Project no. IST-033576

XtreemOS

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

Requirements and specification of XtreemOS services for job execution management
D3.3.1

Due date of deliverable: November 30th 2006
Actual submission date: December 20th 2006

Start date of project: June 1st 2006

Type: Deliverable
WP number:WP3.3
Task number (optional):

Name of responsible:Toni Cortes
Editor & editor's address:Julita Corbalan
julita.corbalan@bsc.es

Version 1.0/ Last edited by / Date

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	
RE	Restricted to a group specified by the consortium (including the	
CO	Confidential, only for members of the consortium (including the	

Keyword List:

Revision history:

Version	Date	Authors	Institution	Sections Affected / Comments
1.0	10/11/06	Julita Corbalan + Toni Corte	BSC	Delivered for review
1.0	10/11/06	Gregor Pipan	Xlab	Delivered for review
2.0	23/11/06	Julita Corbalan + Toni Cortes	BSC	Reviews introduced, some sections pending to introduce
2.0	23/11/06	Gregor Pipan	Xlab	Reviews introduced, some sections pending to introduce
3.0	28/11/06	Julita Corbalan + Toni Cortes	BSC	Final version

Executive summary

In this deliverable we present the main services proposed for job execution management, as well as some preliminary interfaces and the interaction between these services.

Before defining the needed services, we have summarized the requirements applications have as far as job execution and resource handling are concerned. This summary comes mostly from the work done in WP4.2 where application requirements have been studied.

Regarding the main entities that take part in job execution, we have defined jobs and resources.

- A job is one or more Linux processes that collaborate to achieve a certain goal or objective. The idea is that a job is a resource allocation unit and resources are consumed by the processes (or their threads). In addition, we have detected that it would be desirable that kernel has knowledge of this abstraction in order to allow enforcement in the correct use of the resources.

- A resource is any physical or virtual (logical) component of limited availability within a computer system. These resources, besides static characteristics will have some dynamic ones that will also be used in the allocation process. Finally, we will have to take into account that resources can be allocated only party (some processors) and that they may belong to different VO.

Once the main entities have been defined, we have presented a preliminary interface of how to execute, control, and monitor jobs. This interface has been proposed both at system call level (understanding system call as a call to XtreemOS, not necessarily to the kernel) and at command level. Interfaces with the rest of services have not been defined, but their interaction has, as will be seen later.

Regarding the services themselves, we have defined a set of services that allow us to offer all functionality requested by users (from requirements) and allow a good level of control (which is not normally the case in Grid systems). In addition, we have proposed these services taking scalability into account and thus most of these services are not long-term running and do not have a global view of the system. The proposed services can be summarized as follows:

- jScheduler that decides the best resource selection for the job. It does not have a global view of the system and is only active from the submission to the queuing of the job in a resource (or set of resources). We will see that there is no such thing as a global grid scheduler (it would not scale)

- LTScheduler that controls local resources and manages lists of local jobs. This service has a local-system view and runs permanently. If such a service already exists in the resource, our version will only implement a layer interface to negotiate scheduling agreements.

- jController that control scheduling agreements during the whole life of the job and acts like a gateway for the job. It has a local view (the job) and lives for as long as the job exists.

- jExecMng that manages the efficient utilization of the allocated resources. For instance, this process decides the correct placement of processes to resources to improve performance.

- jMonitor that monitors the status, performance and resource consumption of the job.

- jEvent that manages events (i.e. Linux signals) to the job.

- JobDirectory that manages a list of jobIDs and the contact information (jController). This service will be needed to implement ps-like functionality. This service will be the only one that has a global view of the jobs in a VO.

- rMonitor that control resource dynamic information.

- jResourceMatching that performs the matching between a list of resources and the job requirements, taking into account the dynamic information of the resources.

- rAllocation that takes care of allocating resources to a job.

Finally, to clarify any potential missing aspect, we present a set of cases to show the interaction between all these services.

Table of Contents

1. Introduction

This document presents a list of XtreemOS services for Application Execution Management (AEM). This list must cover necessities of users and jobs in executing new jobs, controlling their execution, monitoring their execution from different perspectives (resource consumption, performance, status, etc). The Application Execution Management system of XtreemOS is not targeted to specific users or types of jobs, so it must be as generic and flexible as possible, differentiating the services or functionality from details such as job specification. We propose this list of services taking into consideration what users are expecting from a grid system and what a grid system has to offer in terms of security, scalability, efficiency, fault tolerance, and management of dynamicity and heterogeneity.

From the point of view of users (or the job itself), what is expected from the Application Execution Management services is a set of facilities that allows to efficiently exploit the advantages of executing jobs in a grid. That is, the huge amount of resources should be available in an *easy* and *efficient* way. By *easy* we mean that XtreemOS must support remote execution of jobs submitted in a standard UNIX way without modifications in the binary code, and with minimum user intervention when submitting it. Obviously, this approach reduces the amount of information provided to the Application Execution Management system to the *default* context values. However, our goal is to combine these *default* values with additional user-provided *hints* about *resources* and *scheduling* goals in order to best support the job[1]. We think that hints such as "my job consumes a lot of cpu", or "memory" are something that we can expect from users. By *easy* we also understand that we must provide tools to users (and jobs) to monitor job execution. These tools must be as easy to use as executing a UNIX ps command.

The second goal of this WP from the point of view of a job is to offer an *efficient* job execution management, that selects the most appropriate set of resources, and automatically migrates them in case of resource failures. The characteristics of the Grid (dynamicity and heterogeneity) makes it desirable for the AEM to hide (as maximum as possible) these issues to users.

From the point of view of the system, the AEM has to guarantee the access to authorized resources and their limited utilization. Jobs are executed in the context of a grid user and a Virtual Organization. The AEM will ensure the utilization of allocated resources and will offer the required services by the jobs. In terms of scheduling, the system has different goals from the users. Once the job is submitted to the local system, local scheduling policies will be applied in order to maintain a certain independence from the whole system. Local systems expose their resources but they want to maintain the control. To reach the global objective of managing jobs, we will offer a set of services and we will use services from other parts of XtreemOS.

To this end, we first analyzed the list of requirements coming from jobs (WP4.2). We also discussed which requirements will be fulfilled in XtreemOS Application Execution Management (AEM) services, and which ones will be just partially supplied. We also discussed with the rest of parts of XtreemOS in order to see which functionality were expected from internal AEM services.

---

[1] In this document we use the term job rather than job, but they are the same.

The rest of this document is organized as follows: Section 2 analyzes requirements coming from WP 4.2, where users have made an effort enumerating requirements in terms of scheduling, monitoring, tracing, resource management, etc. Section 3.1 presents our proposal in terms of job management services. We first present a prototype of XtreemOS API for managing job execution and second we present the list of services that will be used for implementing this API. Section 3.2 presents the list of resource management services required for providing required services for users and for internal use of job management services in terms of resource matching and monitoring. Section 3.3 presents some use cases based on the API suggested and relationship between job management and resource management services. Section 3.4 enumerates some dependencies detected with other workpackages. Section 4 presents some related work, and finally section 5 concludes this document.

2. Use cases coming from WP4.2

This section presents the list of requirements coming from the WP4.2. These requirements have been generated as a result of a questionnaire filled by partners involved in WP4.2 and related to the jobs we are going to use in XtreemOS. We have used as starting point requirements generated and discussed in the XtreemOS meeting in Düsseldorf, however, we will adapt our services to new requirements.

2.1. Job requirements

We have grouped job requirements to those related to monitoring, scheduling and control.

2.1.1. Monitoring requirements

- R51, R52, and R54 **The monitoring system should provide a variable amount of detail based on the job being run (either job or use defined**). The resource consumption should be viewable during the whole job execution run time and there should be a message/mail notification for changes in the job/workflow/environment or when certain stages have been reached.

- R52 **Resource accounting**. It must be possible to record the usage of a resource by a user at any given time. A way to implement / use special cost models must also be provided.

- R55 **Tracing system**. Tracing should be provided for both the job execution and the resources being used. Different levels of details are required for both of these, depending on the job, so a mechanism should be in place to set these.

Based on these requirements, we can see that jobs (and users) need to know “what they are doing”. The WP3.3 has to provide functionality to:

- **Monitor job status:** Based on questionnaires it is not totally clear which kind of information would be desired by jobs (or users) but our initial proposal will consider monitoring changes in: job status and job performance metrics such as MFLOPS, Memory Bandwidth, Storage Bandwidth, or Network Bandwidth.

- **Monitor job resources:** In addition, resource usage (how many and which nodes and cpus per node, Memory, Storage, Networks, etc,) should also be monitored. Once again, the level of detail of this monitoring is not clear from the user requests.

- **Notify job changes**: Basic AEM services will provide notifications via callbacks triggered by job state or resource state changes. User-level jobs will be registered as callbacks and notify user-job for example via signals, e-mails, or sms.

- **Record monitored information**: Based on previous notification, we will provide a basic tracing collection mechanism. This user-level service will collect the monitoring information and will generate an ASCII trace file format such as the Paraver trace file format [Paraver]. With this basic trace collection service, jobs (or users) can translate from the generated trace file format to the desired one.

- **Dynamic selection of monitoring/tracing level**: AEM services will include the dynamic selection of monitoring/tracing granularity level. However, from the requirements questionnaire we conclude that it will enough with three levels of detail: high, medium, low.

2.1.2. Job scheduling and control requirements

- R59 **Scheduling**. A co-allocation of job on resources of several different sites must be possible. The response times of certain jobs of specific customers should be emphasized while avoiding starvation of other jobs.

- R60 **Resource planning**. Reservation of resources for specific intervals is important and also being able to specify certain characteristics of the resources required (i.e. CPU speed and load, disk space, memory). Note that some jobs require changing of resource requirements during runtime. Applications will need detailed information to plan resources in advance and they should be able to confirm resource prior to allocation. Applications require running some parts in parallel on different resources (up to 1000 parts) and sequencing of execution must also be possible.

- R61 **Stopping execution**. It should be possible to stop the execution of an individual job as well as the execution of an entire workflow.

- R62 **Changing owner permissions during Application Runtime** Owner permissions should be modifiable during job runtime

- R63 **Spawn jobs to other V.O.s.** A job running on a virtual organization can spawn a job to another VO.

- R64 **Manual exploration and selection of hardware**. The job needs to be able to select the resources it uses.

- R65 **Co-allocation**. Certain jobs need co-allocation of resources on several different sites. It must be possible to prohibit the co-allocation on the job level, e.g. for security reasons.

- R66 **Distribution**. A job must be able to limit its geographical distribution.

These requirements can be grouped as:

- **Resource specification and selection**. Applications must be able to specify certain resource requirements such as number of resources, maximum distance between them, or geographical limits. This specification will be set when submitting the job and potentially during the execution of the job. Specific goals related to resource allocation must be considered: co-allocation allowed/not allowed and advanced reservations.

- **Priority considerations**. Scheduling policies must consider that not all the jobs will have the same priority.

- **Application control**. AEM services will provide deep control about the status of existing jobs. This control will include job dependences specification. Workload management will be offered as a user-level tool such as GRID superscalar [BLSP03] based on the XtreemOS functionality.

3. Job Management Services

This proposal document describes important details about objects that will interact in XtreemOS (such as users or jobs). We also present a list of services related to job management to be included in XtreemOS to ensure a good level of control in the job execution. By control we understand:

- A good service from the point of view of final users
- A good utilization of resources from the point of view of the system

Grid users will ask for resources to execute their jobs and potentially for certain scheduling *objectives* and resource *preferences*. AEM services will offer some of the needed resources and will ensure the job executes according to this agreement. Thus, we also have to provide mechanisms to validate that the execution of the job is performing as expected and the agreed times are followed.

From the point of view of the system, XtreemOS has to control that submitted jobs (executed with specific user-credentials) use just the amount of resources they are allowed to use. The system has to implement a fine grain control about real resource utilization and to provide policies and mechanisms to use efficiently the available resources.

