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

T3.3.5-9: Basic grid checkpointing, VO support with some accounting, dependencies, signals, basic co-allocation and job monitoring.

D.3.3.6

Due date of deliverable: November 30th,2008
Actual submission date: December 18th,2008

Start date of project: June 1st 2006
Type: Deliverable
WP number:3.3
Task number (optional):3.3.5-9

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

Version 2.0/ Last edited by Ramon Nou/ Date 08/12/18

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	08/10/10	Toni Cortes	BSC	Initial Draft
1.0	08/10/20	Ramon Nou	BSC	Monitoring Section
1.0	08/10/20	John Mehnert-Spahn	UDUS	Checkpointing Section
1.0	08/10/20	Matej Artač	XLAB	Basic VO Section
1.0	08/10/22	Uroš Jovanovič	XLAB	Resource Negotiation Section
1.0	08/10/23	Marta Garcia	BSC	Advanced Services Job Control
1.1	08/10/23	Ramon Nou	BSC	Merge, editing
1.2	08/11/10	Ramon Nou	BSC	Xtrace and xcommands update
1.3	08/11/19	Ramon Nou	BSC	XPS with dependencies update, Executive Summary
1.4	08/11/21	Matej Artač	XLAB	Updates and changes
1.4	08/11/21	John Mehnert-Spahn	UDUS	Checkpointing review
1.5	08/11/21	Ramon Nou	BSC	Merge, editing.
2.0	08/12/18	Ramon Nou	BSC	Final comments added

Reviewers
Luis Pablo Prieto (TID), Bernd Scheuermann (SAP)

Tasks related to this deliverable

Task No.	Task description	Partners involved *
3.3.5	GRID checkpointing and migration	**UDUS*,INRIA**
3.3.6	Basic VO support	**XLAB**
3.3.7	Advanced services to control jobs	**BSC**
3.3.8	Co-allocation and resource negotiation	**XLAB**
3.3.9	Job Monitoring	**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 M30. We include several features like checkpointing, advanced job control and resource related features and finally job monitoring.

Checkpointing is able to stop, restart and migrate jobs to another resource. It is important to allow features as fault tolerance and load balancing. If a resource fails the user/system could migrate and restart the job in another resource without losing the work done.

Advanced job control features, enables features that can be used by workflow managers. One of these features is the concept of Job dependencies where we can relate two or more jobs with a tag or label able to be used by other tools. We present an example of dependencies in the *xps* xcommand.

Basic VO support, the job submission and execution highly depends on the VOs, their presence within grid, and the current set of VO-level policies. The Application Execution Manager's operation should therefore ensure that the resources selected for a particular job are actually suitable for running the job. This means that, even though the AEM is not directly affected with the VOs and their policies, it should not remain VO-agnostic if we are to ensure a smooth operation of the system and reliable resource scheduling. The document will explain the AEM's role in the scheme of the VOs, and the way AEM handles requests and limitations pertaining to the VOs in Section 3.

With respect to resource negotiation and co-allocation, AEM needs an efficient way to keep the schedule of the resources available in the grid. It is essential that the schedule is being kept on the grid level to help in the process of negotiating the time slots and the resources to be used by the jobs. Section 5 illuminates the topic of receiving, keeping and changing the resource schedule.

Finally, job monitoring gives us information about jobs using XML output so different tools can use it. We present *xps* and *xtrace* as examples to show job monitoring. *xtrace* produces an execution trace of a running Job (and its associated processes) able to be analyzed using *paraver*.

In this deliverable, we also included the xcommands: a set of non-java standalone commands that allow using, and demonstrating, some of the AEM interface that we export to the user.

As in M30 we have still some interesting features to add to resource, monitoring and job control. Job monitoring enhanced by callbacks when some events fire and the concept of metric, which will allow the user to display its self-made metrics.

Table of contents

1 Introduction

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

The prototype includes features achieved by month 30 of the project: checkpointing, VO support, advanced services to control jobs, resource negotiation, co-allocation (reservations) and job monitoring. There are some features that are still under development, like monitoring callbacks, but we offer a snapshot of what it will do with tools like *xtrace*. We included the Application Firewall (**AFW**) which enables the control and enforcement of the outgoing traffic generated by the job. This prevents malicious binaries that would generate unwanted network traffic, and a fine-grained control on the level of each process that blocks or permits network communication. The technical details of the **AFW** can be found in [D3.5.8].

- Checkpointing: A former version of the XtreemOS checkpointing architecture has already been explained in deliverable D2.2.3, see [D2.2.3]. We give a conceptual overview of the architecture. In this context, we explain how heterogeneous kernel checkpointers can be used in XtreemOS to allow for job-unit migration and fault tolerance for jobs on diverse grid node types. Herein we give details about the integration of the LinuxSSI kernel checkpointer, BLCR checkpointer as well as adaption of AEM. Finally, we describe how the grid checkpointer will be used and what configuration-related prerequisites have to be met before.

- Basic VO support: AEM services need a tight cooperation of the Virtual Organization Membership (VOM) services to assert the user's identity and membership in the VOs, and to avoid scheduling the jobs to such resources that would be categorically rejected at the time of the execution. The prototype uses an integration of the AEM with the VOM services such as the VOPS to check the resources against the current VO policy. The user needs to provide the credentials obtained from the CDA services to be able to exploit the VOs, otherwise AEM rejects the user.

- Job monitoring: In the monitoring side, we provide XML output for xps, so it will be easily parsed with external tools. An example is the xtrace tool that can generate paraver (a visualization tool) traces showing job behaviour. We provide also a set of standalone (not Java) commands able to do a set of operations like job submission, job monitorization and job control. We will show them in the user guide.

- Advanced services for control jobs are also described in deliverable D3.3.3-4, see [D3.3.3-4]. We provide the tools to get job information and to control them (send events). In this deliverable, we include the interface to build dependencies between jobs that can be used by workflow managers.

- With respect to resource negotiation and co-allocation, AEM needs an efficient way to keep the schedule of the resources available in the grid. It is essential that the schedule is being kept on the grid level to help in the process of negotiating the time slots and the resources to be used by the jobs. Section 5 illuminates the topic of receiving, keeping and changing the resource schedule.
-

2 Checkpointing

2.1 Overview

The execution of grid applications can be disturbed by failures. Rollback-recovery is a well-known fault tolerance strategy. A distributed application is periodically checkpointed (a distributed consistent snapshot is saved to disk). In case of a failure, the application can be restarted from the last checkpoint with no need to restart from scratch. Checkpoint/restart can also be used to realise migration.

The grid checkpointing architecture of XtreemOS is designed to support almost all kinds of distributed grid applications, including scientific, business, and interactive applications. Grid environments connect heterogeneous resources (hardware and software) including different existing low-level checkpointing solutions tailored to a specific operating system or runtime library. The challenge is to integrate all these checkpointing solutions to build a grid-wide checkpointing and restart service that is able to consistently checkpoint/restart applications.
Currently, XtreemOS supports the following kernel checkpointers: Berkeley Labs Checkpoint Restart (BLCR) for PCs and the LinuxSSI kernel checkpointer for clusters. These existing checkpointing implementations have different interfaces and capabilities. Translation services implementing a common kernel checkpointer API allows the grid layer to access all these implementations in a uniform and transparent fashion. Different semantics must be translated (e.g. process synchronization, process group addressing) and missing functionality must be emulated (e.g. callbacks).
Furthermore, checkpoint image files need to be protected against unauthorized access because they store sensitive kernel data. In order to be safe and secure the restart needs to tightly interact with the XtreemOS security services. For more details refer to [XGCA]

2.2 Use Cases

Checkpoint/restart is a building block for different important grid functionalities in XtreemOS, e.g.:

- Job checkpointing: A job is checkpointed for fault tolerance reasons. Thus, the job can be restarted from one of the previous checkpoints at any time.
- Job suspension: A job is checkpointed before being killed. It can be restarted later on the same or on other nodes
- Job migration: A job will be checkpointed when job migration has been decided. The job is checkpointed, killed and restarted from the checkpoint on the same or other nodes..

