="true"/>
      </xsd:complexType>

      <xsd:element name="RequireJobCertificate" type="RequireJobCertificate_Type"/>
</xsd:schema>
```

- jsdl-sdl : To use scheduling hints based on used files (inside XtreemFS)

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.ggf.org/jsdl/2009/11/jsdl-sdl" xmlns:jsdl-
sdl="http://schemas.ggf.org/jsdl/2009/11/jsdl-sdl"
targetNamespace="http://schemas.ggf.org/jsdl/2009/11/jsdl-sdl" elementFormDefault="qualified">
      <!--
=============================================================================-->
      <xsd:complexType name="Volume_Type">
            <xsd:sequence>
                  <xsd:element name="File" type="xsd:string" minOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:string" use="required" />
      </xsd:complexType>
      <!--
=============================================================================-->
      <xsd:complexType name="FileSystem_Type">
            <xsd:sequence>
                  <xsd:element name="Volume" type="Volume_Type" minOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="type" type="xsd:string" use="required" />
      </xsd:complexType>

      <!--
=============================================================================-->
      <xsd:element name="SchedulingHint" type="SchedulingHint_Type" />

      <xsd:complexType name="SchedulingHint_Type">
            <xsd:sequence>
                  <xsd:element name="FileSystem" type="FileSystem_Type" minOccurs="0"/>
                  <xsd:element name="Scheduler" type="xsd:string" minOccurs="0"/>
            </xsd:sequence>
      </xsd:complexType>
      <!--
=============================================================================-->

</xsd:schema>
```

- jsdl-vvd: To export Vivaldi coordinates to SRDS service

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Experimental Schema for XtreemOS
-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.ggf.org/jsdl/2009/11/jsdl-vvd" xmlns:jsdl-
vvd="http://schemas.ggf.org/jsdl/2009/11/jsdl-vvd"
targetNamespace="http://schemas.ggf.org/jsdl/2009/11/jsdl-vvd" elementFormDefault="qualified">
    <!--
================================================================================-->
    <xsd:element name="Vivaldi" type="Vivaldi_Type" />

    <xsd:complexType name="Vivaldi_Type">
        <xsd:sequence>
            <xsd:element name="X" type="xsd:double" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="Y" type="xsd:double" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="side_length" type="xsd:double" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
    <!--
================================================================================-->

</xsd:schema>
```

- Access control and network firewall extensions.

Schemas can be found in https://scm.gforge.inria.fr/svn/xtreemos/grid/dixi/trunk/XMLExtractor/Schemas

10.9 User and client side changes

We present here the changes on several xcommands applied over the old prototype.

10.9.1 xsub

xsub xcommand provides a non-java way to submit and execute a job providing a JSDL. There is a simpler version based on a shell script that creates itself the JSDL.

```
xsub [-h] [-V] [-v] [-c cert] [-f jsdl] [-q] [-r rvid] [-j jobid]
```

- -h help output
- -v verbose and debug mode
- -c userCertificateFile will use the user certificate provided. We will get *XATICAConfig.conf* as default
- -f jsdl will submit the jsdl
- -q Do not show jobID to stdout
- -r Submits the job using the reservationId specified.
- -j jobid : Creates a process related to the jobid specified.

There is a wrapper using binfmts,

10.9.2 procps ps

We have modified procps code, ps, adding a new option to show XtreemOS jobs and processes running (related to the calling user).

```
ps --xtreemos
```

10.9.3 Bash commands (wait, kill)

Bash commands *wait* and *kill* are modified to accept jobIds as pids. They work as usual. Installation needs a correct XATICAConfig.conf file in the usual user directory (~/.xos) with user certificates.

11 Conclusions

In this deliverable we have described all new features and optimizations included in the AEM component implemented in the last months. We have proposed new XOS-specific JSDL extensions for Checkpointer, SchedFS, Certificates and Sched_Hints. In addition, we have improved the integration of bash and procps (ps, kill , wait) to support jobs. Several wrappers for C-XATI on monitoring and reservations to make the SAGA support easier. Basic commands to support interactive jobs. C-XATI and XATI clients are now pure clients to make mobile devices implementation easier.

12 References

[CPGGE] J. Mehnert-Spahn and M. Schoettner and C. Morin, *Checkpoint process groups in a grid environment, 2008, December, PDCAT08, Dunedin, New Zealand*

[XGCA] J. Mehnert-Spahn and Thomas Ropars and M. Schoettner, *XtreemOS grid checkpointing architecture and implementation, Technical Report, 2008*

[[CMCHE]. J. Mehnert-Spahn and M. Schoettner, *Checkpointing and Migration of Communication Channels in Heterogeneous Environments*, 2010, May, ICA3PP2010, Busan, South Korea

[D3213] The XtreemOS Consortium, *D3.2.13: Extended version of a service / resource discovery system, December 2009*

[D3214] The XtreemOS Consortium, *D3.2.14: Extended Version of a Virtual Node System, December, 2009*

[D3217] The XtreemOS Consortium, *D3.2.17: Distributed XtreemOS Infrastructure (DIXI), March, 2010*

[BLCR] Berkeley Lab Checkpointing/Restart https://ftg.lbl.gov/checkpoint/

[D335] The XtreemOS Consortium, D3.3.5: AEM Monitoring, November, 2008

[D336] The XtreemOS Consortium, D3.3.5: AEM Prototype, November, 2008

[D345] The XtreemOS Consortium, D3.4.5: XtreemFS and OSS Developer Guide, May, 2009

[D325] The XtreemOS Consortium, *D3.2.5: Design and specification of a virtual node system*, December, 2007