3.1. Job Execution Management

3.1.1. The XtreemOS Job entity

We define job as following[2]: a job is one or more Linux processes that collaborate to achieve a certain goal or objective. Jobs are identified and defined at the Grid level by a global unique id., are associated to an identity, and use the identity credentials. One job is executed in one V.O. Jobs can have dependences between each other (i.e. job A has to finish prior job B starts it execution).

The **job asks for resources** and receives an allocation of these resources. A **process** is a part of the job that **consumes these resources** (much like the relation between processes and thread in a traditional Unix system). A resource is any physical or virtual (logical) component of limited availability within a computer system. A physical resource may be: CPU time, RAM, Virtual Memory, HD space, Network throughput, sensor, … Logical resources are among others: software, services, and files. Resources will be used by jobs based on limits defined by the resource, job requirements and V.O. policies.

We will therefore use job information present at the kernel level to avoid an excessive overhead for the resource control mechanism. Extending the existing job ID, currently used to do group control for SIGSTOP and SIGCONT group functionality, or adding a grid job ID to the kernel in order to control resource usage remains a design issue.

One of the motivations for improving the control of resource utilization is that in traditional grid environments, users specify job requirements and the system "*assigns"* resources to "jobs". The real situation is that underlying systems just know about processes and threads.

[2] The definition of job and resource will be validated with the final version of XtreemOS terminology. We use in this document the current definition.

Once the job starts its execution, the local system usually doesn't validate whether the job is using four or ten cpus (for instance when using threads), and the user is not able to check, through a system call, whether it is receiving ten or four cpus. One of the challenges of XtreemOS is to provide a real control of resources that are allocated to jobs. To force this resource control, we defend that either the concept of job must be known at the kernel level to validate the accesses to resources without requiring external confirmation, or enough information and support have to be exported to a higher level to enforce this resource management.

A job is executed in the context of a Virtual Organization. However, we consider the possibility that a job submit new jobs to different V.O.'s. The job is executed with the user privileges. In the same way as the user_id in Linux determines certain limits and rights, grid users will "authorize" jobs to have access to subsets of resources (potentially all of them).

XtreemOS job control will offer support for managing dependences between jobs, but not workflow management. When submitting a job, its definition could include a list of dependences with previously created jobs. In that case, the job will not start until all the precedent jobs finish. In the case a job was cancelled, all the dependent jobs will be also cancelled.

For monitoring purposes, we consider adding *names* to the definition of a job. In this way, it will be easy to associate a job with its information.

3.1.2. The job life cycle

In this section we describe the life cycle of a job which determines transitions generated by the different events generated by *actors* in the system. By *actors* we mean some component in the system that can generate a reaction in the system (state changes or execution of services). We will also describe the interaction about the different services when executing XtreemOS calls presented in section 3.1.

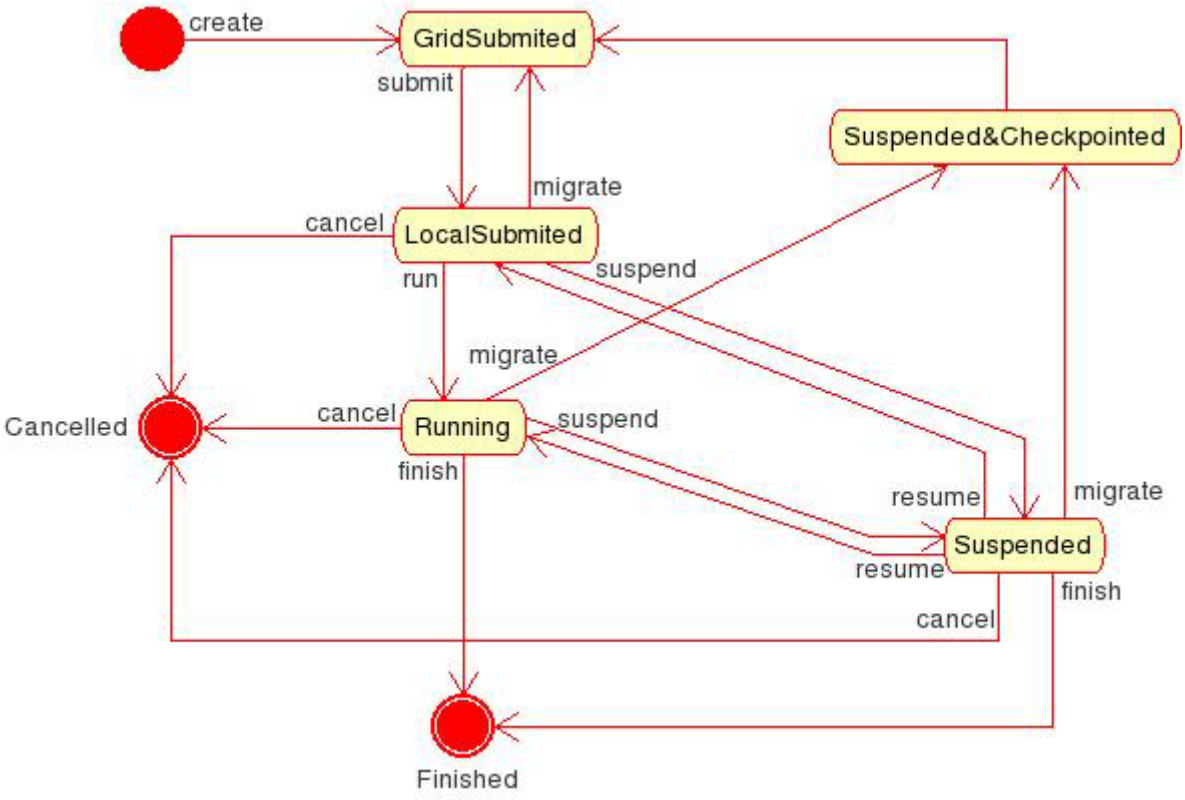

Figure 1 Job life cycle

Figure 1 shows the life cycle of a job. Jobs are created in GRIDSUBMITTED state. Once the scheduler decides for the best nodes submission for the job and the job is free of dependences, it is submitted to local systems and its state goes to LOCALSUBMITTED. In this state, it is under the control of local schedulers which might have different goals from the job objectives. Local schedulers optimize the execution of a list of jobs, not of a single job. The normal situation is that the job passes from LOCALSUBMITTED to RUNNING once the job is selected for execution, potentially it could be cancelled, suspended or migrated before starting.

3.1.3. Resource management

We define "resource" as following: *A resource is any physical or virtual (logical) component of limited availability within a computer system*. Where the resources are divided into several categories: hardware, system software, services and executables, files, IPC and more.

The following process describes how to place and execute a process on a specific resource. First of all, we need to describe resources, which we plan to do with a resource description language – RDL, WS-Resource [RDF2004, WSR2006]. The same description will be used for resource description as also for the resource discovery purpose. The matching is performed in two stages – first the resource discovery process finds resources, which static description matches the requirements, and then later on, the jScheduler through the jResourceMatching services retrieves the information about the dynamic resource properties, and schedules a task on a suitable resource. After the task is matched with the resource, the jController tries to reserve the resource through the rAllocation service, which is negotiating the service level agreement for resource usage. If the negotiation is successful, then the task is placed in a queue and is awaiting the execution.

3.1.4. Job Management APIs: XtreemOS calls

XtreemOS will extend traditional UNIX system calls with the concept of job. In this section we include a provisional list of XtreemOS calls. This list must be interpreted as an initial approach that will evolve and that will be coordinated with WP3.1.

We plan to provide POSIX compatibility related to process management in terms of creation (fork and clone), monitoring (getpid, getppid, wait, getuid, etc), control (kill(SIGKILL | SIGSTOP | SIGCONT)), etc, in two ways:

- In those cases where we propose a new XtreemOS call in the scope of processes, we will map the POSIX call in the new one, e.g. fork will be a default case of create Process.

- In those cases where we don't propose any specific function in the scope of processes, we will use the POSIX call, e.g. we will use the exit call instead of proposing a new exitProcess with the same functionality. However, we don't discard to extend the API proposed in this document if we detect new XtreemOS calls are needed.

Job Control

- `createJob (in jobDefinition, out jobID, out errorCode)`

  Create a job bound to user credentials. jobDefinition will include job description (executable, arguments, environment variables, standard

in/out/error), resource requirements (number of nodes|cpus, architecture, geographical limits, memory, …), resource hints (high cpu speed, high memory availability, …), scheduling hints (response time, economic), and job dependences.

Resource requirements will be interpreted by the scheduler as mandatory to execute the job and resource hints will be used to guide the resource selection. The default values for these parameters will be maximizing the amount of free resources per node and minimizing the response time of the job. If the job is free of dependences it will be scheduled based on its requirements and objectives and will be started as soon as possible. Otherwise, it will have to wait till all dependences have succeeded.

Additionally to the information provided by the user, additional parameters could be added to offer a generic way to automatically extend the functionality associated with the execution of a job. The initial proposal will consist in offering some kind of prolog/epilog approach. This functionality will come from other parts of the system such as the Storage management (WP3.4), Virtual Organization management (WP2.1), Security (WP3.5), or Infrastructure for Highly Available and Scalable Grid Services (WP3.2), e.g. to introduce replication mechanisms. Depending on the extensions details, they could be managed by the jExecMng (when started). [3].

- `createProcess(in resource, out processID, out errorCode)`

  Create one process in the context of the same job that calls the function in the specified resource. It is similar to a Linux fork but with a specific resource where the new process must be created (if `resource` is ANY, any of the available resources will be used). The resource must be one already allocated to the job. A `fork` system call could be translated as a `createProcess` where the resource would be the same where the `fork` is called.

- `jobControl (in jobID, in controlOperation, in user_cred, out errorCode)`

  Where controlOperation could be: WAITJOB, ENDJOB, SUSPENDJOB, RESUMEJOB, CANCELJOB, CHG_UID. In the case of changing those user credentials, we will include the new ones. Some of the functionality presented here could be also provided by sending an event (see `sendEvent in` under jobEvenet section). However, we have decided to duplicate it to follow traditional semantics in Grid systems.

- `exitJob(in exitCode)`

  It finalizes the execution of all processes of the same caller's job.

- `updateJobRequirements(in jobID, in reqOperation, in requirementList, in sync, out resourceList out errorCode)`

  The job requirements are dynamically changed. `reqOperation` could be: REQMORERESOURCES, RELEASERESOURCES, MIGRATIONHINTS. Jobs can

[3] The definition of extensions is even open so the list of services that could manage it is still open.

request additional resources if the job expects to have a good (or better) performance with more resources, in that case `requirementList` will contain the list of additional resources), can release specific resources, (in that case `requirementList` will contain the list of released resources), or can just provide hints for the jController in order to inform about resource characteristics that can improve the job execution if available. The `sync` parameter specifies if the reevaluation of the new resources must be synchronous or not. In the case of a synchronous call, the `resourceList` will contain the list of current resources allocated to this job.

- `attachProcess(in jobID, out errorCode)`

  This function is specifically designed by requirement of WP3.4 in order to convert a UNIX process into a grid job, or to attach a process to an already existing job. This idea must be widely discussed with members of WP3.4 to specify requirements and members of WP3.5 for security reasons.

Job Monitoring

Our main purpose for job monitoring is to be as flexible as possible and to provide as much information as possible. Information associated to a job can be static or dynamic. Static information could be the job definition submitted and dynamic information could be all the information related to the execution of the job. This dynamic information will depend on the status of job: a LOCALSUBMITTED job will be associated to a set of attributes such as number of queue, local scheduler, waiting time, and expected time to be executed; a job in RUNNING state will be associated to attributes such as list of processes, list of resources, user and system time. The main function in job monitoring will be `getJobInfo`. This function will be used to return any information requested. By defining the level of the information and the type of information desired (by means of the flag parameter), jobs could query any data associated to the job. This information will be returned as an XML format, giving the required flexibility.