These functionalities can be used in different cases, e.g.:

- Scheduling: Job migration can be used to realize scheduling. Low priority jobs are suspended to release resources for high priority jobs. Afterwards low priority jobs are restarted.
- Node disconnection: Before disconnecting a node from the XtreemOS grid, the jobs running on this node should be migrated or suspended to avoid computation loss.
- Fault tolerance: To avoid computation loss in the event of a failure, periodic checkpointing can be used. If a failure occurs, the job will be restarted from its last checkpoint.
- Debugging: A job checkpoint allows a developer to restart a job before a failure occurs and thus to examine the job behaviour from a certain time point in the past.

Currently, job checkpoint and restart to achieve fault tolerance is covered by the prototype.

2.3 Architecture

2.3.1 Components

The grid checkpointing architecture (see figure 1) consists of a hierarchy of logical components that tightly interact with each other via the SEDA communication mechanism, see [D2.2.3] and [XGCA]. For the sake of completeness its components are shortly explained again.

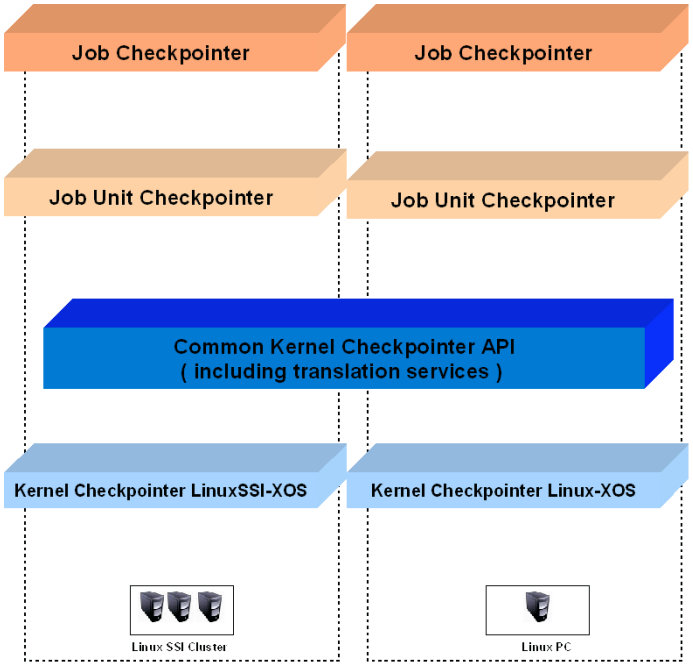

Figure 1: XtreemOS Grid Checkpointing Architecture

2.3.1.1 The Job Checkpointer

The job checkpointer is the central component of the architecture that realises job checkpoint and job restart. It takes over the role of the coordinator in coordinated checkpointing. Thus, it must conduct all involved grid nodes for job-unit synchronization.
Existing kernel checkpointers can only save the states of a single job-unit, since they are restricted to one grid node. In the opposite, the job checkpointer has a global view on all job-units that constitute a job. This knowledge is saved as grid checkpoint meta-data, which is indispensable at job restart.
Additionally, it is the job checkpointers duty to identify a kernel checkpointer that suits an applications' requirements for consistent checkpoints.
The job checkpointer also interacts with the checkpoint file garbage collector to efficiently manage disk space.

2.3.1.2 Job-unit checkpointer

The job checkpointer contacts all job-unit checkpointers, that have been selected for job-unit checkpoint/restart. Each job-unit checkpointer contacts an appropriate underlying kernel checkpointer.
For the job-unit checkpointer-service to be used unmodified on each cooperating grid node, the service provides an abstraction layer, the common kernel checkpointer API. The API allows the job-unit checkpointer to transparently access different kernel checkpointers.

2.3.1.3 Translation Library

To bridge the abstract job-unit checkpointer, running on a all grid nodes, with an individual kernel checkpointer, a translation component must be provided. This translation library implements the common kernel checkpointer API of the upper layer for an underlying kernel checkpointer. It resolves semantic differences between the AEM world and different kernel checkpointers regarding calling semantics, process groups (job-unit vs. Unix sessions, Unix process groups, process trees, LinuxSSI applications), prepares an environment for using a chosen kernel checkpointer (set capabilities, etc.) and resolves inter-job dependencies.

2.3.1.4 Kernel checkpointers

Kernel checkpointers are the low-level components in this architecture that checkpoint and restart process groups.

2.3.2 Checkpoint File Management

Two basic types of checkpoint data exist. Files containing the checkpoint image itself and files describing the snapshot: grid checkpointing meta-data. Generally, checkpoint images are bigger than the meta-data. A garbage collector has to decide which files became unnecessary and thus can be removed. For example checkpoints, created in the context of a migration, can be deleted immediately after the migration has finished.
Checkpoints created in terms of fault tolerance must be analysed properly before removing.
Dependencies between checkpoint images can occur (incremental checkpoints). In the worst case if one image has been removed, the remaining ones can become useless.
Both types of checkpoint files are placed under XtreemFS volumes to make them available at restart independent of the destination node.

2.3.3 Security

Several security threats exist in the grid checkpointing context. Checkpointing job-units include putting processes asleep, which is a security-sensitive action. Unauthorised job interruptions must not occur. Therefore, only authenticated and authorised users are allowed to issue job checkpointing.
Furthermore, it is not enough to identify a user as an XtreemOS user at restart. The user must be the owner of the checkpoint files to successfully restart - the user-identification must match the owner of the checkpoint files. The user-identification must also match the user-identification of the task (process). Otherwise, security mechanisms at kernel level deny the restart due to insufficient file access and task permissions.
Checkpoints are files that will be transferred over nodes and that are stored on disks. Data integrity must be ensured to use an unmodified image at restart time. Eventual data corruptions during image transfers must be detectable.

2.3.4 Fault Tolerance Workflow

2.3.4.1 Checkpoint Sequence

The sequence consists of the following phases:

1 Kernel checkpointer selection: When the job checkpointer receives the job checkpoint command, it figures out the belonging job-units and grid nodes. Then, an appropriate kernel checkpointer per involved grid node will be identified and the matching translation service will be called.

2 Preparation of kernel checkpointer environment: Afterwards the prepare phase is entered. The job checkpointer addresses each job-unit checkpointer in order to set up an environment where the selected kernel checkpointer can be used. Furthermore, the job-unit checkpointer must evaluate, whether the job-units processes can be referenced by a process group the kernel checkpointer uses for checkpoint/restart. If this cannot be ensured, the job-unit checkpointer must notify the job checkpointer about that - checkpointing of all job-units must be aborted to avoid inconsistent checkpoints or disturbance of other processes. At the end of this phase restart-relevant meta-data per job-unit are returned to the job checkpointer to save on disk.

3 (Job Synchronization and) Job Checkpoint: The next phases depend on the checkpoint policy. In coordinated checkpointing the job checkpointer asks all involved job-unit checkpointers to stop the job-units. Before that, the job-unit checkpointer requests the kernel checkpointer to execute user-defined callback functions. In the checkpoint phase, the job checkpointer requests the job-unit checkpointer to take a snapshot of each job-unit. The job-unit checkpointer propagates the request to the kernel checkpointer. Kernel structures representing memory pages, registers, signal structures, etc. are saved onto disk.

