Project no. IST-033576

XtreemOS

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

Design of the architecture for application execution management in XtreemOS
D3.3.2

Due date of deliverable:
Actual submission date: June 5th, 2007

Start date of project: June 1st 2006

Type: Deliverable
WP number:3.3
Task number (optional):3.3.2

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

Version 1.0/ Last edited by Toni Cortes/ June 5th, 2007

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
1.0	07/04/10	Julita Corbalán	BSC	First draft.
2.0	07/04/27	Gregor Pipan	XLAB	Resource description/jResMarching
3.0	07/05/03	Toni Cortés	BSC	Introduction/Exec. Summary
4.0	07/05/31	Julita Corbalan	BSC	Reviews included
5.0	07/06/05	Toni Cortés	BSC	Final checking

Executive summary

In D3.3.1 the list of services for the Application Execution Management (AEM) system in XtreemOS were presented. In that list many per-job and per-resource services were presented. It is now the time to see how all these services can be implemented without a significant overhead as far as resource consumption is concerned. In this deliverable, we present the architecture of the AEM services as they will appear in the XtreemOS operating system.

The AEM software will be divided into client and system components.

The client side of the AEM is in charge of providing the commands that the user will use to execute applications as well as the call to XtreemOS to execute and manage jobs, and to manage resources. This component is divided in three layers: client API, Basic Functionality and Communication layer.

The system side takes care of implementing all functionality offered to the client side. The XOSD daemon will be in charge of this task. This daemon is divided into three threads. The XJobMng is a thread of the XOSD that receives and processes all the requests for the jController, jMonitor, jScheduler, jCheckpointer, jMigrator, and jEvent. The XResMng thread handles all the requests concerning resources. It exports the rAllocator, the Negotiator, the jResMatching and the rMonitor interfaces. The XExecMng thread implements the functionality of those services involved in the execution of the job (all or part of the functionality): jExecMng, jEvent, jMonitor, and jCheckpointer. The XExecMng manages the jobs (or units of jobs) running in the node.

Also in the system side, but with global view (XOSD has a local resource/job view) we have the Job Directory that will keep track of all the jobs in a VO. The Job Directory will be implemented using a Directory service offered by WP3.2 and the architecture will be defined in that work pacakage.

Finally, we present how the AEM components interact with other components such as VOM or Node Management.

Table of contents

Introduction

This document describes the architecture for the Application Execution Management (AEM) component in XtreemOS.
In D3.3.1 the list of services for the Application Execution Management (AEM) system in XtreemOS were presented. In that list many per-job and per-resource services were presented. In this deliverable, we present the architecture of the AEM services as they will appear in the XtreemOS operating system. We have decided to group several of these services in just one daemon (one process) per node and VO. This system daemon, the XOSD, will be composed of three threads (XJobMng, XExecMng and XResMng) that will implement all the services presented in D3.3.1 except the JobDirectory. Each XOSD will manage part of the jobs submitted and executing in the VO and functionality concerning the resource itself. In this way, rather than having many processes running per node to offer the AEM services with the amount of resources this implies and with the security problems associated, we centralize in just one process per node and VO.
This document is organized as follows: First, we will refresh the list of services described in D3.3.1. Then, in Section 3, we will present the global architecture that will be detailed in Sections 4 (client side) and 0 (system side). In Section 6, we will present the interaction of the AEM components with other components such as VOM or Node Management. Finally we will present a set of pending decision that will be taken later in the project (Section 7).

List of services

- Job Management Services (JMSs)
    - jController holds the job information and its three main goals are first, to ensure that the scheduling agreement between the job and the resources is accomplished, second, to validate the job is executing as expected, and third, to act as a gateway for the job. The jController holds most of the information associated to the job and it is the only service that has a global vision of it. It is the service that provides self-management to XtreemOS jobs.
    - jScheduler schedules one job. The jScheduler receives a pre-selection of resources from the jResMatching based on job resource requirements and in a second step it performs a negotiation with pre-selected resources in order to decide the final allocation. The jScheduler service is stateless and from one job scheduling to a next one no status is stored.
    - jMonitor collects information from all the processes of the job and adds them in a job basis. One of the goals of the jMonitor is to provide a monitoring service as powerful and flexible as possible, allowing advanced versions of XtreemOS to add new metrics without changing either the API or the system architecture.
    - jExecMng is a distributed service that implements methods for managing the execution of the job.
    - jEvent is in charge of re-distributing job events and performing the required action to send and receive signals.
    - jMigrator and jCheckpointing manage migration and checkpointing functionality guided by job requirements. During the job submission the user/application will specify migration and checkpointing hint that will be used as a basis for these services.
- Resource Management Services (RMSs)
    - Negotiator. Each resource has associated one Negotiator service that is aware of the local policy associated to the resource (one resource could be in more than one V.O.). The Negotiator is a key service in XtreemOS since it will integrate local resource policies with V.O. policies, being one of the main contributions of the project.

Depending on the local policies the Negotiator of a resource will offer exclusive access to the resource, different scheduling policies, etc.

- rMonitor is used to get dynamic information about resources, and to report the values to other services through push and pull mechanisms.

- rAllocator performs the required actions when submitting the job to a resource and when adding or releasing resources. It will contact the V.O. Manager for updating resources accounting associated to the job.

- Global services

  - JobDirectory is a distributed Information Service that maintains the list of jController addresses. Since each job is created by a grid user in the context of one V.O., the JobDirectory services manages the association (jobID + jController_address + user credentials). This service will be accessed each time the jController address associated to a jobID is required. It will include job access rights validation associated to user credentials. The JobDirectory is a key component in the AEM architecture because it is the only one that has a global list of jobID's in a VO (with the security information associated), being the only way to get the list of jobID's of a user in a VO.

  - jResMatching is a resource matching global service used to store and gather resource information, which makes sense to cache in order to reduce query load on the system. It provides access point - API to the jScheduler, and provides a set of candidate resources, which match requirements passed to the service.

