XtreemOS Application Execution Management: A Scalable Approach

Ramon Nou[1,2], Jacobo Giralt[1], Julita Corbalan[1,2], Enric Tejedor[1,2],
J. Oriol Fitó[1,2], Josep M. Perez[1,2], Toni Cortes[1,2]

Barcelona Supercomputing Center[1], Technical University of Catalonia(UPC)[2]
Barcelona, Spain

Email: {ramon.nou, jacobo.giralt, julita.corbalan, enric.tejedor, josep.oriol, josep.m.perez, toni.cortes}@bsc.es

***Abstract*—Designing a job management system for the Grid is a non-trivial task. While a complex middleware can give a lot of features, it often implies sacrificing performance. Such performance loss is especially noticeable for small jobs. A Job Manager's design also affects the capabilities of the monitoring system. We believe that monitoring a job or asking for a job status should be fast and easy, like doing a simple 'ps'. In this paper, we present the job management of XtreemOS - a Linux-based operating system to support Virtual Organizations for Grid. This management is performed inside the Application Execution Manager (AEM). We evaluate its performance using only one job manager plus the built-in monitoring infrastructure. Furthermore, we present a set of real-world applications using AEM and its features. In XtreemOS we avoid reinventing the wheel and use the Linux paradigm as an abstraction.**

Keywords: Grid Middleware, Monitoring, XtreemOS, Scalability

I. Introduction

Current Grid middlewares suffer from the lack of a global view of the system in several parts of their interface. There are also some semantic problems confusing the non-expert users who are accustomed to work with Linux-like systems. Having the concept of process related to a job can offer the Grid user an experience similar to the one he has in a Linux box, where the well-known process-thread pattern is present. The entities of jobs and processes seem more natural to the user than having multiple jobs spawned over several nodes. Working on the Grid should not be different from using 'ps' or calling *perror* inside a C program. A job and their processes should be connected and the user should be able to find all the related info seamlessly.

When a user executes jobs inside the Grid, he also needs to check their status. For that purpose, we need a monitoring infrastructure that is easy to use, fast and scalable. In other words, monitoring should be able to answer the question *"What is my job doing?"* fast. This is accomplished in AEM, where a job can have multiple processes running on different nodes. With this Job-Process concept, a user only has to check the job status to see whether all the processes are finished or some of them are still running.

The contributions of Application Execution Management (AEM) are the following:

- The concept of job with several processes related to it, similar to the Linux process-thread paradigm.
- Linux signals can be sent to jobs and processes.
- Processes can be dynamically generated and destroyed, even from other jobs.
- We can dynamically increase or decrease job resource reservations from inside or outside the job.
- Monitoring with user metrics and several degrees of information, *i.e.*, jobs, processes and threads.
- Jobs can be tagged with user dependences and several operations, *i.e.*, monitoring, signals, can be applied to them at once.
- User-defined callbacks for several events are implemented in the monitoring system.
- Kernel collaboration, via Kernel connectors, to know when a process is created or destroyed, *i.e.*, with fork().
- The whole system presents a scalable design integrated inside XtreemOS.
- Services can be replicated. The load can be distributed while keeping a global view using distributed hash tables (DHT).

In this paper we present AEM running inside XtreemOS [1]. XtreemOS is a Linux-based operating system designed to work with Grids in a transparent and scalable way and with the support of Virtual Organizations (VO), a set of users that offer resources and exploit them for a common goal. XtreemOS goal is to build a user-friendly environment, which should be extensible and powerful. This paper presents the job management infrastructure with monitoring of AEM using the concept of Job-Process. AEM performance and scalability on job management are analyzed, too.

II. Related Work

There are two widely used Grid environments, Globus [2] and UNICORE [3]. None of them can run a job with multiple processes, *i.e.*, a job is the smallest unit they can handle. Nevertheless, UNICORE supports multiple job execution using the Network Job Supervisor (NJS), a workflow manager. Regarding job scheduling, neither Globus nor UNICORE include it directly. They implement some kind of resource manager (GRAM in Globus, RMS in Unicore) that relies on external job schedulers or batch queuing systems like Condor [4], Torque [5] or SGE [6]. As it will be explained later, AEM follows an integrated approach that provides a *good enough*

scheduling, without using global schedulers. Instead it uses local schedulers and each of them operates on a part of the system jobs. This design decision is key for scalability, as it would be impossible to get the *best* system schedule while keeping performance and scalability.