4 Resume job execution: After all job-units have finished with checkpointing, the job checkpointer has the job-unit checkpointers resume the job-units.

2.3.4.2 Restart sequence

For job-restart a job identification and a checkpoint version are provided. Other restart-related information (e.g. needed kernel checkpointer, resources to be reallocated, etc.) are retrieved from the per job-unit meta-data by the job checkpointer. The job checkpointer rebuilds the logical components that constitute a job. The job checkpointer requests the job-unit checkpointer to recreate the belonging job-

units. The per job-unit resources, as valid before checkpointing, will be reallocated by the job-unit checkpointer.
After having setup these grid OS inherent components, native OS entities such as processes must be recreated. Therefore, the job-unit checkpointers address the underlying kernel checkpointers. Control is thus shifted to the kernel checkpointers for rebuilding processes.
After all job-units processes have been rebuilt, the job checkpointer advices the job-unit checkpointers to have the kernel checkpointer resume the newly recreated processes. The local OS scheduler decides when they proceed with execution.
Finally, the kernel checkpointer causes a job-unit to execute their restart-callbacks (e.g. to read in files, recreate communication channels).

2.4 Implementation

2.4.1 Components

2.4.1.1 Job checkpointer (CRJobMng)

The CRJobMng is a new AEM service implementing the job checkpointer, see 2.3.1.1. The job checkpointer is an extension of the job manager (**JobMng**) to realise fault tolerance and migration for jobs.
The job checkpointer creates grid-checkpointing meta-data at checkpoint time and saves them per job-unit in the grid checkpoint xml (cpx) file. The cpx file is based on a self-defined xml schemata that is optimized for checkpoint/restart use-cases. Meta-data will be saved in a directory structure optimized for job restart (jobId/checkpoint_version/job-unit/) under XtreemFS.
At job recreation, the job checkpointer requires access to the job manager in order to add a job to the job list and the job directory.
Furthermore, resource reallocation is realised by existing code on AEM. Few modifications are still needed to select grid nodes that suit checkpoint/restart related parameters (resource reallocation is not based on jsdl file, but on the cpx file). A new interface to the resource allocation functionality is needed taking cpx file values into account.
The job checkpointer relies on the job manager for setting and releasing a job lock that had to be introduced. It is used to prevent the job from intermediate checkpoints/restarts triggered while such an action is in progress.

2.4.1.2 Job-unit checkpointer (CRExecMng)

The CRExecMng is the service implementing the job-unit checkpointer, see 2.3.1.2. It takes a job-unit checkpoint, and recreates a job-unit. The job-unit checkpointer presents an abstract API that must be implemented individually by each underlying kernel checkpointer translation library.
The JNI technology is used to bridge the Java grid service world to the C kernel checkpointer world.
The job-unit checkpointer relies on accessing the execution manager to achieve: retrieval of job process ids (kernel checkpointer is unaware of the job concept), recreation of a job-unit (members of the execution manager need to be accessed, e.g. job-unit hashtable) and reassignment of processes to a restarted job.
The job-unit checkpointer provides an interface for applications to register functions that can be called before checkpointing, after restart, in a kernel checkpointer transparent fashion.

2.4.1.3 Translation Library

The CRTransLib is the translation library, see 2.3.1.3. It realises the common kernel checkpointer API for a specific kernel checkpointer. The translation library is capable of addressing user-space, hybrid and pure kernel checkpointers. It bridges calling primitives of various kernel checkpointers. Furthermore, it sets up a specific environment for a kernel checkpointer to be usable. The translation library must ensure that the kernel checkpointer can reference the exact processes that AEM detects as a job-unit. Otherwise, checkpointing will not work, see [CPGGE] for more information.

2.4.1.4 Kernel checkpointer

The LinuxSSI and LinuxXOS kernel checkpointers had to be extended in order to be addressable by AEM to realise coordinated checkpoint/restart of a job with multiple job-units.

2.4.2 Limitations of the current prototype

Resource reallocation at restart is yet based on the jsdl file (as job submission). Thus, resource allocation is done without taking a suitable kernel checkpointer into account, as can be done when using the cpx file. The current mapping of a global user identity to a local user identity is not static. If a user gets authenticated at restart, he gets another uid/gid assigned. These results in the abortion of restart since the same user with another uid/gid does not get file access permission and cannot be identified as owner of the task stored in the checkpoint. As long as this issue is not solved, security cannot be applied to job checkpoint/restart.
Resource conflicts at restart following a reboot have to be addressed.

3 Basic VO Support

3.1 Overview

Application Execution Management is a complex set of services that spans the whole spectrum of the grid nodes:

- **client nodes**, where the user sets up the jobs and submits them, inspects the job's progress, and examines the results of the finished jobs,
- **core nodes**, which run the essential AEM services, and
- **worker nodes**, where the jobs execute in the form of the processes.

The user therefore appears on the client end, **submitting** the job request to the core services of the AEM. AEM then **selects the worker nodes** to schedule the job execution on. At the scheduled time, AEM node services **execute** the job. The client enables the user to **check the progress** of the job and **accesses the results**. The job consumes resources, being a subject to local and global policies. This means that the AEM services need a tight cooperation of the Virtual Organization Membership (VOM) services to assert the user's identity and membership in the VOs, and to avoid scheduling the jobs to those resources that would be categorically rejected at the time of the execution.
In this section, we will focus on the measures taken in AEM to ensure the basic VO support.

3.2 Use cases and details

3.2.1 User credentials

In XtreemOS, we employ the user authentication and authorization using an X.509 certificate. The user needs to obtain the valid certificate (XOS-Cert) from the CDA server. AEM then relies on the trustworthiness of the certification authority which signed the user's XOS-Cert. For performance reasons, AEM does not check with any servers for the authenticity of the user, but rather checks the validity of the XOS-Cert, and obtains the needed details on the user from the certificate's extensions.
The user's certificate is a required parameter for all job-related AEM client actions. The internal API also assumes that with AEM service calls the service obtains the public certificate part of the XOS-Cert. AEM services uses this certificate to check its validity, and extracts the certificate's **Common Name** which contains the unique global ID of the user. AEM uses this ID to map the job and the corresponding actions to the user. The further use of the certificate is to pass it on to the other VOM services, so that they can take advantage of the certificate extensions and their values.
Checking the certificate's validity in AEM involves the following:

- checking, whether the current date and time is at or after the certificate's *not before* field value,
- checking, whether the current date and time is before or at the certificate's *not after* field value,
- asserting that the certificate was signed by one of the trusted certification authorities.

3.2.2 Resource Selection

AEM's Resource Management relies on other services to obtain the list of the resources suitable for the user's job. The policies related to different VOs permit or deny the access to specific nodes; therefore

AEM should first ensure that the resource scheduling assigns realistic resources to a job. The user selects which VO to use for the job submission, but the submission has a chance to succeed only if the user has the proper credentials for the VO. AEM consults **VOPS** and **ADS** in this process.
ADS. This service provides the resource discovery. AEM submits a query in the form of a JSDL document that contains the resource requirements for the job. Optionally, in a more advanced scenario, the query also includes the policy filter obtained from VOPS to provide a better early resource selection.
VOPS. This service stores the policies for the VOs. AEM makes a service call to VOPS, providing the outcome of the resource discovery and the user's credentials. VOPS performs additional filtering based on the VO's and user's details, and returns a signed decision containing the certified resources that can run the user's job.
Therefore, AEM does not enforce anything regarding the VOs explicitly, but rather uses other, specialised services as decision points.