- `getJobIDs(in jobFilter, out jobIDsList, out errorCode)`

  Returns the jobIDs matching the jobFilters passed as parameter, The jobFilter parameter will be a complex data that will consider SELF (jobID of the calling process), USER (jobIDs of all the jobs of the user), VO (jobIDs of all the jobs running in the VO), INSTATE (all the jobIDs in a certain state).

- `getJobInfo (in jobIDList, in flags, in infoLevel, out jobInfoList, out errorCode)`

  Return the available information for each job in the provided list. The job can select the level of detail it wants (HIGH, MEDIUM, and LOW). Job Information includes static values such as the authorization values or the submission time, and dynamic values such as the current state, and state specific values such as the list of PIDs or performance information (if RUNNING). With the flags parameter we can select different types of information we are interested in (STATUS, JOBDEFINITION, RESOURCES, PERFORMANCE, etc).

- `monitoringControl(in jobID, in operation, in level, out errorCode)`

  The operation parameter can be set to START, STOP, or CHANGELEVEL. In case of changing the monitoring level, the level parameter will specify it.

- `addAttribute(in jobID, in attributeName, in type, in value, out errorCode)`

  Add a new job defined attribute to the jobInfo data of the job identified by jobID. Values can be ABSOLUTE and RELATIVE which is indicated by type parameter. This function will allow adding attributes and values generated by runtime libraries and exported in a standard mechanism.

- `addCallback(in jobID, in metric, in callbackFunction, in callbackEvent, in value, out errorCode)`

  Define a callback function associated to an existing metric in the jobInfo. Callbacks will be associated to metrics with discrete values such as status, or metrics with continuous values such as memory usage or MFLOPS. The callbackEvent parameter describes when the job must be notified. The callbackEvent is a complex piece of data that, depending on the metric, could be CHANGE, EQUALTO, GREATERTHAN, LESSTHAN. In that cases the callbackFunction will be called when the metric changes, is equal, greater or less than the input provided value.

Job Events

Such as the UNIX signals, the events will have a default action associated to them but the job will be able to change them. These services are the equivalent set of services to UNIX signals but associated to jobs. For instance, it is very important to be able to detect the finalization of a job (some kind of SIGCHLD_JOB).

- `sendEvent(in jobID, in event, out errorCode)`

  Sends the event to the specified job, which means that all processes in the job will receive this event.

- `waitForEvent(in event, out, errorCode)`

  Blocks the calling process until it receives the specified event.

- `addEventCallback(in event, in ProcessID, in callbackFunction, out errorCode)`

  Adds a callback associated to a specific event. The ProcessID parameter indicates which process will manage the event (specific pid, ANY OR ALL). The callbackFunction parameter can be also set to DEFAULT OR IGNORE.

3.1.5. Job Management APIs: XtreemOS command line functionality

Additionally to these functions, this WP will provide to the Grid user with a set of specific command line functionality to facilitate the job execution management. These set of command line functionality will provide a similar look and feel and control than the traditional UNIX commands. Command lines commented here would offer the basic mechanism to submit, monitor, explicit migration, tracing, and sending events.

Job Control

- `Xsub -f jobdefinition`
  `Xsub executable [list of arguments] [<stdin] [>stdout] [>&stderr]`
  `Xsub –f jobdefinition executable [list of arguments] [<stdin] [>stdout] [>&stderr]`

  Creates a job and use the job definition, the traditional UNIX executable specification, or a mixed approach. In the second case (executable + arguments without `jobdefinition`) the resource requirements, resources hints, scheduling hints, and job dependences will be set by environment variables (We assume that most jobs will change the redirection or use other files as input and output, but the UNIX semantic should be maintained in case a job wants to use it.).

- `Xwait [–w ] jobID`

  Waits for the finalization of a job

- `Xmig [-nochk] jobID resourceList`

  Migrates the job to the specific set of resources listed. Optionally it is possible to specify if checkpointing is not required (default is yes).

Job Monitoring

- `Xps [–j jobID] [-a] [-d level] [-spra] [-v VO-credentials, [VO-credentials], […]]`

  It prints the information related to either the specified jobID or all the jobs from the user that executes the command (-a option). The –d option selects the level of information to be shown. The user can specify if he/she wants information related to status, performance, resource usage, and all. If the user has jobs submitted to V.O.'s different from the current one, he/she must provide credentials to access to them.

- `Xtrace [-j jobID] [-i sec] [-spra]-t traceFileName] [submission]`

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

Job Events

- `Xkill event jobID`

  Send the specified event to the job.

3.2. Application Execution Management services

3.2.1. Scheduling and Job management overview

Table 1 shows the list of services for job execution management. Users submit jobs to the Grid through the `createJob` call or `Xsub` command. These calls redirect the request for scheduling to the jScheduler service.

Service name	Scope	Duration	Main functionality	Relationships
jScheduler	Job	Scheduling cycle	Decide the best resource selection for the job	JobDirectory, LTScheduler, jController, jResourceMatching, rAllocation
LTScheduler	Local system	Permanent	Control local resources and manages list of locally queued jobs	jScheduler, jController, jExecMng
jController	Job	Job life	Control scheduling agreements during the whole life of the job and act like a gateway for one job. It could take the decision to start a migration	jScheduler, jExecMng, LTScheduler, jMonitor, rMonitor, jEvent
jExecMng	Job	Job Exec.	Manage the efficient utilization of resources allocated to one job. Reacts to some jController decisions.	LTScheduler, jController,rAllocation
jMonitor	Job	Job Exec.	Monitor status, performance and resource consumption of one job	jController, jExecMng
jEvent	Job	Job Exec.	Manage events addressed to one job	jController, jExecMng
JobDirectory	V.O.	Permanent	Manage list of jobIDs and contact information	jScheduler
rMonitor	Resource	Permanent	Controls resource dynamic information	jController
jResourceMatching	Job	Scheduling cycle	Receives a list of resources and job requirements and returns the resources that match requirements.	jScheduler
rAllocation	Resource	Permanent	Receive requests to allocate resources to a job.	jScheduler,jExecMng

Table 1: List of XtreemOS services

The jScheduler service receives one job definition, which includes a list of resource requirements, selects a list of sufficient resources that match with job requirements, and performs the local submissions to the selected resources. Resources are selected based on job requirements (and filtered using the jResourceMatching service) and a negotiation with local

systems through the LTSchedulers services. In this process, jScheduler does not try to make the best solution for the global system, but best possible submission for the new job given the available resources.

Depending on the quality of the scheduling requested by the user, the jScheduler can negotiate an advanced reservation of resources or just use LTScheduler information for estimating which local systems will provide the best scheduling service. In case the user asks for an advanced reservation, the resource allocation is done by the jScheduler (through the rAllocation service), and therefore accounted at the submission time. Otherwise, it will be allocated when the job starts the execution. If the jScheduler doesn't make an advanced reservation of resources, the negotiation with LTScheduler will not conclude with an *agreement*. In that case, the execution of the job could be different from the expected by the jScheduler when it scheduled the job. Because of that, the jController service is created as the responsible for validating the job is going well, and otherwise to decide to re-schedule[4] it (to try to migrate it to *better* resources). We will refer to the first case like *agreement* (when using advanced resource reservations) and to the second case like *relaxed agreement*. In any case, the jScheduler starts at job submission and ends when the job is dispatched to local systems.

The LTScheduler service offers a common interface to jSchedulers, hiding characteristics of local systems. From the point of view of the jScheduler, it offers the management of local resources and jobs submitted to them. Even when the local system includes a specific queuing system, the LTScheduler will extend it to provide the matching between grid id's and local id's and to translate available information and control API to the API for negotiation with jScheduler. Our idea is to use as much as possible existing works concerning SLA agreements.

The jController, jExecMng, jMonitor and jEvent are per job services. The jController is the only one that runs during the whole life of the job: from GRIDSUBMITTED to FINISHED. It knows about job definition, resource requirements, and job performance metrics[5]. The jController sees the job as a whole. It understands about metrics and algorithms to take decisions about the job such as job migrations. Job migrations could be started, for instance, if the jController detects the job is not going as well as expected and the rMonitor notifies that a new (and *better*[6]) resource is available in the V.O. Additionally, it acts like a gateway for the job, avoiding having multiple contact addresses.

The jExecMng, jMonitor and jEvent are only active when the job is running in a set of resources. If the job is migrated, the existing services will be killed and new ones will be created associated to the new set of resources.

The jExecMng service is in charge of fine grain details of the job execution. The jExecMng sees the job as a set of processes that collaborate. The jExecMng knows about different types of jobs, e.g. MPI , OpenMP, MPI+OpenMP in the case of parallel jobs, and is able to take internal decisions to improve the execution of the job. Moreover, it implements decisions taken by the jController.

---

[4] The jController can decide to re-schedule the job but the jScheduler implements the scheduling algorithm.

[5] Performance metrics will e different depending on the type of job. We plan to work in ways to define and to measure these metrics based on types of jobs.

[6] One resource will be *better* than other depending on the job, see section 3.2.4 for more details.

jMonitor service collects the information associated with the job execution. Before execution, the information of the job is maintained by the jController. Once the job starts, the jMonitor collects information about status, resources used by the job and performance. It offers callbacks notifications and the possibility of adding dynamically new metics to the existing basic schema. The jEvent distributes job events.

3.2.2. Scalability and fault-tolerance issues

The list of services proposed reduces centralized information to a minimum; avoiding the amount of information that could be potentially lost if a node fails. There are two services that must be fault tolerant: JobDirectory (one per V.O.) and jController (one per job).

The JobDirectory maintains the list of existing jobs with the corresponding addresses of jController services. If the JobDirectory fails and the information can not be recovered, the contact addresses of all the jobs in the V.O. will be lost. This service will be used to implement the xps command, or to find the jController address given the jobID.

The jController service provides access to one job. If one jController fails and we cannot recover it, the contact point of its job will be lost. In that case, the job could continue running, but its behavior will be undetermined.

Concerning the scalability of these services, since most of them are in the scope of the job, we expect to have reduced at maximum this kind of problems. However, we will pay special attention to the architecture of jExecMng and jMonitor, which could potentially have to control hundreds of nodes.

3.2.3. jScheduler[7]

Figure 2 shows the main API and functionalities of jScheduler service. The jScheduler service receives a list of resource requirements (mandatory) and hints (objectives), and a list of scheduling hints (for instance “minimize the wait time”, “minimize the response time”, “minimize the cost”, etc. The real list of available hints is still a research issue) for one job, and decides the resource allocation for this job.

jScheduler uses the ResourceDiscovery and jResourceMatching services in order to obtain a list of resources that match the job resource requirements. The jScheduler will negotiate with LTSchedulers that control jobs submitted to these resources. Depending on user requirements, we consider two types of negotiations: agreements and relaxed agreements. In the first case, local resources (through the LTScheduler service) must provide advanced reservation of resources (or similar mechanism) for ensuring the agreement will be fulfilled in any case. In the second case, the local resource (through the LTScheduler service) only provides performance metrics, e.g. average wait time or number of queued jobs, about the local scheduling that can be used by the jScheduler to estimate when the job will be started if it is submitted to this local resource. The jScheduler selects one or more resources to submit the job based on this information.

For advanced versions of the service, we have also considered the possibility of receiving information from the File Manager related to the cost of accessing files involved in the job execution[8]. More details about interaction with the rest of services are given in section. 3.3

---

[7] Services starting with a "j" mean that they are services that apply to a single job and try to make the best for this job, regardless of the global benefit.

We also plan to include an additional functionality to the jScheduler that consists of the possibility to *add* more resources to the job or to change the list of resources. In the first case, the jScheduler will also receive the list of currently allocated resources and will try to add resources closest to these ones. This is the case of an update of requirements. The second case corresponds to a job migration. In that case, the jScheduler receives the current allocation for *affinity* reasons. By affinity we mean that the jScheduler service will try to allocate new resources as close as possible to the old ones assuming that minimizing the distance will improve files access.

jScheduler will also include the required functionality for submitting the job to the selected resources.