Overview of the AEM architecture

The architecture of the AEM component is mainly divided into two parts: the user or client side and the server or system side, see Figure 1. The user side basically offers the API to access AEM services. At this point of the project, we call the implementation of AEM services "system" even if this part of the AEM is not executed in root mode.
The user/client side includes the implementation of the command line and third party libraries such as SAGA to be built on top of the Basic XtreemOS API. This layer mainly pre-process arguments, calls the system side and post-process results.
The system side is mainly composed of the XOSD (XtreemOS Daemon) and some distributed services such as the JobDirectory and the jResMatching. All the messages going in and out of the node are received and generated through the XOSD in order to minimize security problems concerning opening ports.

Figure 1AEM Architecture overview

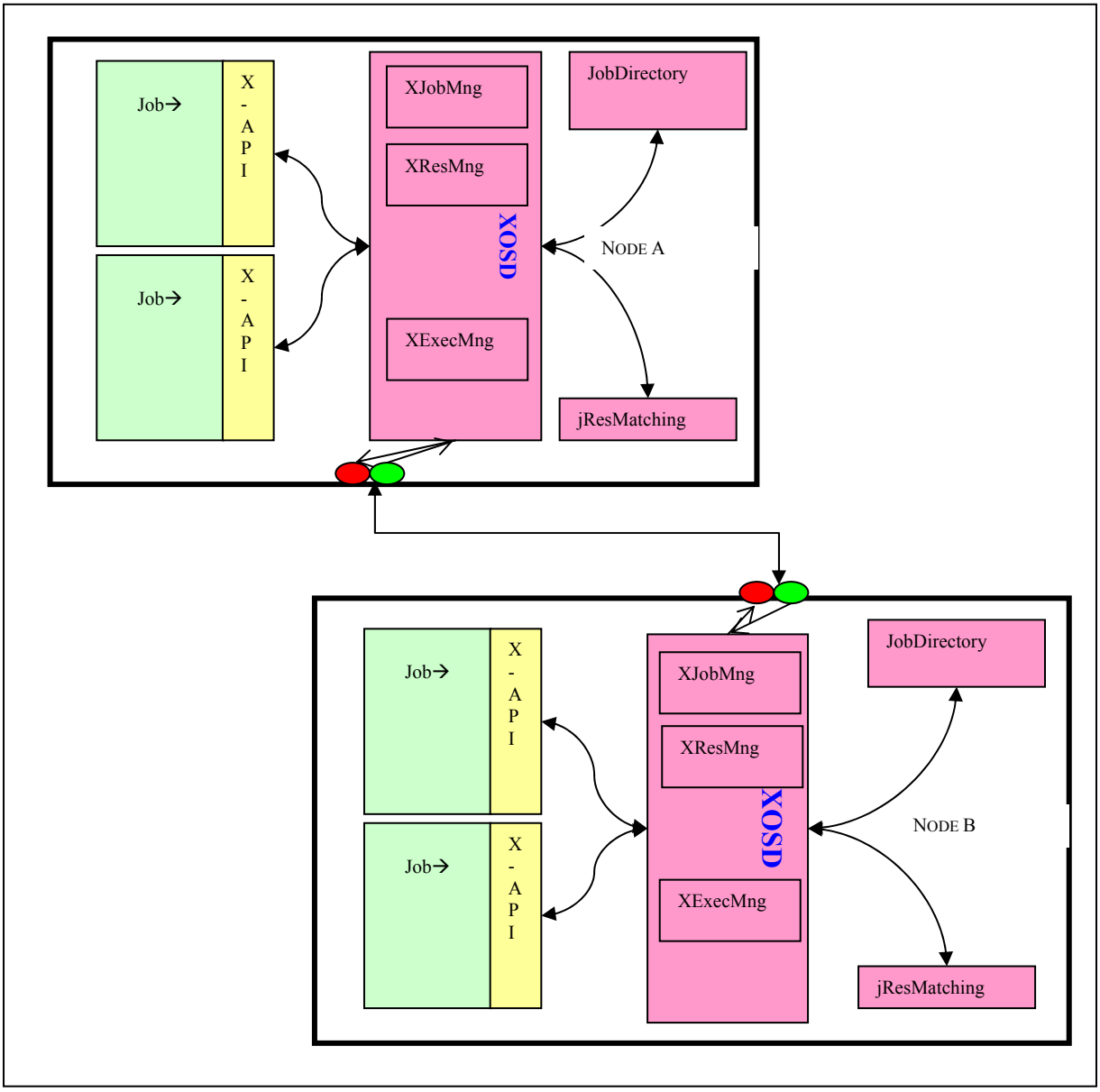
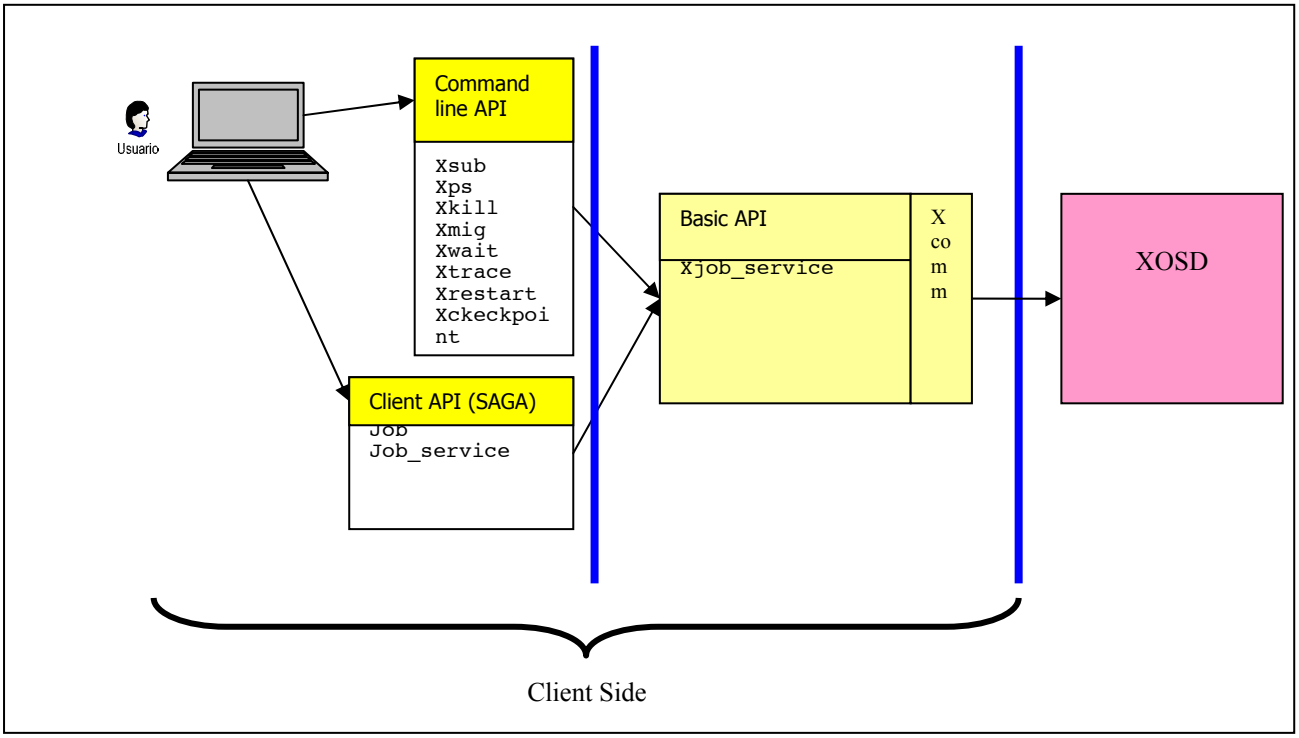