3.2.3 Job Execution and Result Collection

AEM's Execution Manager runs the processes belonging to a job. The processes need to run on behalf of the user.
XtreemOS PAM modules provide a session on a local node to run the job's processes. AEM passes to the modules the user's certificate, letting the job's processes be run as if the user logged into the node using the global user ID. AEM also sets the policies and limitations required by the VO-level policies and the schedule constraints, and the PAM modules enforce them, combined with the local policies.
XtreemFS service provides the distributed storage and networked access to the working files. AEM does not interact with XtreemFS directly, but the jobs executed by AEM can load the input data from or store the output into a VO volume of XtreemFS.

3.3 Architecture

In the prototype, AEM represents the consumer side of the Virtual Organization Management (VOM) services. Figure 2 shows a simplified diagram of the workflow for the job submission, illustrating the overall architecture of the VO support for the job submission. The illustration involves the following steps:

1. **Certificate request.** The user obtains her credentials from the CDA server or through the VOLife's web interface (not shown on the figure). This involves the user first authenticating with the certificate issuing service either using a shell log in or the web account. The user's interaction here is purely with the VOM services, but they are a prerequisite for working with the AEM.
2. **User's certificate** gets installed on the user's client computer either automatically (CDA client) or manually (VOLife web interface). The CDA client installs the certificate in the user's home directory (profile) where the AEM client programs expect it to be.
3. **Job request**, formed by the user, is an input to the AEM client program (e.g., `xsub`). The client program also reads the user's certificate, and sends it along with the job request, to the AEM core services.
4. AEM obtains the **resource candidates** using resource discovery provided by the ADS service. The resource candidates conform to the requirements expressed in the job description.
5. The resource candidates then need to be checked for the VO policies. AEM sends the candidate list to the VOPS and obtains the certified list using the **VO policy filter**. The result is a list of candidates that are both capable of running the user's job, and the VO policies permit the user to actually run the job. This is an important step in using the VOs because it makes for a high likelihood that the job scheduled on the node will actually be able to run on the node.
6. Once the **jobs are assigned to the nodes**, the AEM executes them on the worker nodes. To do this, it passes the user's credentials to the PAM modules installed on the worker nodes.

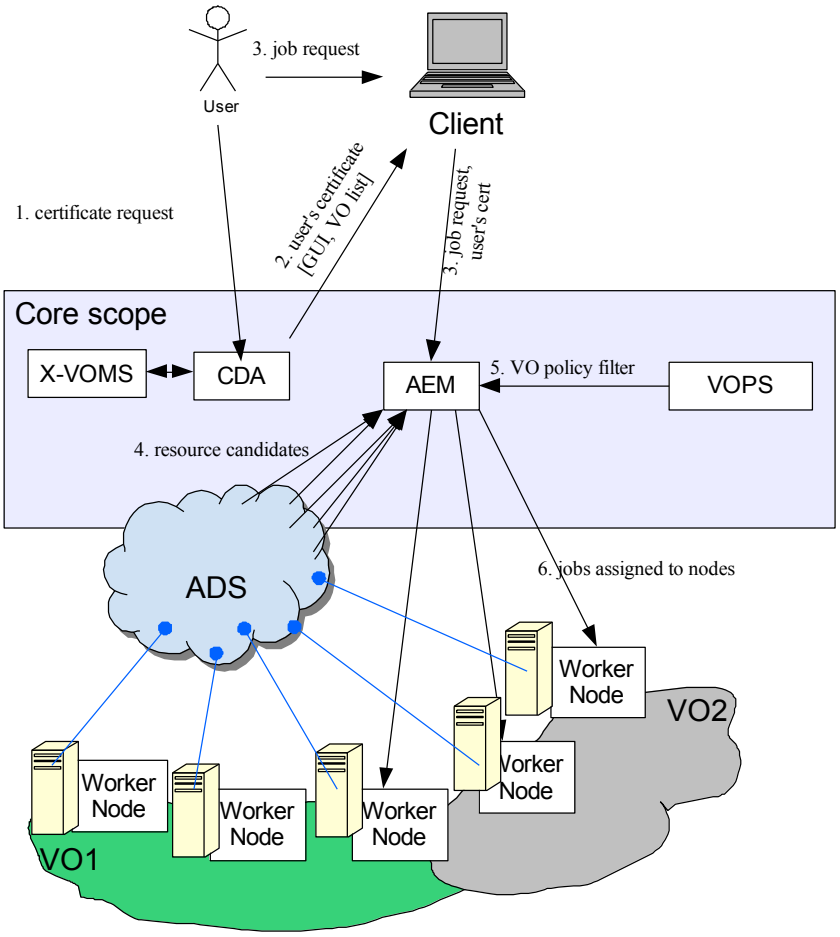

Figure 2: VO support in the AEM.

3.4 Implementation

The considerations that went into the implementation of the prototype relating to the VO support involve the following:

- The AEM client programs use configuration files to learn the **path to the user's certificate**. The default configuration points to `$HOME/.xos/truststore/certs/user.crt`, and the private key in`/$HOME/.xos/truststore/private/user.key`.
- The client programs also provide a way to pass the path to a user key alternate to the one in the current configuration.
- AEM's Resource Manager uses the API of the VOM's VOPS to have the policy decisions done.
- PAM modules' API expects the user's certificate content, and the AEM's node-level services pass them through the Java's JNI.

4 Advanced Services to control jobs

4.1 Overview

XtreemOS as in M30 provides the users with a deeper control of job's execution. The two main contributions that appear are the following:

Handling of signals for the jobs running in the Grid. AEM offers two mechanisms of signals for the jobs running in the Grid, on one hand we export the existing automatic signals in Linux to the grid job.

On the other we provide a set of grid signals that can be sent to the jobs. On both cases the AEM system will provide that all processes of a job receive the sent signal.

Relationship of dependencies between jobs. In order to offer the needed tools for the implementation of workflows by third parties AEM offers an interface to manage the relationships of jobs.

4.2 Use Cases

The use cases targeted by this task can be divided in two groups: to improve the cohesion of the processes of a job running in the grid, and to enable the relationship between different jobs running in the system.

The basic use cases are:

- Send the XOS_SIG_CHLD to the processes of a Job when a process finishes.
- Add a relationship of dependence between jobs associated to a tag.
- Get all the jobs that depend from a job given for a specific tag.

Examples of more complex use cases:

- Create a workflow manager for jobs.
- Send a signal to all the jobs that are dependent from one.

4.3 Architecture and implementation

4.3.1 Signals

The AEM implements a new signal XOS_SIG_CHLD. This signal will be sent to all the processes of a job still alive when one of the processes of the job ends. This feature will need to extend the kernel to add this new signal. For this prototype we change the signal to SIGRTMIN.

When a process of a job running in XtreemOS finishes (or an SIG_CHILD is received), a XOS_SIG_CHLD signal will be sent to all the processes of the job in a transparent manner independently of where they are running. Any XtreemOS-aware application can catch this signal to manage its processes.

For the rest of the signals the following interface is provided to send any signal to a job.

void sendEvent(string jobId, int signal, int operation, ArrayList<int> list, X509Certificate userCtx)

4.3.2 Dependencies

The interface to manage dependencies between jobs will provide the programmers with the tools to implement workflows. The dependencies will be organized by *tags* that will ease the use of dependencies by different layers. The tags will be represented as a string, and each relationship of dependence will be associated with a tag.