With respect to monitoring, there are similar infrastructures like MDS [7], which provide monitoring in a Globus Grid, even with aggregation and accounting.

On the contrary, in AEM we separated job monitoring from resource monitoring and discovery. Job status monitoring in XtreemOS is comparable to the one in Globus GRAM service, but the former also provides some of the features seen in MDS like aggregation of monitoring information and offers more metrics to the user.

Concerning the AEM internal structure, the integrated approach is again present. Where Globus MDS offers a pluggable architecture and collecting services, we have a hierarchical organization following the Job-Process paradigm. AEM distributes monitoring data at different scopes, always choosing the one that is closest to the source on a job basis, as opposed to the resource-based approach.

About monitoring capabilities of multijobs in current Grid middlewares, they are inefficient. Monitoring a job in Globus ends up in the GRAM5 service of a specific node. UNICORE uses NJS and the UNICORE client to monitor jobs, but NJS is not able to give a global view directly. Only a VSite (UNICORE Virtual site) can be checked on each call. There is a lot of overhead to ask for a job status, more costly or impossible if we want to know what is the basic Linux status of a process of the job. Extensibility is also different from other approaches. We offer a flexible API that allows applications to specify and update values to its own metrics associated to the job. The ability to trigger callbacks for user-specified events is also an important feature.

The XtreemOS monitoring API has been designed based on our experience in earlier works such as the HPC-Europa project [8]. The aim of this project was to put together several middlewares in a single API. With this objective, we designed multiple XML schemas and templates that join different sources of information. Other projects such as SAGA [9] introduced the idea of providing callbacks associated to metrics. Our work is also based on more traditional monitoring mechanisms such as the /proc filesystem [10] in Linux or the API for monitoring in queueing systems such as IBM LoadLeveler [11].

Accessing the Grid as an interactive system is an important feature of XtreemOS. We believe that current middlewares have too many layers, which reduce their usability. AEM has some unique features. It allows one to execute interactive jobs in addition to batch jobs. Jobs, and their processes, can receive Linux signals. It also has the concept of tagged dependencies with no predefined semantic, which helps managing related jobs together. AEM provides advance reservation features and, unlike other implementations, those are dynamic. This means resources can be added at any time to a currently alive reservation, even if it is already in use. That's not the case in Condor [4] or Portable Batch System (PBS) [12], for example, and Globus does not implement them on its own. The scheduler can also be made aware of the files a job uses by specifying them on submission. In that case it tries to allocate processes near the nodes where files are stored.

III. XtreemOS

In Figure 1 we show the global architecture of XtreemOS, designed with abstraction in mind. A normal user does not see any difference from a traditional Operating System, and accesses the Grid seamlessly. XtreemOS has two layers, Foundation Layer (F-Layer) and Grid Layer (G-Layer); The F-layer supports Virtual Organizations and checkpointing, while the G-layer provides a set of services to the application layer. In this paper, we focus on the Application Execution Management module responsible for the execution and management of Jobs, their local schedule and their monitoring. AEM is not a front-end for submitting and managing jobs, it is a part of XtreemOS, a complete Linux-based Operating System. One of the key points of AEM is the support from the system kernel to increase its performance.

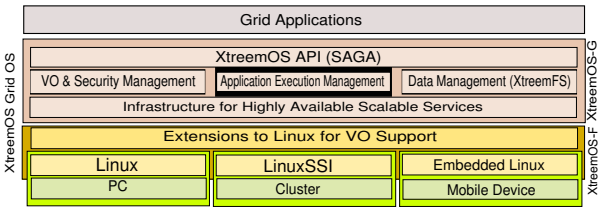

Fig. 1. XtreemOS architecture.

Details of other components can be checked on: XtreemFS, a distributed filesystem [13], checkpointing [14], resource discovery (with overlays from ADS/RSS [15], Scalaris [16] and SRDS [17]) and SAGA API [18], which provides the XOSAGA high level abstraction.