Figure 2 Client side of the AEM architecture



Client side architecture

The client side of the AEM architecture is divided in three layers: client API, Basic Functionality and Communication layer, see Figure 2. The client API includes command line API and implementation of user libraries such as SAGA [5] that will be built on top of the XtreemOS Basic API. The Xcomm layer provides the required functionality to contact with the XOSD server.

We have followed a similar approach to that existing in Linux systems where the client side mainly offers some libraries that implements the initial process of arguments, the call to the system, and the post-process of results (such as the errno error code). In XtreemOS, *system call* arguments can be a bit more complex, for instance having a XML schema when submitting a job (i.e. a JSLD file).

The Basic layer provides the functionality described in D3.3.1 [6] with some small modification introduced to support the user requirements in terms of resource management, and some missing functionality in terms of job checkpointing and migration. Table 1 summarizes the XtreemOS basic functionality. We just have included the name, the context to which the function applies (job, global,..), whether it is new or have been modified with respect to the content of D3.3.1 and a brief description.

Table 1XtreemOS Basic API [1]

Name	Context	Changed	Description
Job Submission			
Create_job	Job	yes	Create a new job
Run_job	Job	new	Start an existing job
createProcess	Job	no	Create a process for a job in a resource
Job Management			
jobControl	Job	no	Encapsulates waitJob, endJob, SuspendJob, ResumeJob, CancelJob and ChangeUID
updateJobRequirements	Job	no	Includes: reqMoreResources, releaseResources and migrationHints
exitJob	Job	no	Finish all the processes of the Job
attachProcess	Job	no	Adds a UNIX process to a previously created job
sendEvent	Job	no	Send en event to a Job

[1] New functions are still under internal definition, they could be modified

Name	Context	Changed	Description
waitForEvent	Job	no	Block a process until it receives the specified event
addEventCallback	Job	no	Subscribe to a job event
Job Migration			
migrateJob	Job	New	Migrates an existing job from a set of resources to a new one specified by a set of resource requirements (potentially to pre-reserved resources)
restartJob	Job	New	Restart a job context. Creates a new job and re-scheduling is done.
checkpointJob	Job	New	Checkpoint a job.
Job Monitoring			
getjobIds	System	no	Return the JobIds matching the conditions
getJobInfo	Job	no	Return the information of the list of jobIds receives as parameter
monitoringControl	Job	no	Start, stop or change level of monitoring
addAtributte	Job	no	Add attribute to Job Info
addCallback	Job	no	Subscribe to a certain characteristic (CHANGE, EQUALTO, GREATERTHAN, LESSTHAN) of a monitoring attribute
Resource Management			
ResMatching	Global	New	Returns a set of resources list that fits in a certain resource requirements
setReservation	Resource	New	Creates/modifies a resource reservation
rmReservation	Resource	New	Removes a resource reservation

The Xcomm layer provides the communication functionality between the client side and the system side (XOSD). All the messages sent by the client side to request a service and the responses that are received from the system side are handled by this layer. All the communication is encapsulated in this element, so that there is a unique function to send requests to the system. Once the communication is initialized to send a request for the information of a job is as easy as one line of code, i.e. to ask for the information of a job, where you know the jobId, the getJobInfo request must be sent to the XjobMng as follows:
jobInfo = xcomm.sendrecv (new RPCreceiver (XJobMng.class.getName(), "**getJobInfo**", RPCTypes.withreturn), **jobid**);

System side architecture: XtreemOS Daemon (XOSD)

The system part of the XtreemOS AEM component includes the implementation of the AEM API and data management of job information and dynamic resource information (static information will be managed by Node Management services [7]).
Services described in section 0 are offered by one XOS Daemon (XOSD) per node. Each XOSD contains a subset of the total amount of information about jobs actives in the VO (concerning execution management). Each XOSD will manage[2] information about jobs submitted[3] in the node, running in the node, and about the node itself (one node is a resource) .For the rest of the cases, the XOSD will either know who is the owner (the XOSD *owning* the data) or will ask the JobDirectory which stores this information, see section 0.
All the requests for a service are received by the XOSD (as this is the only entry point to the system side). Each time the XOSD receives a request for a service, it redirects the request to the XJobMng, the XResMng, or the XExecMng (each one will be a thread). These three execution flows offers the API for all the jobs and resources in the GRID.
The current version of the XOSD is implemented in java except some pieces of the XExecMng that are implemented in C because we need a fine grain control that is not provided by the Java Virtual Machine. The XExecMng is the one that uses local services (system calls) to create processes (fork&exec), manage Linux signals (signal, kill, etc), etc. Since we want to provide a deeper control on resource usage, we can not let the