Dependencies are accessible in both directions. This information will be stored with the information of the job, in the Job Manager.

The AEM offers an interface that allows adding dependence between two jobs or deleting an existing dependence. The interface provided to manage dependencies is as follows:

addDependence(string JobId1, string JobId2, string TAG, X509Certificate)

deleteDependence(string JobId1, string JobId2, string TAG, X509Certificate)

The interface to obtain the dependences needs a jobId and the TAG associated to the dependence. We can specify the levels and the direction in which to search the relationship.

getListOfDependences(JobId1, TAG, levels, direction, X509Certificate)

5 Resource Negotiation and co-allocation

5.1 Overview

Service for reservations of resources: user is allowed to allocate reservations for single resource or a group of resources. Reservations are made for a time interval and if needed the amount of the resource is also provided (for example, how much RAM is needed). Service provides transactional reservations and migrations, allowing for safe group reservations. Also, the service architecture is based on static types – each attribute of a reservation must be represented by an object, making handling of errors in reservation requests easier. Requests for allocation are queued, allowing concurrent writes, however, the back-end is single threaded.

5.2 Use cases

- basic operations: adding, removing (outline of the image: actor, then a box with operations, stating that actor selects an operation, another actor – execution unit, performs the operation on the time-table, notifies the user about results)
- group resources (user makes a reservation, several time-tables are checked by the execution unit, result is returned)
- transactions (execution unit makes an operation over time-table, migrations manager records the changes)

5.3 Architecture

Figure 3 shows the architecture of the reservations service: having three parts, front-end, execution unit

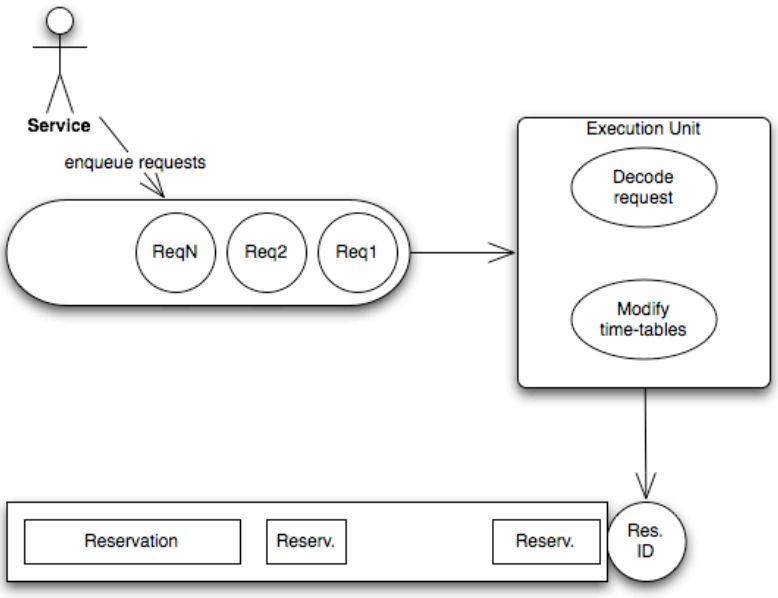

Figure 3: Overview of the reservations architecture.

and time-table. Front-end contains a queue of requests, which are then pulled by execution unit one by one, processed and then required operations are performed on the time-table. The operations are recorded to the migrations mechanism. Interaction with the service is shown on **¡Error! No se encuentra el origen de la referencia.**.

5.4 Implementation

The components are divided into the following groups:

- attributes
- attribute handlers
- basic elements
- manipulators of the basic elements
- groups
- migrations
- front-end

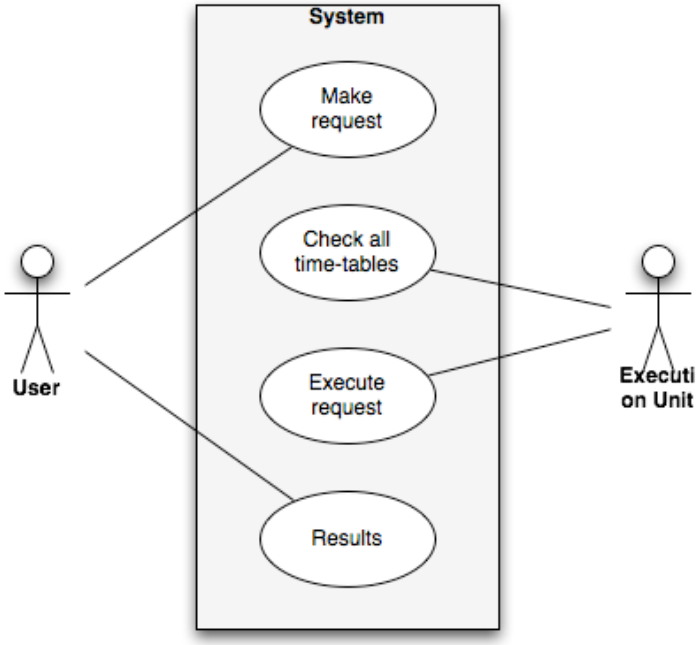

Figure 4: Abstract use case for the service.

Attributes represent the object that is mapped from the reservation document. This allows for static typing and provides greater control over the execution and mapping than using simple maps with strings. However, such rigid environment also demands an active involvement when new attributes are introduced.

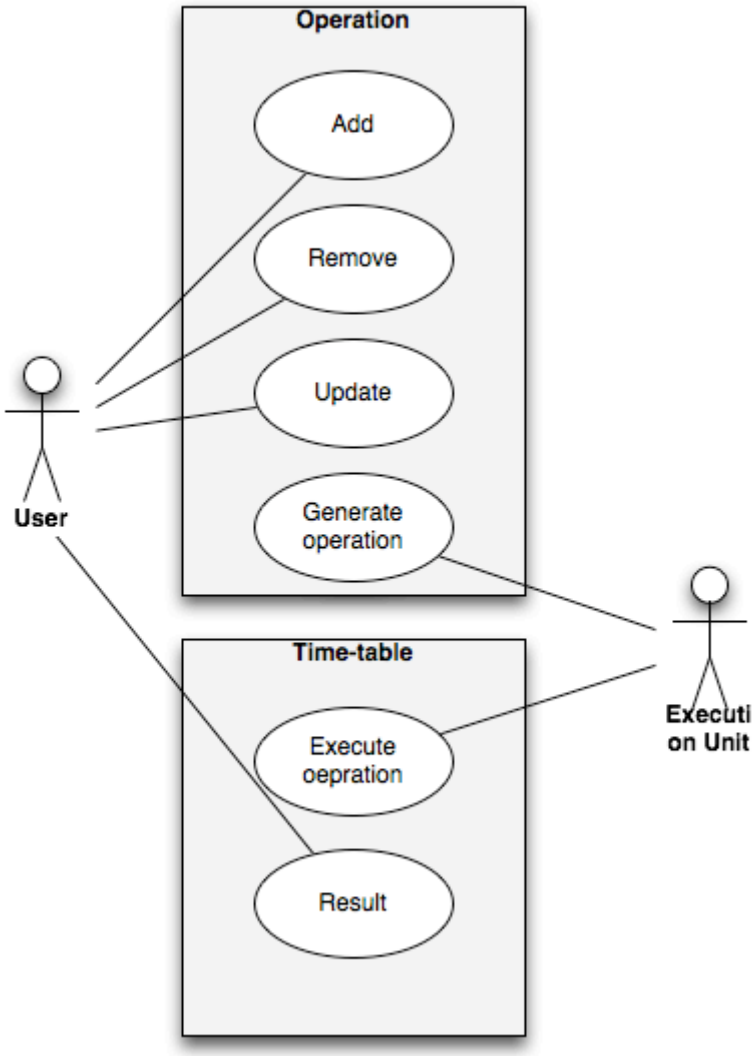

Figure 5: Use case for operations

Each attribute has a dedicated handler, which knows what to do with an attribute. These handlers provide operations that are allowed over different attributes as shown on **¡Error! No se encuentra el origen de la referencia.**.
Basic elements are the basic building blocks of the service. Like an entry in the time-table, time-table, etc. Each of these elements has a dedicated manipulator which represents an operation over the element (for example adding, removing, and searching in the time-table). Collective reservations are handled within the groups' package.