The AEM will not include any global scheduler for the Grid. Load balancing will be provided by the negotiation included in jScheduler policies and validated by the jController service. If the jController detects the job has been submitted to a high loaded resource (because of the asynchronous scheduling on multiple jobs), it could decide to migrate the job.

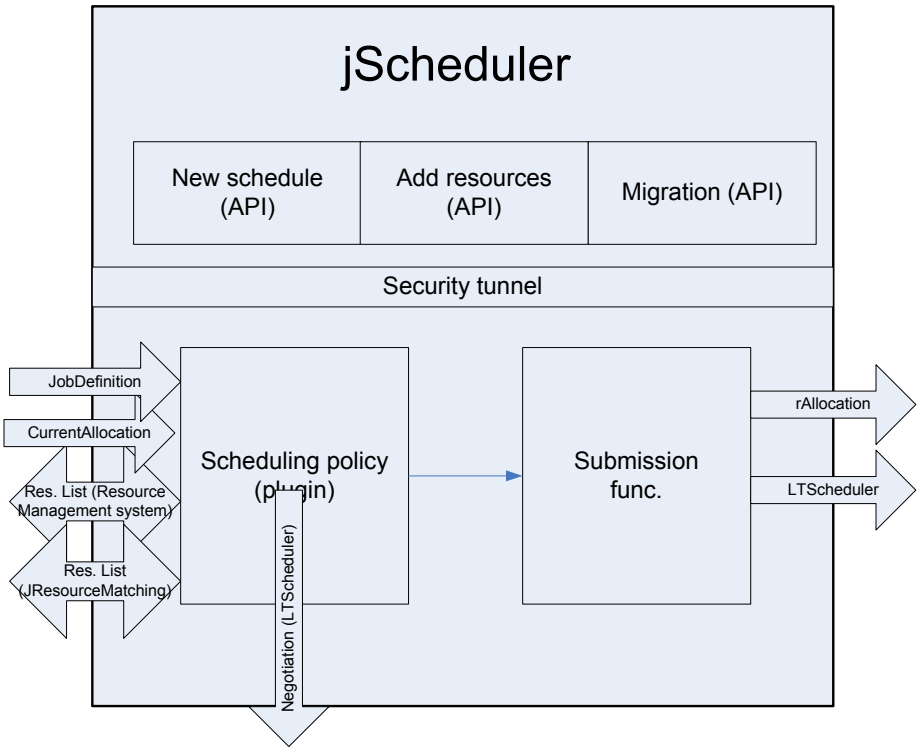

Figure 2: jScheduler service

3.2.4. jController

The jController service holds the job information and its main goals are first, to control that the agreement between the jScheduler and LTSchedulers is accomplished, second, to validate the job is executing as expected when it is RUNNING, and third, to act as a gateway for the job. The jController envisions the job as a whole and it is the only service associated with the job during its whole life (from GRIDSUBMITTED to FINISHED).

[8] Those files known in advance such as the executable, stdin, stdout, etc or specifically defined by the user in the job definition.

It offers an API for job control operations and also automatically reacts when the execution of the job is unexpectedly delayed or when the execution doesn't reach the expected performance. This may happen in the case of resource failures or relaxed agreements.

The jController acts like a gateway for one job, see Figure 3. It must be **fault tolerant** and will be implemented using advanced services developed in the WP3.2. The jController centralizes all events related to the job scheduling (or re-scheduling). Re-scheduling can be initiated by the user or automatically started by the jController. A potential scenario that could generate an automatic re-scheduling (or migration) is, for instance, better resource availability (based on scheduling hints) and bad performance of resources used (i.e. not meeting the specified MFLOPS). It is important to notice that re-scheduling will only be an option if the job is not meeting the agreement. If better resources appear, but the job is meting its agreement, no migration will be evaluated. We plan to work in schemas and mechanisms to define application performance metrics and specific monitoring plug-ins for jobs and resources that allows the jController to decide whether a job is performing well or not and whether a resource is better than other for a specific job. We understand this is one of the challenges of XtreemOS concerning job execution management.

Due to the dynamicity of the Grid, the resource availability changes through the time, and jController service has to cope with this changes. When jScheduler is searching for appropriate resources to place a job, it distributes the resource requirements. In the nodes, where the dynamic properties restrict the match, the description is stored, and in that case that at some later stage the resource meets the dynamic requirements, a notification is sent through the jEvent system in order to notify jController that some other resources have become available for task (process) placement.

The jController includes two internal services for managing the special case of job migrations: the jMigration service and the jCheckpointing service.

jController::jMigration

Dynamicity and heterogeneity are two of the characteristics of grids. These two characteristics can happen because of resource failures, appearance of new (better) resources, or changes in resource characteristics. In some of these cases, the jMigration started by the jController could decide to migrate the job. Job migration is a hard task because it includes checkpointing and moving process state from a set of nodes to another. The jMigration service will take care of all of these issues by means of the jScheduler service (to decide on the new scheduling) and jCheckpointing service (to checkpoint the job if required).

jController::jCheckpointing

The jCheckpointing is the service responsible for supervision of checkpoints for a job: it applies the checkpointing strategy to all running processes.

- It registers the processes with the checkpointing service on the nodes running the job's processes.
- It provides resources to store the checkpoints.
- It is able to launch jobs in a checkpoint context.
- It coordinates the checkpoint of a job running on different nodes.

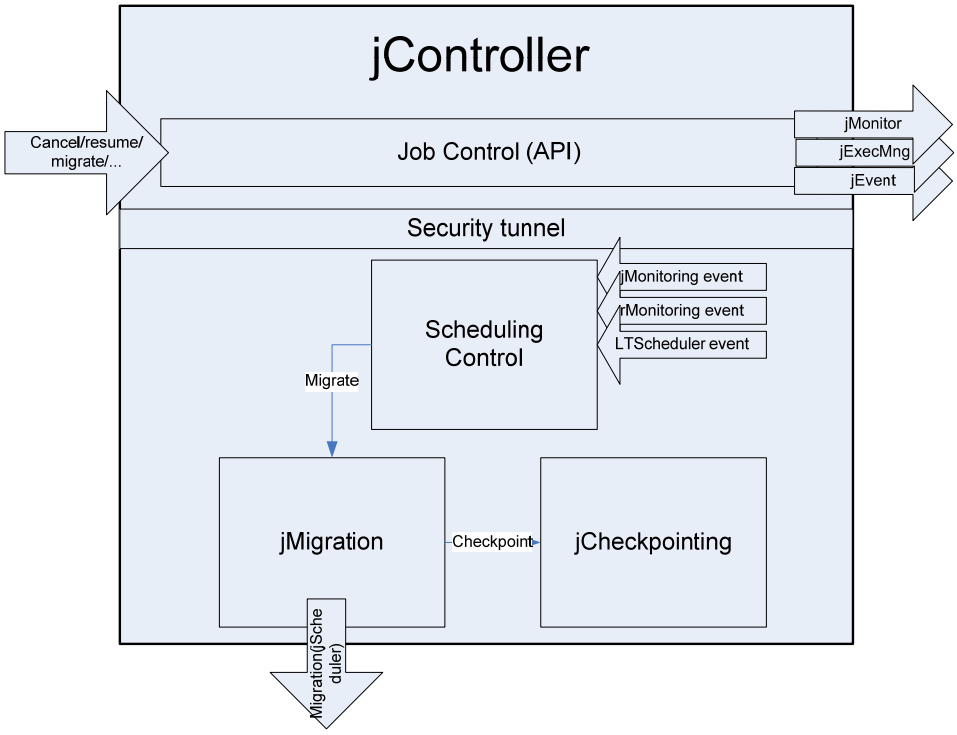

Figure 3: jController service

3.2.5. LTScheduler (Long-Term Scheduler)[9]

The LTScheduler service is the traditional "long-term scheduler" or queuing system, see Figure 4. Each instance of the LTScheduler service will manage one *exported computing resource.* LTScheduler offers a complete API in terms of resource negotiation, job submission, monitoring, and control. An *exported computing resource* can be a PC, a cluster or federation. If the local system includes its own queuing system such as MAUI [Maui], the role of the LTScheduler will be to provide a common API from the point of view of the rest of the execution framework. If the local system doesn't provide it, the LTScheduler will offer the API and the functionality. In any case, from the point of view of the rest of XtreemOS services, the LTScheduler will be in charge of scheduling jobs submitted locally.

The LTScheduler manages a list of submitted jobs and schedules jobs based on per computing resource policies. These policies affect the negotiation with the jScheduler, i.e. if the negotiation includes advanced reservation of resources, this reservation will be done according to the existing policy; if the negotiation doesn't include resource reservation (i.e. relaxed agreements), and it is only based on performance metrics such as average wait time or expected start time, these metrics will be calculated or estimated based on the scheduling policy. However, considering that the LTScheduler is system-oriented and not job-oriented, it is possible that it breaks some of the relaxed agreements with jScheduler for the global interest[10]. In that case, it will notify the corresponding jController in order to start re-scheduling actions if required.

[9] Services that do not start with a small letter (j or r) are services with a view wider than a single job and try to optimize the benefit of a group of jobs (or resources) regardless of the individual performance of each one.

[10] In that case, advance reservations of resources are not used.

Negotiation with jSchedulers and notifications in case of failures in agreements are the two main challenges in XtreemOS in terms of local scheduling.

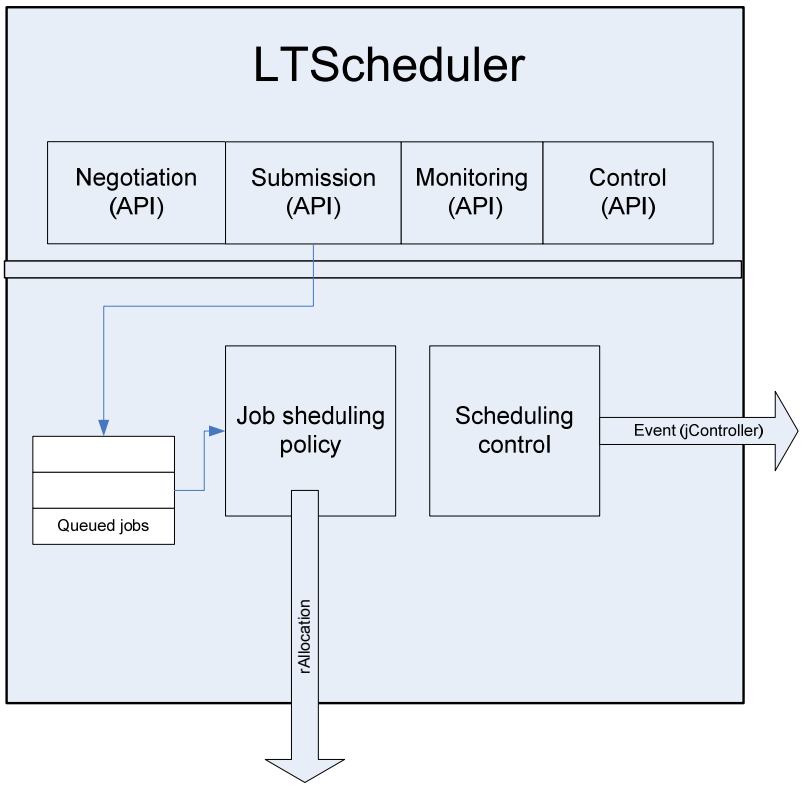

Figure 4: LTScheduler

3.2.6. jExecMng

The jExecMng service is in charge of the *efficient* utilization of resources allocated to one job. It sees the job as a set of individual components (processes and set of processes) that collaborate. We understand the term efficiency depends on the type of job. Because of that, the jExecMng will be based on extensible API's and plug-in mechanisms to calculate the value of the *efficiency* depending on the type of job. In this WP, we plan to work in schemas and mechamisms to **define efficiency metrics per type of job**, **algorithms for evaluation** of the efficiency, and list of **required actions** based on this evaluation. The calculation of the efficiency will be done based on basic metrics calculated by the jMonitor, which will have as first design requirements the extensibility to include application specific metrics.