[2] This is the default data managed by each XOSD, but information concerning the jController services could be migrated in advanced versions due to node failures

3 We will also refer to these jobs as controlled or owned by this node, it is the information associated to the jController

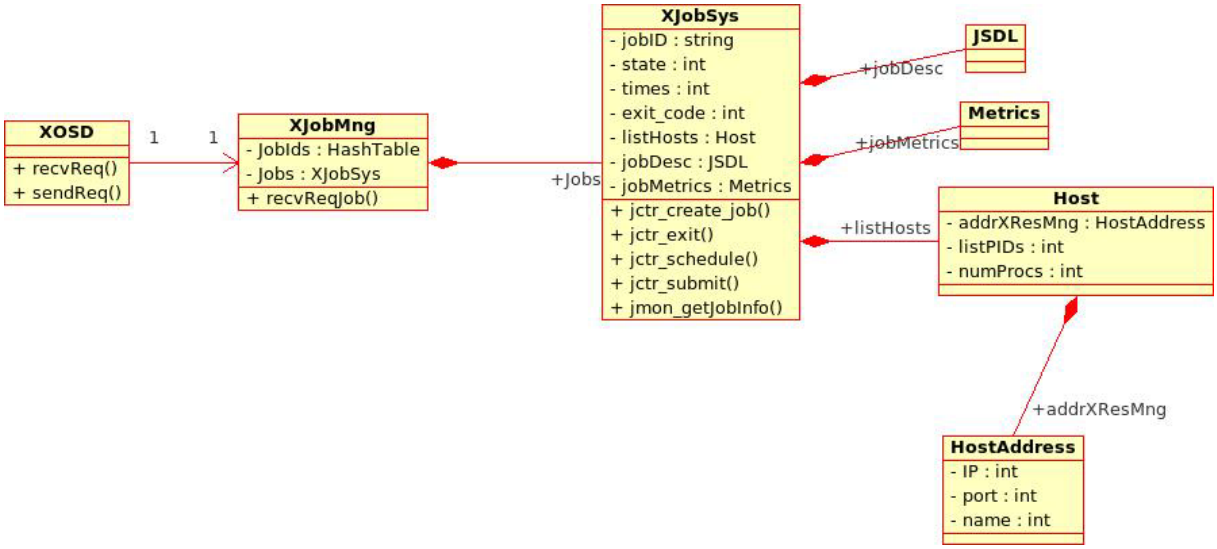
implementation of these operations to the java virtual machine, which doesn't export, for instance, the pid of created processes. Because of that, we decided to implement the usage of Linux system calls using the jinni mechanism [REF].

XJobMng

The XJobMng is a thread of the XOSD that receives and processes all the requests for the jController, jMonitor, jScheduler, jCheckpointer, jMigrator, and jEvent. The jController and jScheduler are fully implemented by the XJobMng whereas the rest are distributed between XJobMng and XExecMng; the basic functionality (such as create a process or send a signal) is mainly implemented by XExecMng while the XJobMng controls or aggregates data in a job basis. The basic idea is that the XJobMng *decides* and the XExecMng *executes* its decisions. XJobMng holds and stores data for those jobs *controlled* by this node. By default, a job submitted in Node A will be *Controlled* by Node A.

XJobMng manages the XJobSys class where main information and functionality concerning jobs is represented. Information can be static such as the JSDL schema submitted with the job or dynamic such as the list of metrics and the list of hosts where the job is running. The list of methods provided is the same exported in the Basic API plus some internal methods. Figure 3 shows the main data and methods managed by the XJobMng.

Figure 3 XJobMng Class diagram



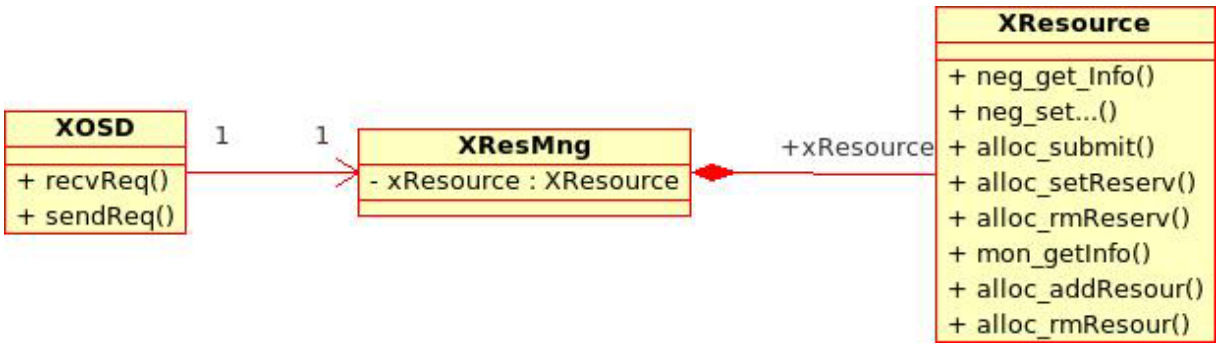
Those requests to JMSs concerning jobs not controlled by the node are redirected to the node owning the jController of the job. This information is stored at the JobDirectory.

XResMng

The XResMng handles all the requests concerning resources. It exports the rAllocator, the Negotiator, the jResMatching and the rMonitor interfaces. The XResMng of Node A exports functionality and data for Node A and redirects calls to the rest of nodes.

Data and methods used by the XResMng are implemented by the XResource class. All the static information concerning the resource is not stored here because it will be maintained by the Node Management server provided in WP3.2 [7]. Figure 4 shows the main data and methods associated with the XResMng.