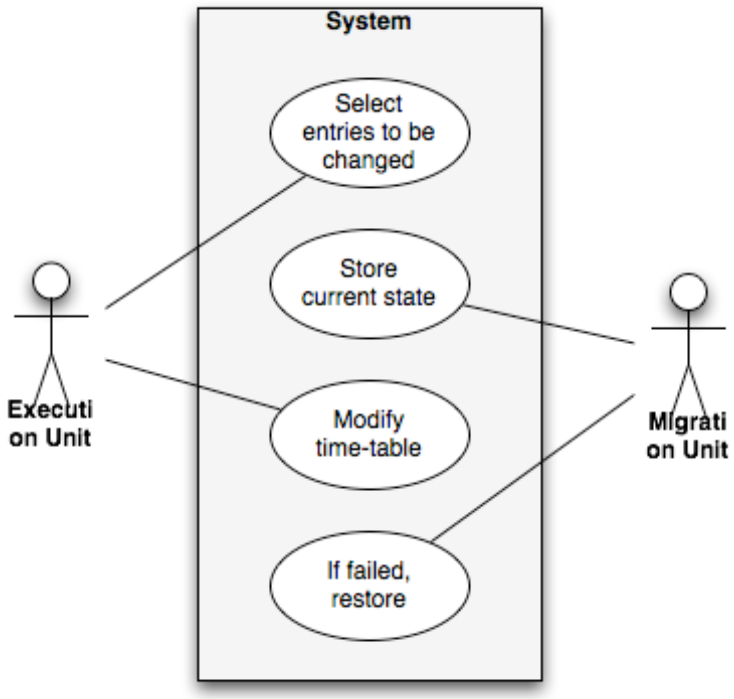

Figure 6: Use case for migrations.

Migrations use simple object mapping mechanisms that provide atomic operations over the manipulators as shown in **¡Error! No se encuentra el origen de la referencia.¡Error! No se encuentra el origen de la referencia.**.

6 Job Monitoring

6.1 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. In XtreemOS we will provide them into the job monitoring interface.

The job monitoring capability on XtreemOS as M30 will provide information about the jobs, and processes related to these jobs, of the caller user.

Although we will provide basic features now, more advanced ones that will be described on D3.3.5 will be developed in the next months. These features are monitoring callbacks and the ability to add any monitoring value by the user using the concept of Metric.

6.2 Use Cases

Job Monitoring will have the next aim:

- To automatically provide typical information associated to jobs such as execution time
- To provide mechanism to limit the type and the granularity of information collected
- To provide mechanism to easily add new information to the generated by the system

- To provide mechanism to be notified when certain monitoring events fire (callbacks)

We can describe several use cases using the monitorization that we will provide on this prototype:

- Get information about the execution of a job (typical Linux ps command)
- Generate trace information about the behaviour of a job (tracing / profiling tool).

6.3 Architecture

The architecture of job monitorization is provided by **JobMng** and **ExecMng** inside AEM. It uses **XATI** and its **C-XATI** (C-interface) to export methods to the users. The architecture for the prototype doesn't include any work on callbacks export to the user and buffering mechanism. Getting buffering working over some metrics will need to create several circular queues that will be filled up with the values (and their timestamp). We will need to provide events to notify when a buffer is full.

In Figure 7 we can see a diagram showing relations between components to enable monitoring.

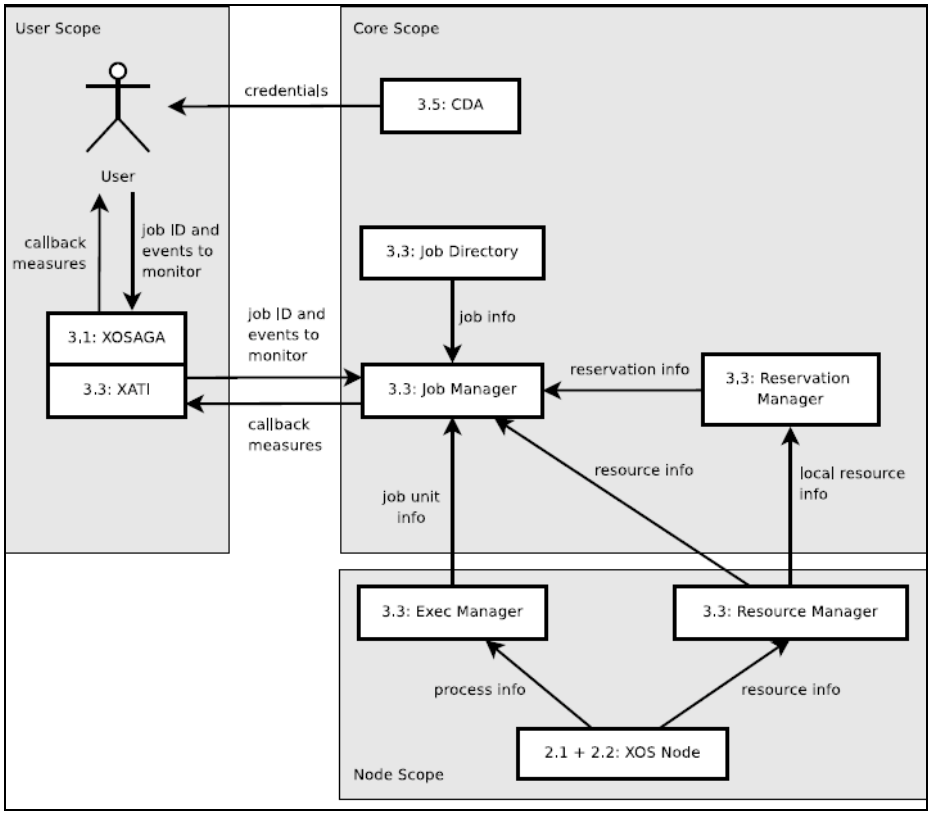

Figure 7 - Monitoring Diagram (D3.1.7)

6.3.1 JobMng

JobMng provides us with job related information, we get into contact with the job manager that "controls" the job and ask for information of the status of the job (like submit time, or status).
When we enable buffering of metrics in the future, **JobMng** will store those per job in case of metrics related to the job.

6.3.2 ExecMng

ExecMng provides us with information about the execution of a Job; in concrete we can get processes' job information. So the different **ExecMng** are contacted to get all the information of the running processes. As in the **JobMng** buffering for metrics related to the **ExecMng** will be stored here per job-unit.