The jExecMng service is a distributed service (in case of co-allocated jobs) running in all nodes where the job is running. It coordinates job control operations if the job is co-allocated and takes care of the efficient execution of the job. The jExecMng is in charge of starting the job, creating additional services such as the jMonitor and the jEvent. It will also take care of the automatic execution of functionality coming from other WP's, providing some kind of extension of system functionality.

For instance, in the case of parallel jobs (co-allocated or no), the jExecMng could detect that two processes communicate a lot and therefore to place them in two nodes with high bandwidth. Other example could be to measure floating point instructions per process and to migrate high loaded processes to the more powerful processors from the set of available for the job. We plan to include this kind of features in advanced versions of this service.

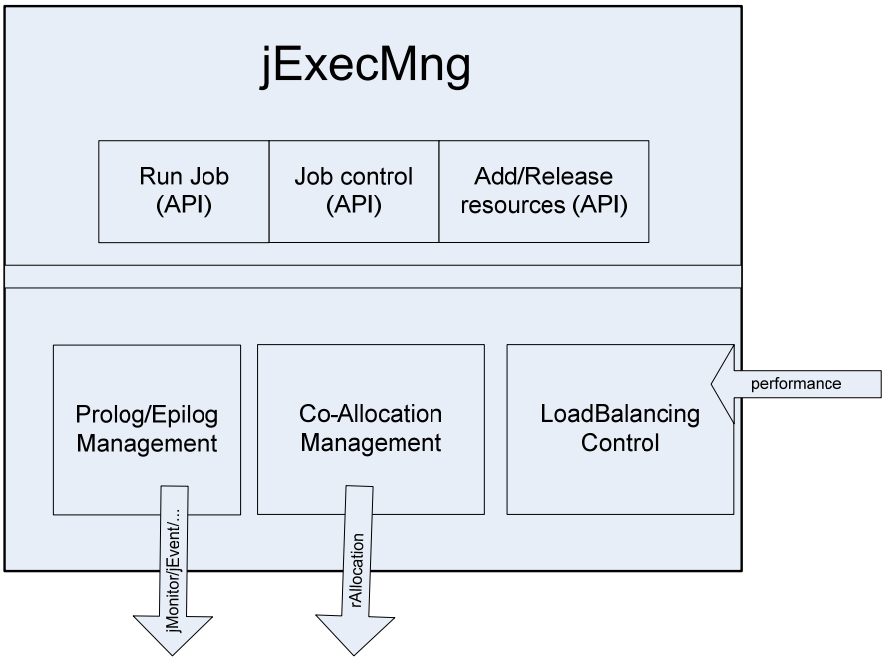

Figure 5: jExecMng service

3.2.7. jMonitor

The jMonitor service is a distributed service (in case of co-allocated jobs) running on all nodes where the job is running. It is in charge of collecting all the information related to the job execution. This information will be exported following some XML template. The idea is not to limit the amount of information that can be passed to the job assuming part of the information could be provided by runtime libraries not pre-defined. jMonitor will offer an API for adding metrics and attributes to the job information exported. In this way, we can dynamically improve the semantics and the amount of the information collected.

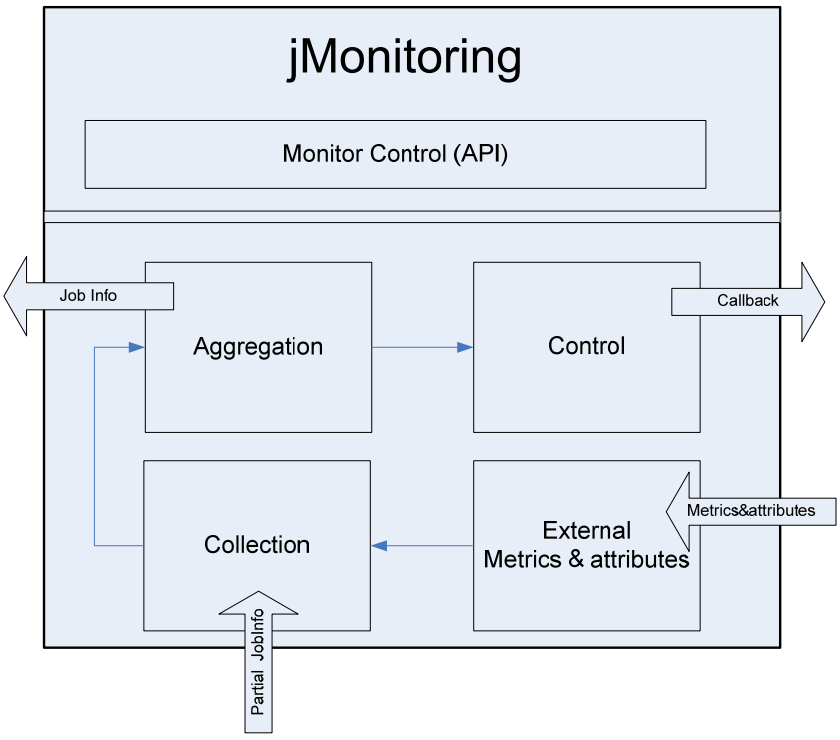

Figure 6: jMonitoring

3.2.8. jEvent

We need some mechanism to distribute job events. This service will receive events from different producers (rMonitor, jExecMng, XtreemOS calls) and will redirect it to consumers: jController, processes or the job, etc. It will manage both job-level events and system-level control events. However, it is not clear it we need any additional service or not because we plan to use the Publish/Subscribe service that will be provided by WP3.2. Depending on the available features, this service could be substituted by a *publisher* in the part of the job and *subscribers* will be directly the jController, jExecMng, and the jMonitor.

3.2.9. JobDirectory

The JobDirectory service will take care of storing and managing the information about the list of active jobs per user (associated with their credentials) and the required information to find the jController service associated to each running job[11]. This service is very important and it must be **fault tolerant**. In the same way that the jController, we plan to implement it on top of the advanced services developed in WP3.2. JobDirectory service manages a data base with all the relationship between jobIDs, user credentials and jController addresses.

The architecture of the JobDirectory service is the most important one because it is a potential bottleneck in the job execution management system.

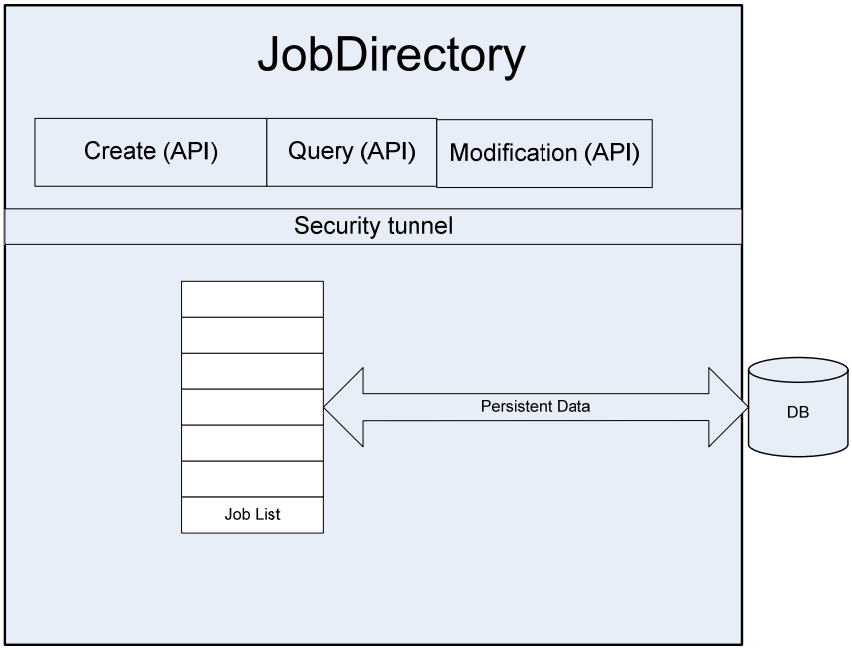

Figure 7: JobDirectory

[11] Remember the jController acts like a gateway for the job.

3.2.10. rAllocation[12]

The role of this service is to receive requests to allocate resources to a process and to execute it under conditions that have been negotiated. The most important aspect of the service is to maintain service level agreements – SLAs even in case of failure or unexpected circumstances. The rAllocation service will expose two important APIs. First API is to allocate a time slot in which the task can use the specified resource. Second API provides access to the information about the task or resource state, which is important for other higher level services.

Advanced versions of the service will provide possibilities to re-negotiate the allocated resources through two different protocols. The first protocol will try to allocate additional resources to keep the SLA intact. The second version will start SLA re-negotiation with the service submitting the job, if the service in question supports such API.

Both allocation and query API are accessed by jScheduler or jExecMng (still to be decided). One of the services will be responsible to submit a task to this service, but it is not yet decided, whether this task can be executed by jExecMng or jScheduler. LTScheduler is responsible for accepting and keeping track of the tasks (process) queue. The reservation of the resources will be implementing DRMAA interface (**¡Error! No se encuentra el origen de la referencia.**). The **resource allocation policy** (the limits for each resource type) is responsibility of XtreemOS-F services. As an example, each memory allocation call performed on behalf of a process would have to check the amount of memory currently allocated to that process and fail if the newly requested amount would cause the limit to be exceeded.

The rAllocation service will keep a local copy of the information related to the task, in the figure denoted as “Task struct” in order to link the task with the job to which it belongs, which will allow sending notification events to jController. In addition to JobID information it also stores the resource hints and other task related information.

[12] Services starting with an “r” mean that they are services that apply to a single resource and try to make the best for this resource, regardless of the global benefit.

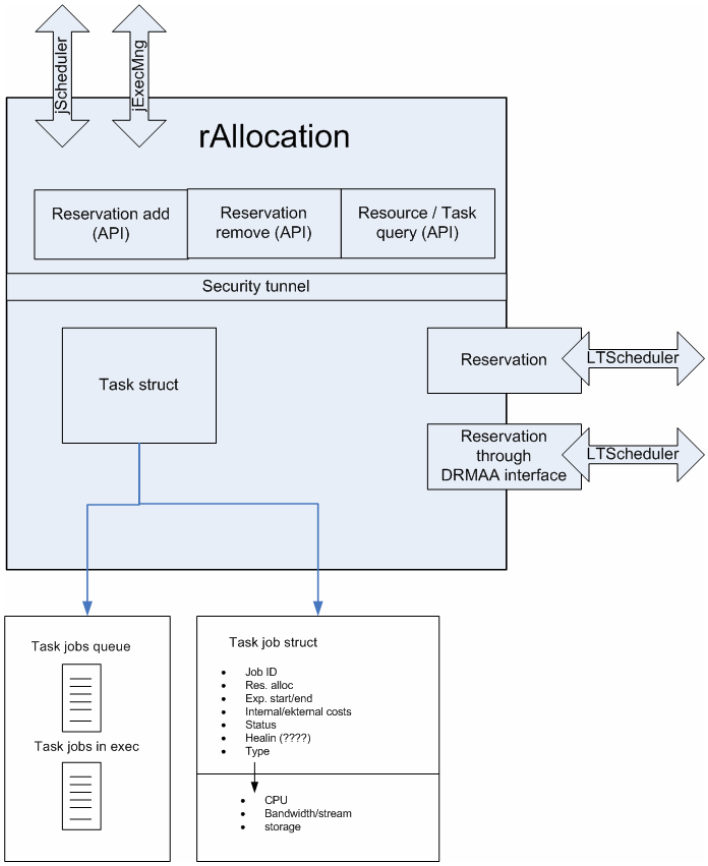

Figure 8: rAllocation service

3.2.11. rMonitor

The rMonitor service functions as a resource monitor, and is the lowest service, closest to the resource and task (process). Most of the typical information on resources are available in the Linux kernel: CPU(s) information, virtual memory and available network bandwidth is gathered by the kernel and exposed via the /proc pseudo-filesystem. Therefore in order to gather information this service must implement mechanisms which will allow to be extended to monitor different resource or task (process) parameters. This is achieved by two layered structure, where the first layer – probe periodically gathers information from the system, and submits it to the analyzer. The Analyzer decides if there is a need to send an event to the higher level services, like load balancing scheduler or jController. The rMonitor service provides two APIs. First API provides a publish / subscribe access to the events generated by rMonitor, and the second API provides to jResourceMatching service dynamic information on the resource state for the resource discovery and matching purposes.