Figure 4 XResMng Class diagram



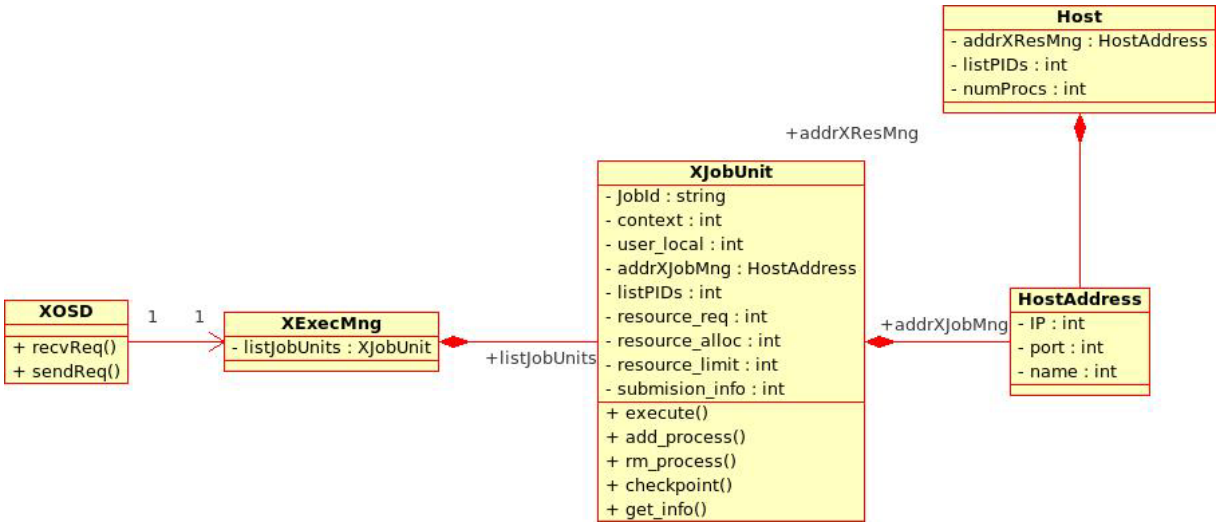
Dynamic information will be gathered using a plug-in mechanism in the rMonitoring service that will be instantiated when required. Information gathered on the resources and the processes, associated with Grid jobs, have to be retrieved from the operating system. E.g. information about process status can be retrieved through

procfs (Linux pseudo file system). Besides using already available information on the resource and the process, XResource will also support dynamically loaded plug-ins for gathering various system parameters (i.e performance measurements such as MFLOPS, or Cache misses that can be constructed based on standard libraries such as PAPI [8]).
The information offered by the XResMng doesn't need to be fault tolerant because if it is not available, it will mean that the resource is down.

XExecMng

The XExecMng implements the functionality of those services involved in the execution of the job (all or part of the functionality): jExecMng, jEvent, jMonitor, and jCheckpointer. The XExecMng manages those jobs (or units of jobs) *running* in the node. The XJobUnit class shown in Figure 5 exports data and methods associated with the runtime details of a job.

Figure 5 XExecMng class diagram



JobDirectory

The JobDirectory will be implemented using a Directory service. This Directory service will be implemented over a Distributed Hash Table (DHT) [9][4]. From the AEM point of view it provides global information concerning jobID's, user credentials and XJobMng contact address.

jResMatching

The purpose of the jResourceMatching service is to provide information regarding the available resources to the jScheduler through a set of well defined APIs.
The information gathered by the jResMatching service can be coarsely divided into two groups. The first group of information is of static nature, which does not change frequently (i.e. CPU frequency), and the second group consist of highly dynamic information (i.e. current CPU load, queue length, …). Based on this fact, we have also divided the approach on how to find the information.
Searching for **static information** is done by the Node Management service, which is a gossip based service, devised in WP3.2, where the main purpose of the protocol is to provide reliable service, which gracefully handle intermittent members, high churn rates at small overhead network traffic. The Node Management service will provide an API, which will be accessed by jResMatching service in order to gather the candidate list of resources, which will match query requirements in static attributes.
After the initial list is retrieved, the service will check the availability of resources and filter away those which do not match with the search criteria of dynamic properties. At this step the check is also performed on resource responsiveness and matching permissions with VO policy manager (devised in WP3.5).
The so gathered information about resources is returned to jScheduler, which later on allocates resources to specific tasks through rAllocator.

[4] This decision is still pending. It will be decided based on the work done in WP3.2