6.4 Implementation

We have the next set of operations ready to use:

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

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

int getJobInfo(string jobId, int typeOfInfo, int infoLevel, list<String> metrics,
X509Certificate userCert, @OUT XMLString jobInfoList);

These operations are related to **JobMng**. *getJobInfo* can ask for extra information (about running processes) to **ExecMng** (using getProcsInfo). Now, the output information is fixed but it will be customizable with the input parameters (*typeOfInfo*, *infoLevel* and the list of metrics) to reduce monitoring overload provided when contacting other components. Currently we present the results as an XML with the next structure:

Finally xcommands, *xps* and *xtrace* are built using the previous interfaces (with **XATI-C**). The advanced version of *xtrace* will be built using callbacks and metrics functionality, now we are providing a poll version. The xcommands should be taken as examples on how to use the **XATI-C** interface.

7 User guide

Code is accessible through XtreemOS SVN repository at https://scm.gforge.inria.fr/svn/xtreemos/WP3.3/

7.1 Checkpointing

Two new commands can be issued on the console to address fault tolerance functionalities. The job checkpoint command is:
xcheckpoint jobID.
The CRJobMng implements the command via the interface:
public void checkpointJob(String jobID, X509Certificate userCert)

The job restart command is:
xrestart jobID checkpointVersion
The CRJobMng implements the command via the interface:
public void restartJob(String jobID, String checkpointVersion, X509Certificate userCert)

A job checkpoint can be taken by issuing 'xcheckpoint jobID' at the xconsole.
During checkpointing two types of data will be set up: grid checkpointing meta-data and kernel checkpointer generated image files. Currently, the grid checkpointing meta-data will be saved under the XtreemOS checkpoint path '/etc/xos/job_checkpoint_image_directory/ which is created automatically if non-existent. After issuing a checkpoint command, the directories jobID/checkpointVersion/JobUnit_1/ will be created under the XtreemOS checkpoint path, containing an xml file with the meta-data.
In LinuxSSI the kernel checkpointer stores image files under /var/chkpt/. In case of a successful checkpoint, the following files will be created: global_VERSION.bin, node_NODEID_VERSION.bin, task_APPID_VERSION.bin, task_mm_APPID_VERSION.bin.
For LinuxXOS the directory /etc/xos/blcr/ must exist with read/write access permission for any user. Furthermore, the BLCR kernel modules (blcr, blcr_vmadump, blcr_imports) must be loaded by issuing 'make insmod' applied to the BLCR installation directory. In order to checkpoint a job using BLCR, the application has to be linked statically with BLCR user library (gcc -o test test.c -lcr). This overhead will be replaced in a later AEM version. In case of a successful checkpoint, the following files will be created: PID.VERSION.blcr, PID.VERSION.xos, proc.txt, queue, thread_for_process_PID.txt.
To restart a job one has to issue 'xrestart jobId version' at the Xconsole. For a restart to be successful the grid checkpointing meta-data and the kernel checkpointer image files must locally be available under the before-mentioned locations.

7.2 Basic VO Support

To configure Job Manager to properly check whether the certificate is trusted, the node that runs Job Manager locally needs a folder containing the public certificates of the certification authorities that we trust. Currently, this includes the public certificates of the CDA services of each organisation that our organisation trusts, as well as their root certification authority public certificates. By default, these certificates are placed into /etc/xos/truststore/certs/. The Job Manager reads the path from its configuration file */etc/xos/config/JobMng.conf*, defined as the value of the **trustStore** option.
The clients using **XATI** needs to take care of loading the user's certificate and the private key, prompting the user to enter the password to obtain the private key contents. The AEM client programs provided in the XtreemOS Linux distribution, such as **xconsole** and command-line *xsub*, *xkill* etc., however, already take care of loading the necessary files. By default, the command-line CDA client places the certificate files in the folders where the AEM client expects them:

- ~/.xos/truststore/certs/cda.pem is the user's public certificate,
- ~/.xos/truststore/private/cda.pem is the private key corresponding to the user's certificate.

If the user prefers a different location for the certificates, then the **XATI** configuration (found by default in ~/.xos/*XATIConfig.conf*) and **C-XATI** configuration (found by default in ~/.xos/*XATICAConfig.conf*) need to be modified:

- **userCertificate** option contains the path to the user's public certificate file, and
- **userKeyFile** option contains the path to the user's private key.

7.3 Advanced Services to control jobs

We will explain the different methods and interfaces provided via **XATI** and **C-XATI** interfaces.

The new interfaces provided to manage signals and control jobs is this one:
void sendEvent(String jobId, Integer signal, Integer operation, ProcessList list, X509Certificate userCtx)

This method allows choosing the processes of the job that will receive the signal. The operation parameter indicates if the signal should be sent to all the processes of a job, just to the master process of the job, only to the processes that appear in the list or all the process of the job except the ones that appear in the list.

We have also an xcommand, xps able to send events directly without java console.

The interface provided to manage dependences is as follows:

addDependence(JobId1, JobId2, TAG, X509Certificate)
Will add the dependence from JobId1 to JobId2 associated to the TAG.

deleteDependence(JobId1, JobId2, TAG, X509Certificate)
Will delete the dependence from JobId1 to JobId2 associated to the TAG.

getListOfDependences(JobId1, TAG, levels, direction, X509Certificate)
Will return the list of JobIds that have a dependence relationship with the job with the JobId given and the TAG specified. The direction parameter indicates if we want the dependences from that job or to that job. The level parameter will indicate how many levels of dependences we want. If the level is 0 we will only get the jobIds directly related to the one we are indicating.

7.4 Job Monitoring

We provide the next methods for the job monitoring, they are shown through **XATI** and **C-XATI** interfaces being able to use them through JAVA code and C code.

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

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

int getJobInfo(string jobId, int typeOfInfo, int infoLevel, list<String> metrics,
X509Certificate userCert, @OUT XMLString jobInfoList);

More information about these interfaces can be found on D3.3.5 [D3.3.5]. The output expected on *getJobInfo*/*getJobsInfo* is the XML shown in **¡Error! No se encuentra el origen de la referencia.**.

Table 1 - XML output from getJobInfo

```
<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>
</metric>
</jobInfo>
</jobInfoList>
```

There are a two xcommands related to Job Monitoring as they use the *XATICA* interface, they need a *XATICAConfig.conf* file similar to the one in **¡Error! No se encuentra el origen de la referencia.**.

Table 2 - Sample *XATICAConfig.conf* file

```
xosdaddress.host=XOSDHOST
xosdaddress.port=55000
address.host=LOCALIP
address.port=10001
certificateLocation=/etc/xos/truststore/certs/xati_dummy.pem
privateKeyLocation=/etc/xos/truststore/private/xati_dummy.pem
trustStoreSSL=/etc/xos/truststore/certs/dixi_ssl
useSSL=false
cdaaddress.host=CDAHOST
cdaaddress.port=60000
userCertificateFile=/home/rnou/.xos/truststore/userCert.pem
```

xps command has this signature:

```
xps [-a] [-A] [-c userCert] [-j jobid]
```

-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)

We can see a sample output in **¡Error! No se encuentra el origen de la referencia.** where we can see the job, the resource where it is executing and the information of the related processes. The output is generated from the previous XML file so the user can adapt it to any use he wants.