The advanced version of the rMonitor service will include adaptive algorithms enabling the event triggering to be dynamically adjusted based on the information provided from the subscribing services. In addition the rMonitor could also trigger an event, which would send a request for a task (process) reallocation to the rAllocation service.

The first interface of rMonitor service is accessed by the jController, which is the only service alive the hole job lifespan. The second interface providing information to the jResourceMatching service, which is responsible for matching resources with resource request based on the dynamic resource information. The last interaction foreseen for the rMonitor service is the interface with the load balancing services, which will initiate job migration when a node gets over a certain threshold load.

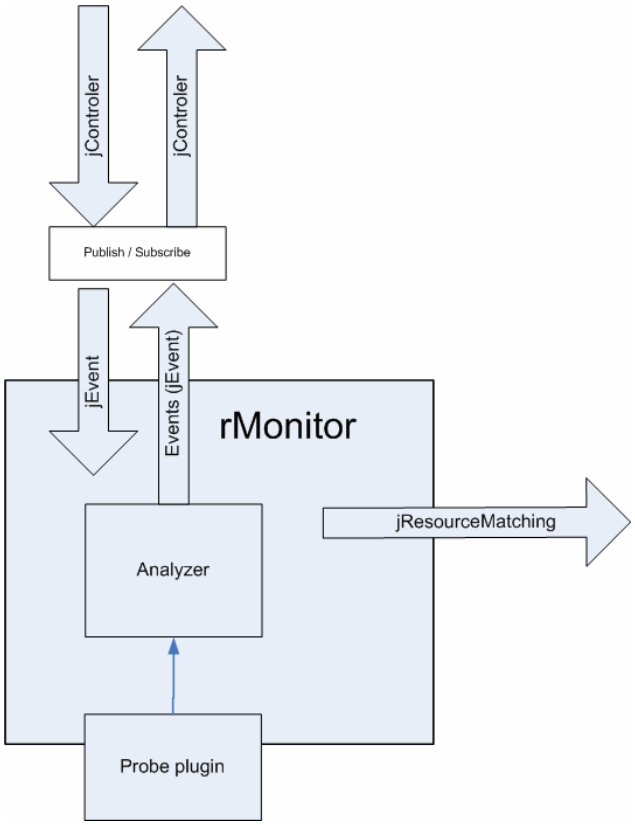

Figure 9: Monitor Service

3.2.12. jResourceMatching

This service collects the resource state information from the local rMonitor service, and stores it in local cache. The information in the cache is updated from the rMonitor service by means of events – push method, where we believe that this will considerably reduce traffic compared to periodic updates. The service provides access to so gathered information to jControler in order to support jScheduler scheduling and re-scheduling decisions. The information can be retrieved through querying in the case of initial scheduling of a job, or pushed through a jEvent service when the resource become available to accept some task (processes) to execution.

The resources have to be described in a common data format. As third-party extensibility is our goal, we propose to define an extensible XML schema – RDDL and RDF as a container for information on various resource types, and a separate schema for describing each resource type.

An advanced version of the service will expose the resource information available in the local cluster through a resource discovery protocol, which will be used to aggregate information about available resources and their dynamic state in order to reduce network traffic and improve possibilities to discover resources.

jResourceMatching service in relation to resource management system provides dynamic information about the resource state, where on other hand the function of the resource management system is to find resources based on their static requirements.

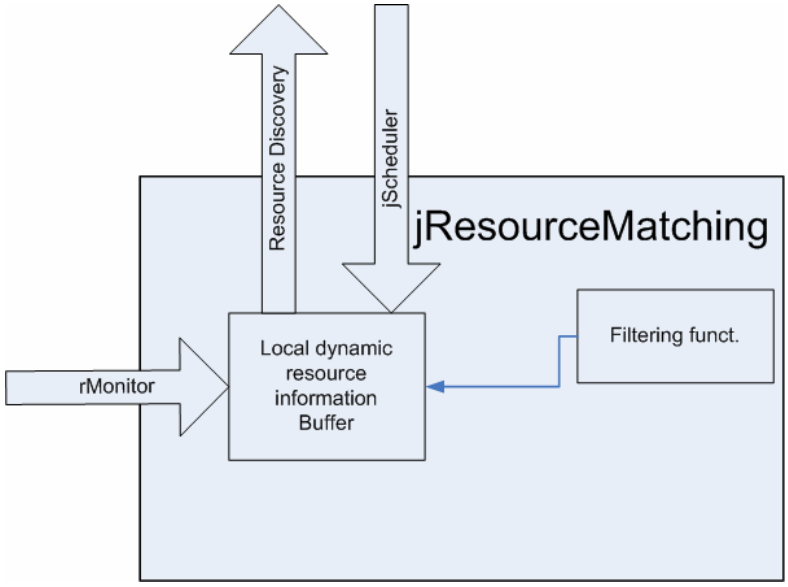

Figure 10: Resource matching service

3.3. Relationship between services

3.3.1. Job submission

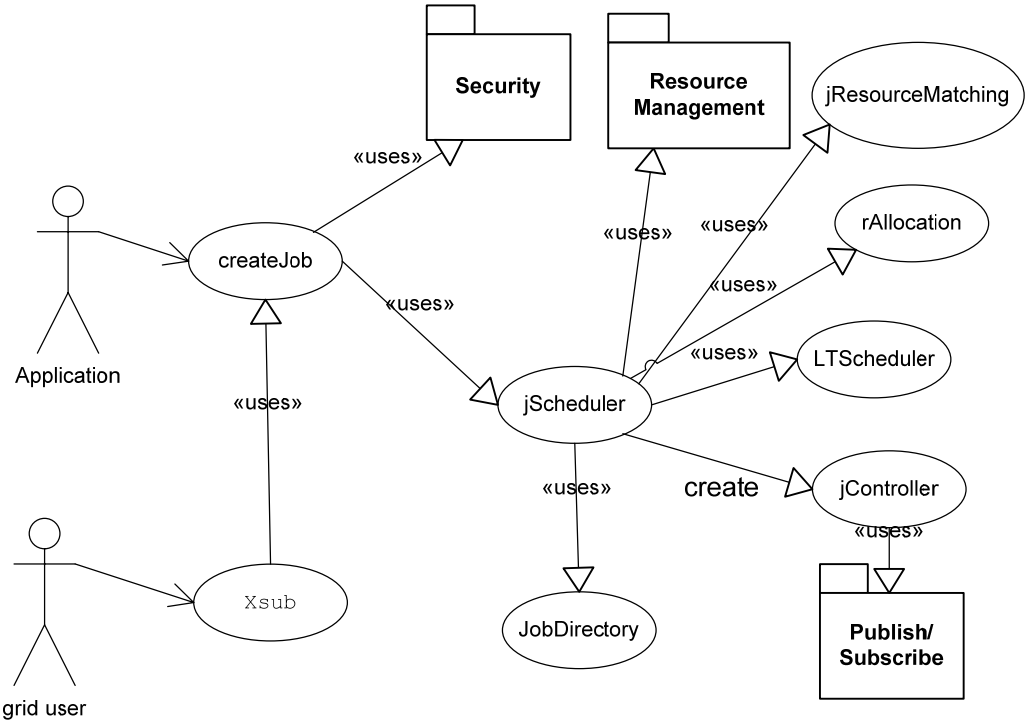

Figure 11: Create Job

Figure 11 shows the case of creating and submitting a job. Jobs are created using the `Xsub` command or the `createJob` call. The `createJob` function uses the required security services in order to validate the user identity. The jScheduler uses the Resoure Management System to get the initial list of resources that match with the static requirements. The jResourceMatching service receives a list of resource requirements (provided in the job

definition), the list of resources provided by the Resource Management system and it returns a filtered list of resources.

Once filtered, the list of resources and the job definition are passed to the jScheduler, which decides on the final resource selection. The scheduling policy will consider job resource hints, job scheduling hints, and the list of resources. To consider scheduling hints, we plan to include a negotiation with LTSchedulers services that manage each resource (assuming a resource could range from a supercomputer to a single PC). We expect from these LTScheduler services as detailed hints as possible regarding metrics such as expected waiting time, number of jobs queued, average wait time, "economic" parameters such as cost of AU (Allocation Units)[13].

Once the best set of resources has been selected, the jScheduler executes the submission functionality required to deliver the job to the LTScheduler service (or services if the job is co-allocated).

The jScheduler also creates a jController service that will act as a proxy for many of the actions applied to the job. The contact point of the jController is then passed to the JobDirectory, which is the service that provides and manages the list of jobID's.

We have considered that the jScheduler could potentially receive information from the File system related to "best location" to schedule the job based on files involved in job execution. This interaction is not yet decided but we think about issues such as "latency and bandwidth when accessing to file X from node Y". If some (of all) of the files that are involved in the execution of the job are known (either specified by the user or using some historic information), this information can be considered as input to the scheduler. We have not included this relationship in the figure for simplicity.

In advanced versions of services we plan to coordinate all the LTSchedulers that take part in the co-allocation of a job distributed on several nodes.

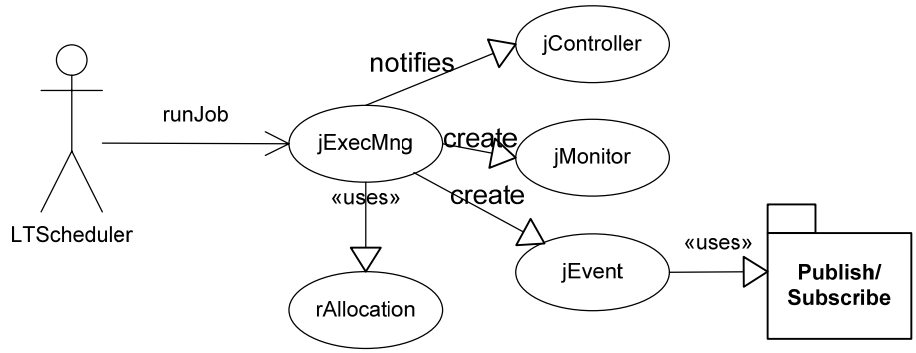

Figure 12: Run Job

Figure 12 shows some of the reactions in the system when one LTScheduler selects a job for execution (this behavior will be replicated in all the resources that participate in the execution). The LTScheduler will start its part of the job and it will create the jExecMng. The jExecMng creates the jMonitor, and the jEvent services to control details concerning the execution and monitoring of the job.

[13] AU is a metric used in European projects that could be used to calculate the cost of executing a job on a specific system

For basic services, the functionality of LTScheduler will just provide services offered by underlying queuing systems. In the advanced version we plan to include specific mechanism for advanced reservations and co-allocation between nodes.