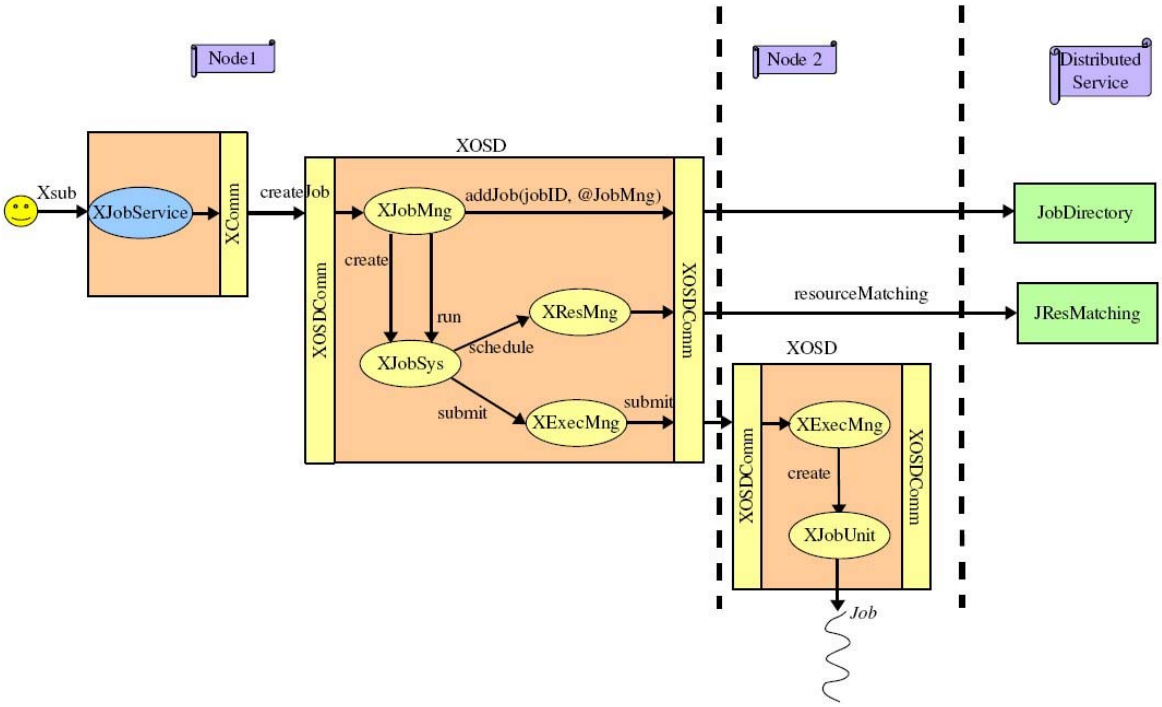
Resource description – GLUE scheme

We have decided to use GLUE scheme [2] to describe resource properties and characteristics. There are many different approaches to describe and search resources (RDF [3], ClassAdds [1], CIM [4]…), but we have decided to use GLUE scheme based on the fact, that we would like to have simple light weight resource description, which will allow performing efficient searches, and, in our opinion, RDF is too generic and CIM too complex. In our opinion GLUE scheme is providing just the right amount of information about a resource with the right amount of flexibility.

Interaction between AEM sub-components: examples

Figure 6 shows a simple Xsub command execution. The first part executed in the client side is where the command is launched by a user, the JSDL file is parsed and the information is sent to the corresponding XOSD via the Xcomm layer. The request is received by the XOSD that redirects it to the XJobMng.
When the *create_job* request is received by the XJobMng a JobId is given to the Job and it is registered with the address of its XJobMng (@JobMng) in the JobDirectory (represented in the picture outside any node because is a distributed service).

Figure 6 Basic sequence of Xsub command execution



Once the contact information is in the JobDirectory a new XjobSys object is created to store the static information of the job and then a run order is sent to it. The JobSys will ask for available and suitable resources to the XResMng that will use the jResMatching distributed service to find them.
Once the resources are selected, the XJobSys.submit method will send a submit request to the XExecMng including the address of the target resources. The XExecMng will forward the request to the corresponding XExecMng via the XOSD.
When the XExecMng receives a submit order (addressed to it) it creates a XJobUnit object to store the dynamic information corresponding to the part of the Job being executed in that node and starts the execution.

Figure 7 Basic sequence of Xps command execution

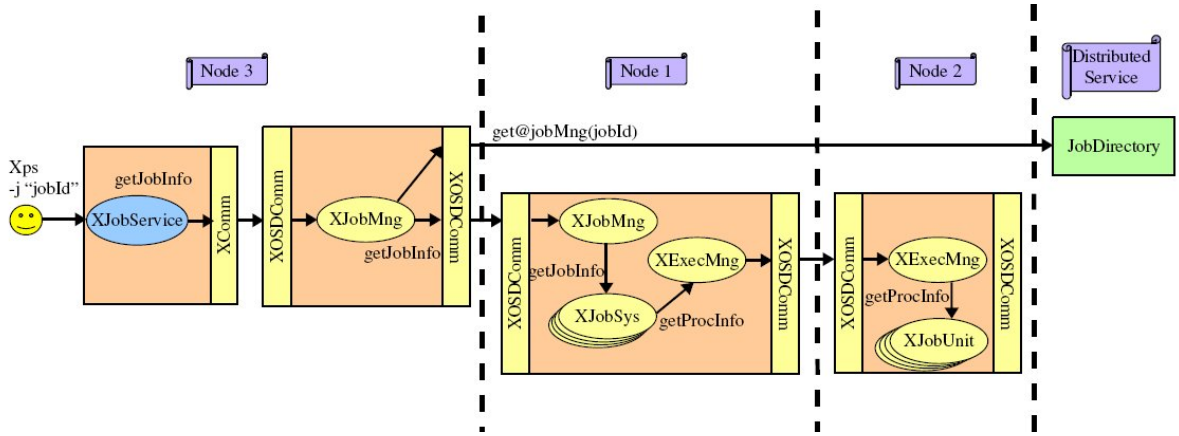


Figure 7 shows the basic sequence of steps when executing a Xps of the previously submitted job X (as shown in Figure 4). The sequence starts when a user launches an Xps -j <jobID> and this becomes a getJobInfo request in the XJobService.

The getJobInfo request is redirected to the XJobMng of the node executing the Xps (in our example Node 3), the XJobMng checks if the job is managed by him (i.e. the Xsub was executed in this node). If not (in our example the manager of the job is Node 1) the XJobMng will ask the JobDirectory the address of the node that is the manager of the job.

When the address is known a getJobInfo request is sent to the XJobMng of the node managing the job. Once again the XJobMng checks if the Job is in his list of managed jobs and passes the getJobInfo request to the corresponding XJobSys.

The XJobSys will send a getProcInfo request to the XExecMng for each different node that is running a process of the job (only one process in this example).

When the XExecMng receives a getProcInfo checks if he is the target of the request (comparing the target address and his own) if not, redirects it to the XExecMng of the target node (Node 2 in our example). Once the XExecMng has received the getProcInfo request of a process running in its node it asks for the process information to the corresponding XJobUnit.

Integration of the AEM with other XtreemOS components

Virtual Organization Management

AEM services are operated in the context of a VO with the support of VO management services (aka. VOM). VOM is a logical representation of a collection of VO infrastructure services, which include, but not limited to, membership, identity, and policy management services.