Table 3 - xps output

```
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]

xtrace generates a trace file that can be open through *paraver* visualization tool [REF], it can also accept a JSDL file to launch and get the trace in one shot. JobID or jdsl file is mandatory.

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 on Figure 8:

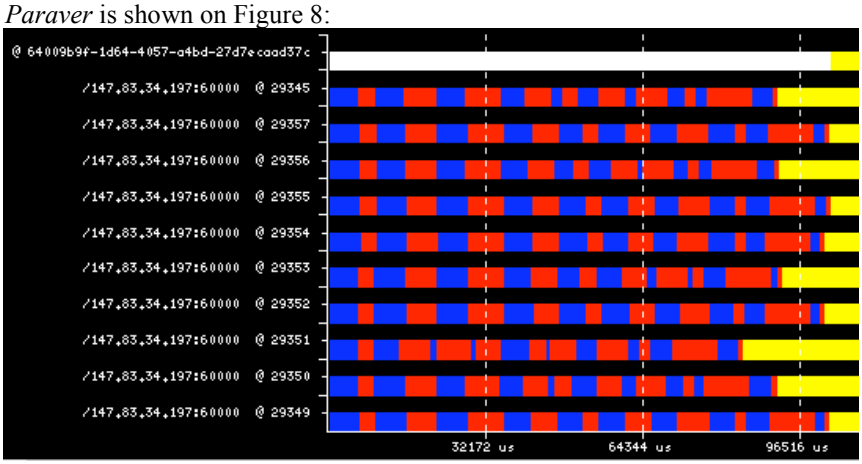

Figure 8 - xtrace output on Paraver

7.5 Other xcommands

With the **C-XATI** interface we provide a set of user commands (like *xps* or *xtrace*, shown before) to execute several operation without the need of a JVM. Those commands (from now xcommands) can be used on several scripting and testing jobs. For example, one can execute an *xps* command and process its output to see if a recently submitted job is running. **¡Error! No se encuentra el origen de la referencia.** contains a list of those xcommands currently available in the prototype.

Table 4 – XCommands available on the prototype

XCommand	Feature / Equivalence
xsub	Job Submission / xsub
xsub.sh	Job Submission without providing JSDL
xps	Job monitorization (job and process state) /xps
xkill	Send job events / sendEvent / jobControl

xwait	Waits / receives job exit status / jobWait
xtrace	Generates a Paraver trace

It is planned to have a wider subset to be able to have an MPIRun being able to execute on XtreemOS. Those xcommands work with *XATICAConf.conf* configuration.

7.5.1 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]
```

-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

xsub.sh calls *xsub* but the user needs to specify only the executable file, the parameters if any and –in –out –err files to redirect the different channels.

```
xsub.sh <executable> <parameters> -in <input> -out <output> -err <error>
```

7.5.2 xkill

xkill provides a way to send events to jobs providing the jobID and the number of the event (or signal).

```
xkill [-h] [-v] [-j jobID] [-c cert] [-e event]
```

-h help output
-v verbose and debug mode
-c userCertificateFile will use the user certificate provided. We will get *XATICAConfig.conf* as default
-j jobID
-e Event. Event to be sent to the job with the specified jobID

7.5.3 xwait

xwait will block until the jobID is done/finished.

```
xwait [-h] [-v] [-c cert] [-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
-j jobID. JobID of the job to wait.

8 Dependencies / Roadmap

In this section we will show the roadmap for the components of the prototype and introduce the dependencies with other components.

8.1 Checkpointing

Recently, the LinuxSSI checkpointer has been extended to checkpoint / restart a subset of multithreaded and multiprocess applications. This translation library must be refined to work properly with the new LinuxSSI kernel checkpointer functionality.
Currently, checkpointing using BLCR works only for single-process applications. The BLCR translation library will be made working for multiprocess application to checkpointable by the end of November 2008.

Issuing a job checkpoint works only on grid nodes where the job manager resides. Check-pointing of a multi-job-unit job must be enabled. Both functionalities will be available by the end of March 2009.

Checkpoint related files must be made accessible to all grid nodes, since it is unknown which grid node will serve for job restart in case of a grid node failure. This can be achieved by using XtreemFS. Each VO sets up an XtreemFS volume to host migration-related checkpoints that can be removed immediately after the migration has finished. User-initiated checkpoints can be saved in the XtreemFS HOME volume of the user.

For realising load balancing, job migration must be enabled. In XtreemOS migration will be realised by checkpointing a job on a source grid node and restarting it on a destination grid node. Integration of XtreemFS as well as job migration may be realised by the end of March 2009.

A static mapping of the global and the local user and group identification is needed. The changing uid/gid between multiple PAM authentications leads to denials of the checkpoint/restart service by the kernel checkpointers. Resulting id collisions must be handled. However, this functionality needs to be provided by security/authentication related work packages.

8.2 Basic VO support

We see the further development of the AEM support in the following areas.

- The first release of the XtreemOS supports only one VO. As a result, AEM expects that the user wants to submit the job as a part of the first VO listed in the user's certificate. In the next release, the users will be able to take part in any number of VOs, and the AEM client programs and its API will need to accommodate the selection of the VO to be used.
- Currently, the services or the sets of services individually take care of reading and handling user's credentials. Further development should also focus on taking advantage of common tools already developed by the project partners. For instance, the Credential Obtention Framework [D3.6.3], developed for the mobile devices in the WP3.6, could be extended for the desktop devices as well. While this is the task for the WP3.5, the AEM should take advantage of it.
- AEM should support a more advanced management of the jobs. Currently, the job belongs to the creator (i.e., the user submitting the job), and the owner is the sole user who can monitor, stop or checkpoint the job. AEM would therefore need to support a kind of an access control to a job, letting the owner set permission to the other users within a VO to manipulate the job.
- ADS/RSS currently provide the resource discovery only depending on the job requirements. This increases the likelihood that many or all of the retrieved candidates would be denied by the VO policies. In the next release, ADS is to support the selection refinement based on the VO policy filter, reducing the retrieval of unsuitable resource candidates even further.

8.3 Advanced services to control jobs

Scheduling hints and resource related behaviour will be completed on March 2009. With this prototype we have fulfilled R58, R68 from [D4.2.5]

8.4 Resource Negotiation and Co-allocation

Front-end to the other services must be built. Front-end is composed from an enqueuing mechanism and reservations objects, which are pushed to the evaluation and execution part of the reservations service.

8.5 Job Monitoring

The Job Monitoring needs a set of features and capabilities (will be described on D3.3.5). Next steps includes building metrics operations (creation, description, adding user metrics) and finally building callbacks infrastructure to export callbacks to the user. It is planned to add buffering on several events, in order to reduce calling overhead on some cases (like tracing tools). These features need some work that will be finished in March 2009. The monitoring work is related to accounting so we will need interaction with VOM. We fulfilled partially R62, R63, R64, R65, and R57 from [D4.2.5]

9 Conclusions

A lot of improvements and new features have been included on this deliverable and most of the technical issues (as checkpointing) have been solved.

In the next months we need to work on AEM to improve and complete monitorization features, with key points as user callbacks and metric implementation, complete checkpointing for multiprocess and multi-job-unit job, job migration to provide load balancing. Regarding VO further improvements need to be done but the main infrastructure is done. Finally reservations services need to be offered to other services via a front-end that needs to be built.

Within the next months we will work also in the scheduling hints and related behaviour that will create a collaboration between resources and jobs, and to provide VOM with accounting tools.

10 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*

[D2.2.3] J. Mehnert-Spahn and M. Schoettner, *Design and implementation of basic checkpoint/restart mechanism in LinuxSSI D2.2.3, 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*

[D3.3.5] Toni Cortes, Julita Corbalan, *AEM Monitoring. D3.3.5, 2008*

[D3.5.8] Jaka Močnik, *Specification of application firewall, D3.5.8, 2008*

[D3.6.3] Luis Pablo Prieto, Jesús Malo, *XtreemOS-G for MDs/PDA, D3.6.3, 2008*

[D4.2.5] SAP, *Evaluation Report and Revision of Application Requirements, D4.2.5, 2008*