3.3.2. Job Control operations

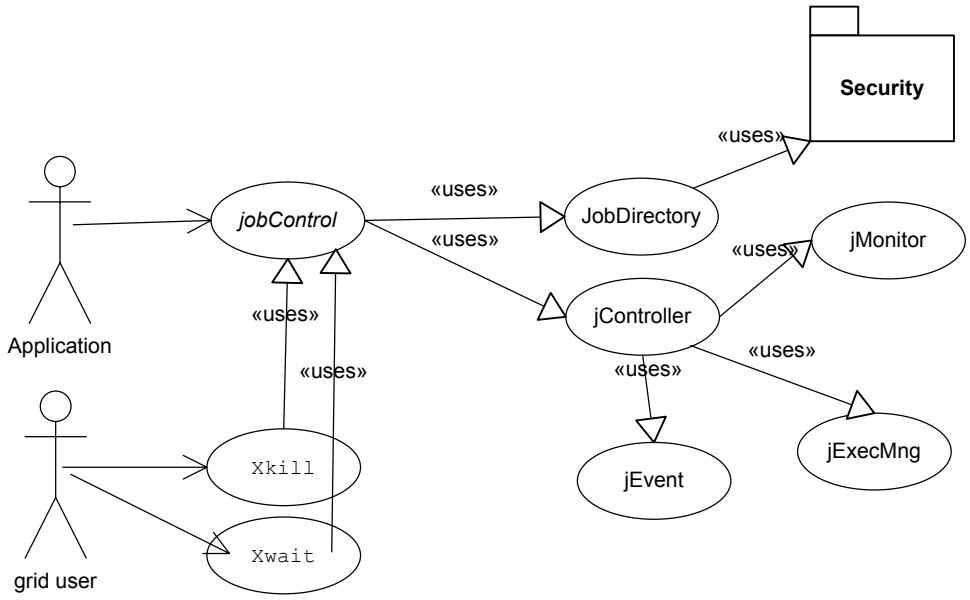

Figure 13: Job Control operation

Figure 13 shows the case where an job or a grid user wants to apply some control operation to an existing job (not necessarily RUNNING). The first step is always validating user credentials. Since the jobID is not yet known, it is possible that this validation requires some special query to some Security services. Depending on the jobID it could be just some internal validation. Once the user credentials are validated, the `jobControl` implementation will contact the jController service that depending on the required action will contact the jExecMng or the jMonitor.

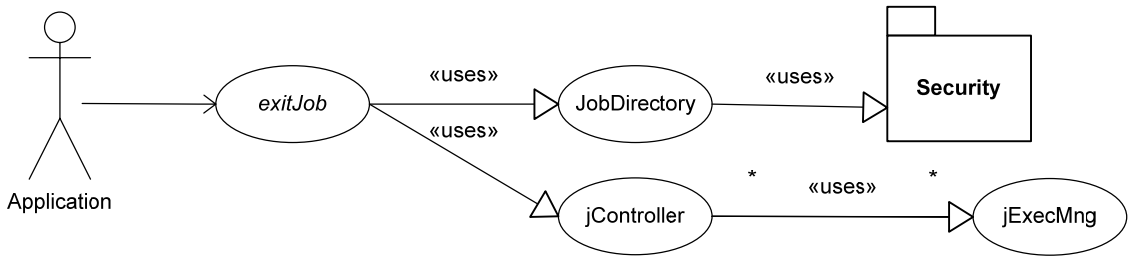

Figure 14: exitJob

The `exitJob` call is called by one process in the job and it will finish the execution of the whole job. It is the same behavior than an exit called by one thread in the context of a process. The jController will *propagate* the order to the rest of processes by means of the jExecMng.

3.3.3. Job Monitoring

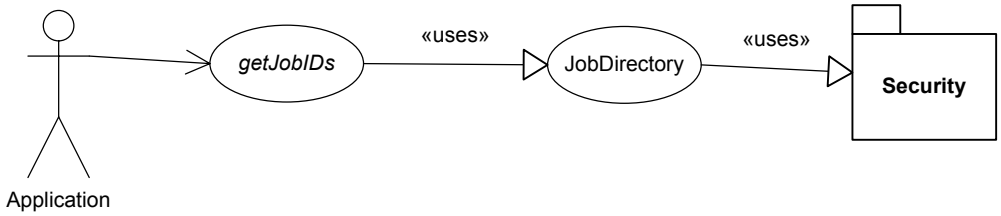

Figure 15: Monitoring use cases (I)

As we have introduced before, one of the challenges of XtreemOS is providing a good job monitoring system. Our goal is to be able to offer as much information as possible and let the user or the job decide about the level of granularity it wants. Figure 15 shows `getJobIDs`, which returns a list of jobIDs that fits in a list of filters. Figure 16 shows the relationship between services that manage job monitoring: `getJobInfo`, `monitoringControl`, and `addCallback`.

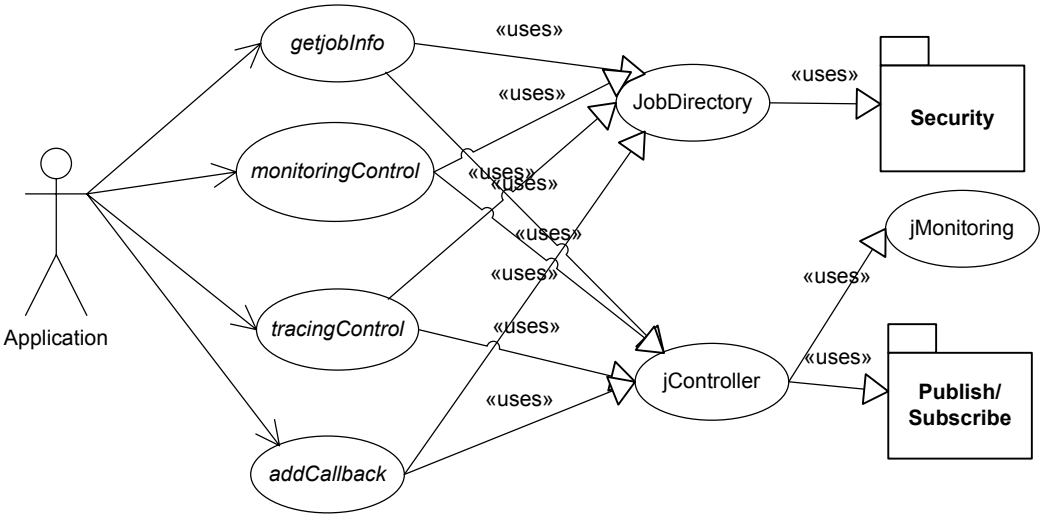

Figure 16: Monitoring use cases (II)

Figure 17 shows the behavior of commands for monitoring and tracing. We understand job monitoring is something that will be managed by the job itself, by some workflow manager tool, or by the user itself to see how his/her job performs. Tracing is just a continuous recording of this information in a specific format. For this reason, we will manage the collection of traces by the user command line `Xtrace`.

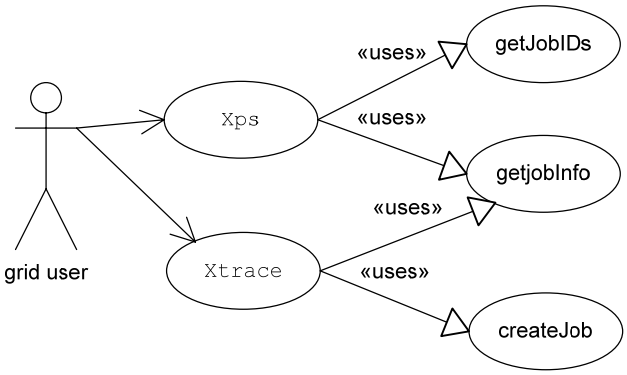

Figure 17: Monitoring use cases (III)

3.3.4. Requirements update and Job migration

Figure 18 and Figure 19 show the more complicated situations to manage: the case of dynamic request of requirements and the case of job migrations. By *dynamic request of requirements* we mean those cases where jobs ask for additional resources or release some of the current allocated resources, but the main part of the job remains in the same set of resources. We assume this is going to be a use case for advanced jobs (probably workflow managers or runtime libraries for grid management). For this reason, we don't plan to include this advanced functionality as basic services. By *job migration* we understand the case where the set of resources allocated to the job is totally (or mostly) changed. Some of these services can need jCheckpointing services to save the context of processes (or jobs units) of the job and restart them in a different set of resources. Figure 18 shows the case where resource requirements change or job migration is started by the job (or the user).

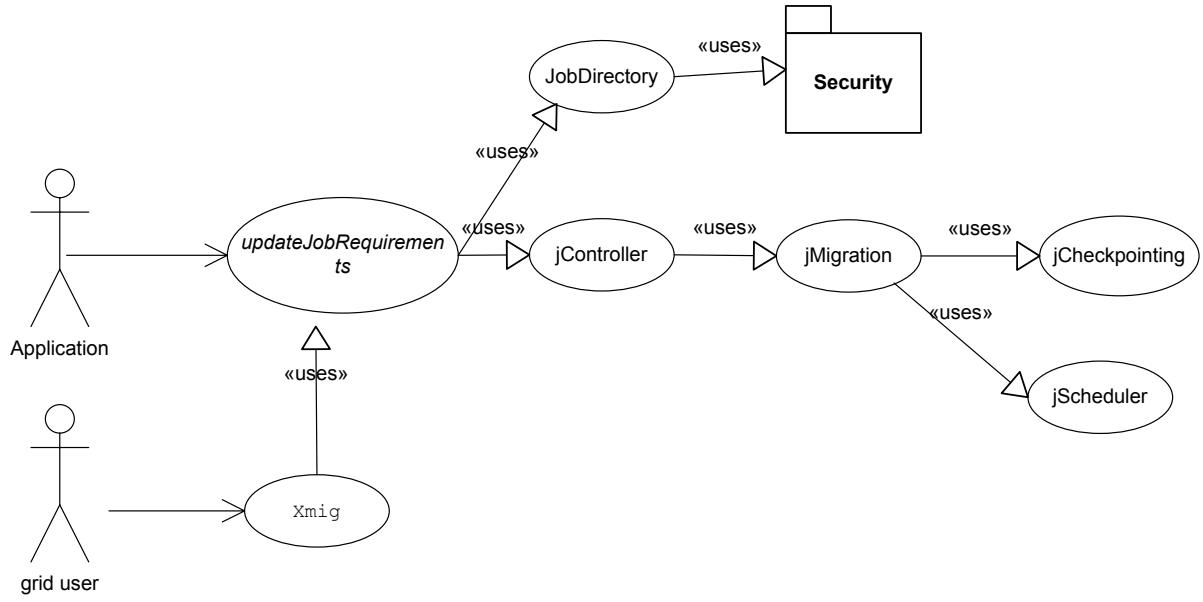

Figure 18: User requested changes in requirements

Figure 19 shows the case where the job migration service is required as a result of an event coming from an internal service: jMonitoring, LTScheduler, or rMonitor. We plan to control cases where some of the resources requested by the job in the job definition (which was

mandatory) fail. We plan to include this functionality as a basic service because we think it is important to manage the case of resource failures.