General security requirements for AEM services

There are two fundamental security requirements for all AEM services: 1) only registered users are allowed to access AEM services. A registered user means that the user is registered with a VO membership service; and 2) fine-grained and scalable access control should be in place to ensure AEM services are available to authorized users and they can perform efficiently even in the presence of a large number (e.g. thousands) of VO users. There are two levels of controls: one is the control of AEM services themselves (e.g. jController and JobDirectory), and the other is the control of the information provided by AEM services (e.g. accounting or monitoring information).

Security requirements for job submission

The security services should ensure that: 1) only authenticated VO users are allowed to submit jobs so that the use of VO resources is accountable; 2) the selection of VO policies should be context-aware. That is, VO policies should be associated with the context of job submission. Contextual information may include users attributes (e.g. role, location, origin organization); and 3) cross-VO job submission should be possible. This last requirement is probably the most complicated because it requires federation of security services (e.g. identity and policy) from multiple VOs across multiple administrative domains.

Security requirements for resource matching

One of the most important aspects of AEM resource matching is its incorporation to VO policies for resource selection. The security mechanisms for resource matching should ensure that resources selected for application

execution conform to VO policies throughout the entire lifespan of the execution. The major challenge here is to ensure such conformance in the presence of dynamic resources or application execution across multiple resources.

Security requirements for job accessing

The security services for accessing remote jobs are mainly for applications that require interactions during application execution. They should ensure that: 1) users are attached to the correct sessions of application execution; and 2) users are given the same level of privileges in all the sessions; and 3) once a job is completed, users' credentials should be safely revoked and their privileges be removed across the entire VO .

Security requirements for job execution

A job executing on a local resource should be granted appropriate access rights to the resources it needs, even in the presence of dynamic resource changes or dynamic job resource requirements.

Security requirements for accounting

In the context of AEM, the security services for job resource accounting should ensure that resource usage is recorded with accuracy so that resource consumption is accountable. By accountability, we mean that the accounting information should associate with individual VO users [3]. The recorded information should also be made available to authorized VO entities so that they can run follow-on services (e.g. reputation or billing) after a job is finished.

Node Management

During the job submission, the AEM has to find enough resources that fit into the job resource requirements. Some of these requirements concerns attributes of resources that are invariant during the whole life of the resource modified with a very low frequency. Having this preconditions, special mechanisms can be applied to optimize the search of resources, even with a very high number of resources. Node management services provided by WP3.2 will offer AEM the first searching mechanism.

Local Resource Management

AEM negotiates with resources that have their own local schedulers and their own local policies (a resource can be in more than one VO). We envision the interaction between Resource Management Services (RMMs) and Local resources manager (LRM) as like an API that hides the different features supported by local systems to the scheduling process, and that is able to understand situations, such as co-allocations involving several resources that are unknown to LRM.
We also consider two possible scenarios concerning LRMs : LRMS offering queues or not. In the first case, it is possible that the LRM offers resource reservations that can be used during the negotiations phase. In that case, the AEM will use, and potentially extend, the services offered by the LRM. In those interactive LRMs, the AEM system will offer the complete negotiation service. Depending on the resource, it will be able to offer more or less scheduling features. These scheduling characteristics will be part of the resource selection. For instance, if a job request for exclusive access to resources, those nodes that do not offer this feature will be filtered from the list of eligible resources.

Interaction between AEM and other XtreemOS components: Simple job Submission use case

AEM services can be seen as part of the whole VOM. After investigating the interactions between AEM and VOM, we have defined the execution flow when executing a job submission which is our basic use case, see Figure 8 .

1. Ideally, there is a first step where user asks for a VO User credential to be used when submitting the job. However, but this step is out of our control.

2. Job submission through Xsub command. It includes the job definition and the context (i.e. Xsub –f job.jsdl –VO –cred user_cred eScience –role programmer)

3. XJobMng (jController) validates the VO user credentials with X-VOMS. X-VOMS contacts with HAA, checks the VO user credentials, and issues an XtreemOS user certificate (called XOS-Cert) to the user. From this step on, the VO user credentials, in the form of XOS-Cert, will be associated to the rest of the scheduling process to allow the user to authenticate to other services, such as the VO policy service and XResMng-Negotiator.

4. XJobMng (jController) registers the new JobID associated to the VO user credentials into the JobDirectory.

5. XJobMng (jController) starts the scheduling process (jScheduler)

6. XJobMng (jScheduler) ask to the XResMng (jResMatching) for a set of list of resources that satisfy job resource requirements.

7. XResMng (jResMatching) contacts with Node Management service to get a first set of list of resources that match wit static requirements

8. XResMng (jResMatching) filters with VO Policy service those lists of resources that follow VO Policies. A pre-selected set of list of resources is returned (end of step 6)

9. XJobMng (jScheduler) authenticates with the XResMng (Negotiator) involved and starts a negotiation, asking for specific reservations. The scheduling algorithm is applied resulting in a resource selection.