IV. Application Execution Manager (AEM)

The Application Execution Management module covers job management, job execution, monitoring and resource management on the Grid among other functionality. AEM is the access point to submit and control jobs in XtreemOS and is composed of different asynchronous distributed services. Depending on the set of services appearing in a node we can classify the XtreemOS node in three classes: core, resource node and client. Clients simply access the XtreemOS system. Resource nodes provide execution and storage capabilities. Finally, core nodes offer VO administration, job management, and eventually can export resources as well. However, each node can be particularly configured outside this classification. We can distribute the load by running the same service in different nodes. We can find similarities with web servers and the distribution of load among them.

AEM design conforms to a distributed architecture attempting to achieve a high degree of scalability. Centralised

databases are forbidden and it handles load balance by spawning several service instances on different nodes. Shared view among them is implemented by using DHTs. This design has proved to be very convenient for extensions such as the replication mechanism currently under development in order to achieve fault tolerance on scheduler services.

A. Jobs

AEM defines a hierarchy of entities composed of the *job*, *job unit* and *process* (also threads). The latter is directly a UNIX process while a *job unit* is the set of processes of the same job running on a node. A job is owned by a user, who is identified by a set of credentials. Other users can do operations on this job if they have the needed credentials. Processes can be created (even using fork) or destroyed dynamically, they will be attached to the correct job unit. Linux signals can also be issued to a job, job unit or a particular process of it. If the job depends on others (via dependences), operations can affect all of them.

B. Resources

A resource is where a job (and their processes) is executed, the job can use more than one. To use a resource, a user should reserve it. A resource can be shared among other users or jobs or be exclusive. Finally, a reservation contains a set of resources that can be used by the user (in any job) inside a period and it is identified by a ReservationID. The reservation can be grown or shrunk dynamically by the user. To reduce complexity for the normal user, reservations can be automatically created when submitting a job and will be released when the job is finished.

C. AEM Services

The main services that we find inside AEM are:

- **Job Manager [Core]**: provides all the job management features. It communicates with the Resource Manager / Reservation Manager to get resources and with the Execution Manager to execute jobs. It has several scheduling algorithms, always on a subset of the resources as there are no global scheduling policies. It stores Job related monitoring information (job status, submit time, ...).
- **Job Directory [Core]**: Distributed storage of all JobID and the address of the JobMng that controls each job.
- **Execution Manager [Resource]**: Manages execution of the jobs. Execution is performed at the process level with the Grid user credentials. Its basic unit is the *job unit*. ExecMng (Execution Manager) stores all the monitoring information related to them, this goes from process status (e.g. R for running, Z for zombie, S for sleep,...) to user and system time. Internally the Execution Manager service gets information about its processes from the kernel using kernel connectors. ExecMng is informed of any fork in its job units, binding automatically the newly spawned process to the correct job unit. At the kernel level we also know when a process ends, allowing to accelerate the notification to all related processes and to send more detailed information.
- **Reservation Manager [Core]**: The Reservation Manager is similar to a Job manager for resources. It provides reservations, allocations of resources. However we use a cache, to improve performance as it is a very used operation when submitting a set of jobs and their processes.
- **Resource Allocator [Resource]**: Stores details about the reservations of the current node. A reservation is built upon a set of local resource allocations.
- **Resource Manager [Core]**: Provides the resource matching routines with the discovery mechanism through DHTs. For example, if we are looking for machines with a least 4 GB of memory, we can specify it in JSDL format in the submit file. The Resource manager will query the DHT with the given restrictions and will return a set of machines.

D. Job Submission and Execution Flow

Figure 2) shows a typical job execution flow inside AEM:

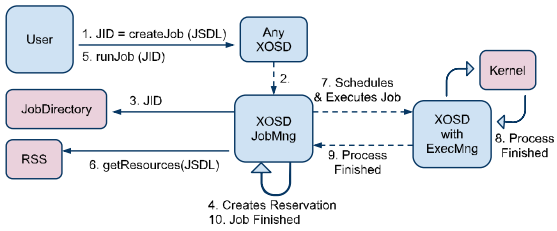

Fig. 2. Simplified Job Flow inside AEM.