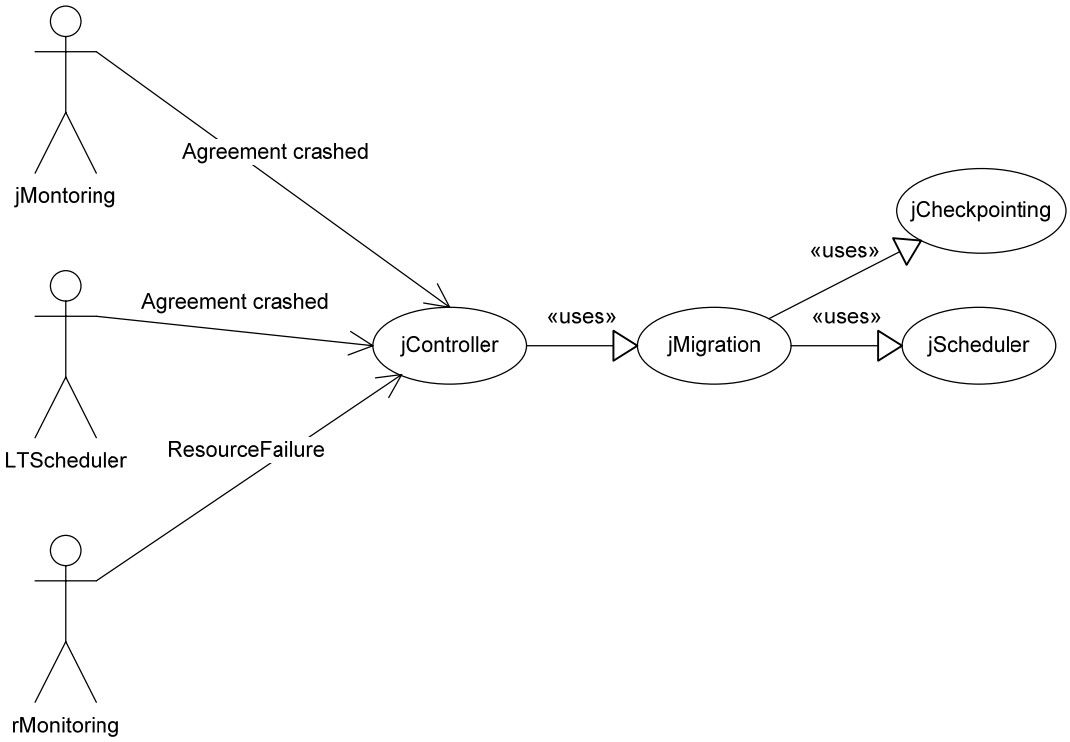

Figure 19: Automatic migration

3.4. Inter-WP dependencies

Regarding the Application Execution Management we have identified following inter-WP dependencies:

- **WP2.1:** definition and use of VO functionality is important to determine the set of resources which VO is authorized to access to execute its jobs/tasks. In addition jMigration and jCheckpointing have to be able to access the Migration and Checkpoint/Restart functionality, which is implemented in this WP for a single Linux machine.
- **WP2.2:** defines the federation level scheduling architecture, which will have to provide interfaces to Resource Discovery protocols initiated by jAllocation, Resource Matching protocols initiated by jScheduler. In addition jMigration and jCheckpointing have to be able to access the Migration and Checkpoint/Restart functionality, which is implemented in this WP for single LinuxSSI systems
- **WP3.2:** Promises to provide a stable service access point, which could be used to implement jController. In addition this workpackage is developing protocols for highly available services, which will be used for Resource Discovery to generate a set of possible matches for a specific resource matching request.
  Resource Discovery is an important service devised inside WP3.2, which is used by two services; jScheduler and jResourceMatching. jScheduler interfaces the Resource discovery service in order to receive a set of resources, which match certain static requirements. The resource discovery service, which has to provide this information on other hand uses jResourceMatching to gather resource description in the first place.
  Publish/subscribe protocols devised in this WP will be used by jEvent and jMonitor service to distribute events through the Grid.

- **WP3.4:** Local replication algorithms are important to provide fault-tolerant task execution, monitoring, etc. What we have in mind is to create several instances of local jExecMng service, which oversee the execution of the task. If one service for any reason disappears eg. through node failure, there has to exist a backup instance, which is taking over the appropriate task. In advanced versions of jScheduler we plan to evaluate speculative executions. For this purpose, we need transactional files from WP3.4.
- **WP3.5:** The security infrastructure is an important and crosscutting concern, which we have to implement in order to achieve the global goals defined by this WP.

4. Related work

There are a large number of projects and products that try to solve the problem of providing a user friendly API that enables jobs to easily access the Grid. From these projects one can find a wide variety regarding the field, the development stage or even the purpose.

A number of projects originate from the Open Grid Forum [OGF] whose main goal is to enable grid technology for both research and business environments. They focus their effort mainly in developing open standards. Two of these projects have been a reference in the proposal of XtreemOS calls: the Distributed Resource Management Application API Work Group (DRMAA-WG) [DRMAA-Web][DRMAA2004] and the Simple API for Grid Applications Research Group (SAGA-RG)[SAGA].

The main goal of DRMAA-WG is to develop the specification of an API to enable job submission and monitoring in a distributed resource management (DRM) environment. The extent of the specification is the high level functionality necessary for a job to manage jobs submitted to a DRM. They offer a very basic set of operations to create, monitor, control (start, stop, restart, kill) and retrieve the status of a job.

The SAGA group is close to DRMAA one, in the sense that they share the same objective, but differ in the way to achieve it. The SAGA objective is more ambitious and somehow it is built on the DRMAA experience. They aim to develop a much more flexible and complex API than DRMAA. Besides this, the SAGA specification is still being discussed, and the DRMAA's was finished in 2004.

What we propose to offer with the Application Execution Management is also an easy but at the same time powerful and flexible way to execute and control jobs. By easy we mean that a job can be submitted as you would do it in a Linux environment. The goal is similar to that of the DRMAA and the SAGA groups want to achieve. The main difference with our proposal is that SAGA is more job oriented, with much more details, and in our case, we have tried to reduce the number of calls, assuming we will have some API such as SAGA that will offer jobs the level of detail they need.

The SAGA group is based on the work done by DRMAA group and GAT [GAT] layer from the GridLab [GridLab][GridLab2] project. GridLab is an European project that has developed an environment to enable developers to exploit all the possibilities and power of the Grid. It is divided in different layers. Among them the Grid Application Toolkit (GAT) is the layer that provides access to the different Grid Services. The service that supplies control, monitoring and scheduling of jobs is called Grid Resource Management Service (GRMS) [GRMS].

The GridLab project was based on some core services coming from the Globus project [Globus], in particular in GRAM services [GRAM] for job management and GridFTP

services [GridFTP] for file transfer. Globus has been the reference product when talking about solutions for the Grid. It is complete open source software that covers all the needs when working in a Grid environment. It consists of a number of components that can be used together or separately combined with others to provide solutions to a wide range of contexts. The main potential of the Globus Toolkit is that its components are decoupled enough to offer their functionalities individually but at the same time Globus itself offers a complete independent product. The Globus Toolkit component whose functionality closest to our AEM is the Grid Resource Allocation and Management (GRAM). This service is in charge of submitting, monitoring and cancelling jobs on Grid computing resources, but it does not act as a job scheduler. GRAM services are offered to construct job schedulers on top of their basic services. Other projects that also rely on Globus are, for instance GridWay Metascheduler, which also provides basic and advanced services such as opportunistic migration but always based on a centralized scheduler and Globus services.

One of the main differences of these projects with XtreemOS is that all of them envision the Grid infrastructure as something far from the OS, with a lot of levels between them, resulting in a loss of control and usually a high overhead. Our plan is to extend the OS to allow a better coordination between the OS services and the Grid ones. What we intend to do with the AEM is to unify the API, the job controlling and the job scheduling in a single component with multiple services, offering a top-down functionality for better control. This control will be two sided. On the one hand, a tight integration with the kernel will allow enforcement in the usage of the agreed resources. There will be no way to consume more resources than the ones allocated. On the other hand, users will have more information and control on how their jobs are running, not just whether they have been submitted, are running or have finished. We will be able to forward performance metrics, occurred errors, exit status etc. to users allowing them to react whenever needed.

5. Conclusions

The Application Execution Management system has to provide the required services for submitting, monitoring, and managing the execution of jobs in grid execution environments. In this document, we have described a set of services to cover job requirements expressed in WP4.2 considering grid characteristics.

Grid environments have two main characteristics from the point of view of jobs that make their usability complicated: dynamicity and heterogeneity. Grids are composed by different types of resources (storage resources, computing resources, etc) and resources with very different characteristics (instruction sets, cpus, memory, storage, etc). Moreover, these resources usually offer different services to jobs with different interfaces. Dynamicity is a characteristic that includes the possibility of using more or better resources, but that can also imply resource failure, much more probable in grid systems than in local systems. Both situations are difficult to manage by users, therefore AEM must provide automatic support to such issues.

From job requirements we have concluded that the AEM must be powerful and flexible. There are a lot of potential grid users with different requirements. Our proposal is to provide a set of basic services at system level and let the rest of required functionality be in charge of user-level jobs, for instance the case of workflow managers.

Additionally to some internal services, the AEM will include components for submitting (jScheduler service), monitoring (jMonitor service), and controlling (jController service) jobs. The control of jobs includes validating scheduling decisions and deciding on job migrations

(jMigration and jCheckpointing) in case of failures. Due to grid characteristics, job management is based on a good resource management support: discovery (Resource Management system), matching (jResourceMatching service), allocation (rAllocation service) and monitoring (rMonitoring service). We have designed an AEM system that is mainly distributed, with just two services for centralized information: the JobDirectory service (recording the list of JobIDs and how to access them), and the Resource Management system (directory of resources). The rest of services are related to specific jobs or resources, avoiding, as much as possible, bottlenecks when accessing services.

With this set of services, we can fit most of the requirements of jobs and the rest can be offered by user-level jobs using the new XtreemOS calls.

6. References

[BLSP03] R. M. Badia, J. Labarta, R. Sirvent, J.M. Pérez, J.M. Cela and R.Grima. Programming Grid Applications with GRID superscalar . Journal of GRID Computing, Vol. 1 Issue 2. Pages: 151-170 , June 2003.

[Paraver] Paraver Documentation http://www.cepba.upc.edu/paraver/

[RDF2004] Resource Description Framework http://www.w3.org/RDF/Overview.html

[RDDL2002] Resource Directory Description Language http://www.rddl.org

[DRMAA-Web] Distributed Resource Management Application API Working Group http://drmaa.org/wiki

[DRMAA2004] Distributed Resource Management Application API Specification 1.0 http://www.ggf.org/documents/GWD-R/GFD-R.022.pdf

[GridLab] G.Allen, K.Davis, K.N. Dolkas, Nikolaos D. Doulamis, T.Goodale, T. Kielmann, A.Merzky, J.Nabrzyski, J.Pukacki, T.Radke, M.Russell, E.Seidel, J.Shalf, I.Taylor . Enabling Applications on the Grid - A GridLab Overview International Journal of High Performance Computing Applications, Vol. 17, No. 4, 449-466 (2003) http://www.gridlab.org .

[GridLab2] E.Seidel, G.Allen, A.Merzky, J.Nabrzyski . GridLab—a grid job toolkit and testbed. Future Generation Computer Systems Volume 18, Issue 8 , October 2002, Pages 1143-1153 http://www.gridlab.org.

[SAGA] T.Goodale, S.Jha, H.Kaiser, T.Kielmann, P.Kleijer, G.von Laszewskik, C. Lee, A.Merzky, H.Rajic, J.Shalf. SAGA: A Simple API for Grid Applications. High-Level Application Programming on the Grid. http://wiki.cct.lsu.edu/saga/

[GAT] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann,A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Sch¨utt, E. Seidel, and B. Ullmer, The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid, Proceedings of the IEEE, vol. 93, no. 8, pp. 534–550, 2005.

[HMHL] J. Herrera, R.S. Montero, E. Huedo, I.M. Llorente DRMAA Implementation within the GridWay Framework. http://www.cs.vu.nl/ggf/apps-rg/meetings/ggf12/llorente-final.pdf

[GRAM] GT Execution Management: GRAM http://www.globus.org/toolkit/gram/

[GridFTP] GridFTP Documentation http://www.globus.org/toolkit/docs/3.2/gridftp/

[Globus] The Globus Toolkit http://www.globus.org

[Gt4] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. http://www.globus.org/toolkit/

[GridWay] The GridWay project. http://www.gridway.org/documentation/guides.php

[Maui] Maui Cluster Scheduler. http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php

[OGF] The Open Grid Forum http://www.ogf.org/

[GRMS] Grid(Lab) Resource Management. http://www.gridlab.org/WorkPackages/wp-9/documentation.html

[WSR2006] Web Services Resource, http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-cs-01.pdf