10. XJobMng (jSCheduler) starts the job submission to the selected list of resources through the XResMng (rAllocator)

11. XResMng (rAllocator) notifies the VOM about the job submission, and the specific agreement with resources selected.

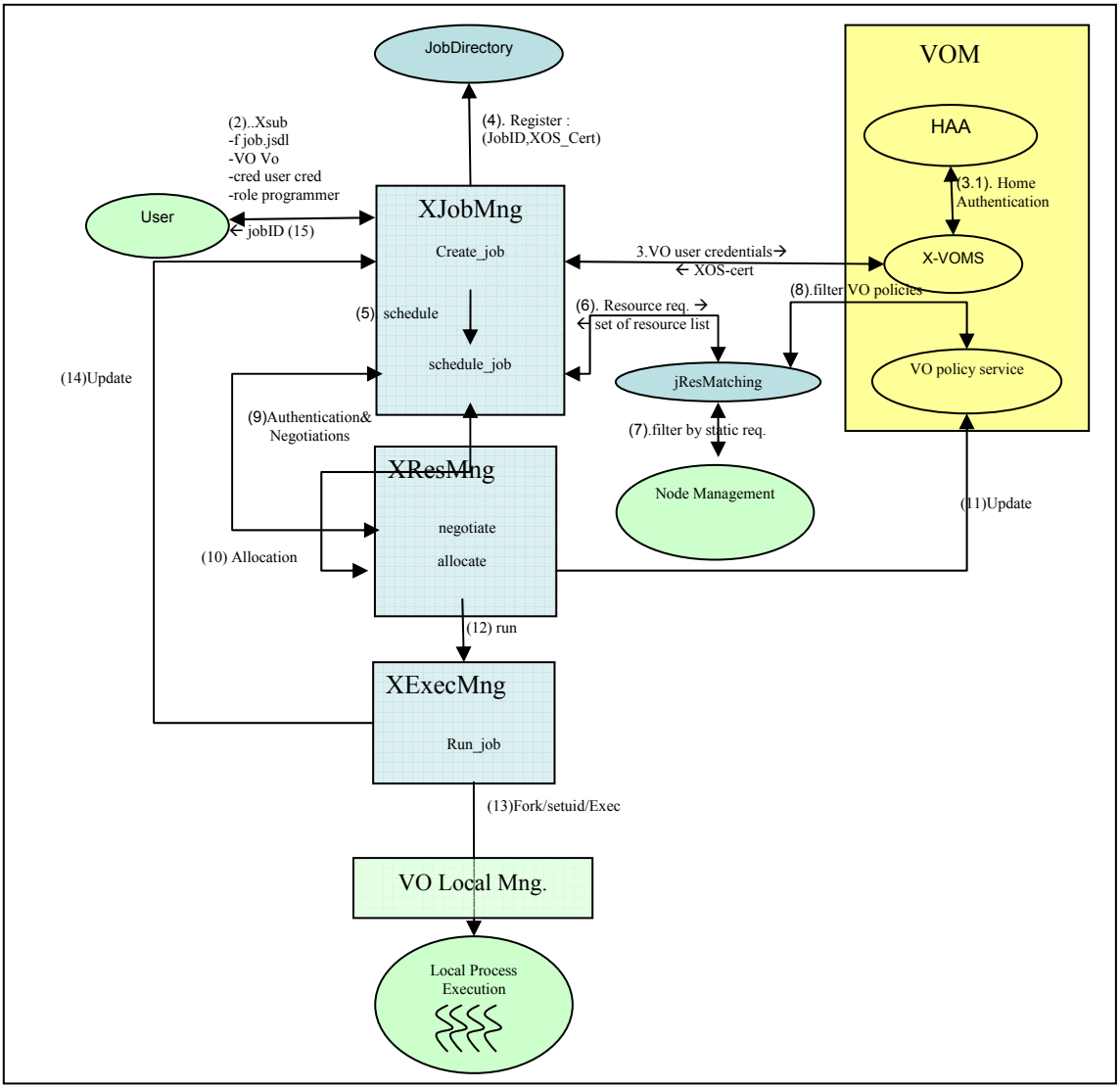
12. XResMng (rAllocator) starts the execution through the XExecMng (jExecMng)

13. XExecMng(jExecMng) maps the user's Grid identity to a local UID/GID and performs the fork/setuid/exec operation

14. Required local information about pids, etc is returned to the jController

15. Submission status and jobId is returned to the user (end of step 2)

Figure 8 Interaction between AEM and VOM: Job Submission



Pending architectural issues

Although most architectural decisions have been taken, there are still some issues that need further discussion at this point. In this section we will detail these issues and the reasons for the delay.

Fault tolerance

We have mentioned that we have two services that need to be fault tolerant: JobDirectory and jController. Currently, we plan to use the Virtual Node solution proposed in WP3.2, but we are not sure if using it will imply some changes in the current architecture (i.e. need to isolate the jController functionality in a single process). Once the Virtual Node functionality is ready, we will work on ways to use it to achieve fault tolerance trying to minimize the changes to the current architecture.

JobDirectory

Currently we intend to use the directory service offered by WP3.2 to implement the JobDirectory. The problem is that as of today, there is no clear position in WP3.2 on what will be really delivered and whether it will fit our needs. We need to wait the outcome of the discussion in WP3.2 to take decision on how to implement it.

jresMatching

Although we have presented the architecture for jResmatching that will be used in the first version, it will probably change once we want it to maintain some state and be able to solve more complex queries.

Publish/subscribe

Finally, we are initially thinking on using the publish/subscribe mechanisms provided by Wp3.2 to handle many of the communication between jobs (i.e. signals), but we still need to evaluate its behavior before taking a final decision.

References

[1] M. Solomon, The ClassAd Language Reference Manual, http://www.cs.wisc.edu/condor/classad/refman/

[2] S. Andreozzi, S. Burke, F. Donno, L. Field, S. Fisher, J. Jensen, B. Konya, M. Litmaath, M. Mambelli, J. M. Schopf, M. Viljoen, A. Wilson, R. Zappi, GLUE Schema Specification version 1.3, Draft 3 - 16 Jan 2007, http://glueschema.forge.cnaf.infn.it/Spec/V13

[3] W3C, RDF Vocabulary Description Language 1.0: RDF Schema, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

[4]CIM Schema Version 2.14, http://www.dmtf.org/standards/cim/cim_schema_v214

[5] T.Goodale, S. Jha, T. Hielmann, A Merzky, J. Shalf, C. Smith. A Simple API for Grid Applications (SAGA). http://forge.ggf.org/sf/docman/do/listDocuments/projects.saga-core-wg/docman.root.drafts

[6] XtreemOS consortium. Requirements and specification of XtreemOS services for Application Execution Management http://www.xtreemos.org/publications/public-deliverables

[7] XtreemOS consortium. Design of an Infrastructure for Highly Available and Scalable Grid Services D3.2.1. http://www.xtreemos.org/publications/public-deliverables

[8] Performance Application Programming Interface. http://icl.cs.utk.edu/papi/

[9] Puppin D. and Moncelli S. and Baraglia R. and Tonellotto N. and Silvestri F. A Grid Information Service Based on Peer-to-Peer. in Proceedings of EuroPar2005. Springer LNCS 2648/2005