1) The User submits a JSDL file to a node (Core or Resource). **2)** The node contacts/looks for a Job Manager service (maybe on another node). **3)** JobManager creates a JobID and stores it in the JobDirectory. **4)** A resource reservation is created, empty, as this will be an automatic reservation. **5)** The created job starts to run. **6)** The Job Manager asks for resources and updates the empty created reservation. We obtain resources from a DHT (Dynamic Hash Table) infrastructure (RSS). **7)** The First process of the job is created into the selected resource node (using ExecMng service). **8)** Resource nodes (Execution Manager) detect when the process is finished through the kernel connectors. **9)** Execution Manager sends the finished state event to the Job Manager. **10)** Job Manager detects that no more processes (*job units*) are running on the job, signaling the job as finished.

We can consider additional steps here, first a process can create a fork() or call createProcess to start a new process. The new process will be related with the Job. This relation in the case of the fork will be acknowledged via kernel connectors to the ExecMng.

E. Monitoring in XtreemOS

In this subsection we will detail the monitoring infrastructure which is distributed between the Job and Execution Manager. As a summary, monitoring has the following features: System metrics and user defined metrics, different levels of

information (from jobs to process and threads), buffering and callbacks.

The monitoring infrastructure provides a set of features not fully supported in other middlewares. The XtreemOS philosophy is to offer an experience similar to a normal Linux system: getting job status should be as easy as doing a 'ps' in a Linux box. The monitoring system tries to mimic this with the Job-Process concept. XtreemOS monitoring should be easy for the user, but powerful for the system and extensible for both. The ability to include user metrics inside the system provides with a valuable resource for research in performance. One of the recent extensions that made use of this, is the publication of GPS coordinates from a mobile device [19] and access them from any job. The same API used for internal tools is offered to the user to instrument its applications.

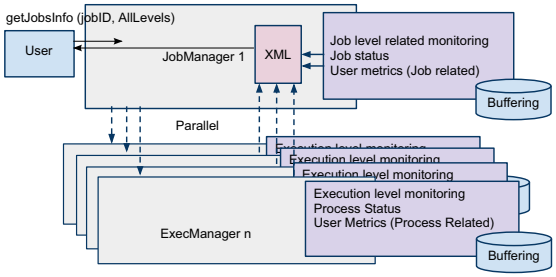

Fig. 3. Monitoring infrastructure

We will introduce some of the monitoring semantics (metrics, levels of information and scope):

A *metric* is the description of a kind of monitoring data. It can be created by the user or by the system. A metric has a description, a value type (boolean, integer, timestamp, ...) and a scope. A *scope* (`JOB`, `JOBUNIT` or `PROCESS`) is the place where the info belongs. A `PROCESS` scope metric is stored in the Exec Manager, as the value is related to a process, in a similar way a `JOB` scope metric is stored in the Job Manager.

The AEM monitoring service is used by different types of jobs which require different levels of information and, potentially, by other XtreemOS services. There are three *levels of information* (JOB, PROCESS and KERNEL). Each level adds extra cost to access the information as AEM needs to collect it from an additional source. For example, the KERNEL level needs support from the kernel via PAPI [20] or LTTng [21].

When we ask for a metric value we go through the Job Manager to get the list of available metrics. Job Manager contacts all involved Execution Manager in parallel to get the values. Like all services in AEM, if the contacted instance of JobMng does not contain the requested information, the request is redirected in a implicit way to the appropriate one. Metrics, ranging from job status to the distributed information about processes and threads running in the Grid, are stored in several services depending on their nature, as shown in Figure 3. This division is natural and helps to distribute the job information among the nodes involved in the job life cycle.

With AEM monitoring we can get information about job status (Submitted, Running, Done, Failed, Suspended) and process status. Process status, among other configurable features, could be obtained via LTTng providing a lot of information with a low overhead. This feature is unimplemented since it requires heavy changes to the kernel. Nevertheless, reading LTTng buffers will be easy to implement as an extension when available. Currently, we get process status from a daemon in the nodes reading /proc/pid and storing the resulting values.

Users and the middleware can augment the monitoring information with their own metrics, as we introduced before. A user can register a new metric into the Job Manager or Execution Manager, depending on the scope, and pushes metric values that will be read by other processes of the same job, by another job of another user (based on permissions in the Virtual Organization), or appear in some system utilities. The access to this information is granted on a per job and user certificate basis. These metrics behave like the predefined system ones, so they will get all the features of this design.

A practical example of this feature could be instrumenting an application with PAPI and exporting its counters to the monitoring infrastructure. An advanced user could write a workflow manager that accesses to this information and takes decisions based on it towards the instrumented job, like modifying its resource allocation. Subsection VI-B proposes a prototype for a web server scheduler.

We can also enable buffering for a metric. This way a circular buffer is allocated for that particular metric and the user who requested it. Buffering increases the scalability of the system while reducing the system overload that polling would generate if a user wouldn't want to miss any value, *i.e.*, tracing utilities. Since buffers can become full, a user may program a callback to signal when a buffer is being almost full. We can also set callbacks on events like "Value X is greater than Y". Having the monitoring interface integrated into the system is a key point. XtreemOS system utilities like *xtrace* and *xps* can use those values and generate a global trace of the jobs of an application (from job to thread level) that can be analyzed inside a graphical utility like Paraver [22]. With this tracing feature implemented inside the system we give users a powerful capability to analyze the behaviour of their jobs. Due to its XML nature this monitoring information can be easily stored as a trace.

The user interface is simple and is shown in Table I. The XML format offers the user (and system utilities) an easy way to build tools using monitoring information. XML is used for interoperatibility, more performance could be obtained if a non-XML solution is used.

V. AEM Performance

In this section we will analyze how AEM by itself (without DHTs) scales to hundreds heterogeneous nodes (results presented with a hundred nodes) while offering a set of features not available in other software such as the job-process paradigm, kernel integration and convenient monitoring. Beyond that, in very large systems (thousands/millions

TABLE I
MONITORING INTERFACE METHODS

Method	Description
getJobsInfo(jobIds, flags, infoLevel, metrics)	Gets XML with requested information, detail level and metrics.
getJobMetrics(jobId)	Returns the list of available metrics for a specific job.
setMetricValue(jobId, metricName, resourceID, pid, value)	Sets the value of a Metric.
setMonitorBuffering(jobId, metricName, resourceID, pid, flags)	Switches on and off buffering for the specified metric.
addJobMetric(jobId, MetricsDesc)	Adds a new user defined metric to the job.
removeJobMetric(jobId, metricName)	Removes a user defined metric from the job.
addMonitoringCallback(String jobId, MetricEvent metricEvent)	Adds a monitoring callback expecting an event to fire.

of nodes), our system scales adding more jobMng (among other) instances which keep a shared view of the Grid using DHTs (Distributed Hash Tables). We provide performance and scalability results for job submission and job status query.

The performance targets of AEM are being as efficient as Globus [23] in the aspects with similar features and yielding similar efficiency for improved ones.

We describe in the next subsection our test environment plus the different experiments.

A. Environment

Our first scenario uses a Grid5000 (G5K) node with an installation of Globus Toolkit 5 (out-of-the-box) and an XtreemOS core node. It shows a basic and comparable unit of execution for XtreemOS and Globus. The node is a quad-core with an Intel Xeon at 2.33GHz processor. The second scenario uses a setup formed by 100 heterogeneous nodes where we deploy one XtreemOS core and 99 resource nodes. On this second scenario we run scalability tests. We don't compare directly with Globus as we make use of different features. C.i. of 95% are used for the intervals.

B. Job submission and execution experiments description

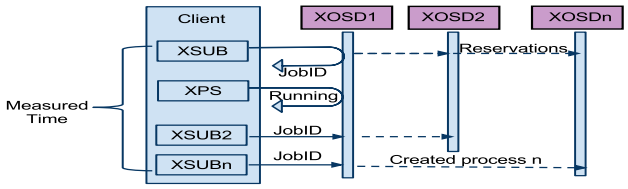

Fig. 4. Job submission scalability test diagram.

In this subsection we will describe the job submission and execution experiments using environment 1 for performance and environment 2 for scalability. The first test compares the submission and execution time with XtreemOS and Globus. This test only uses features available in both. More precisely, XtreemOS is not using resource location services, XtreemFS, reservations nor the Job-process paradigm. The test measures the performance of the middleware by submitting a job that just runs `/bin/true`.

The second test, presented in Figure 4, compares the scalability on the submission of a job with n processes in n resources. 95% confidence intervals are used on all evaluations.

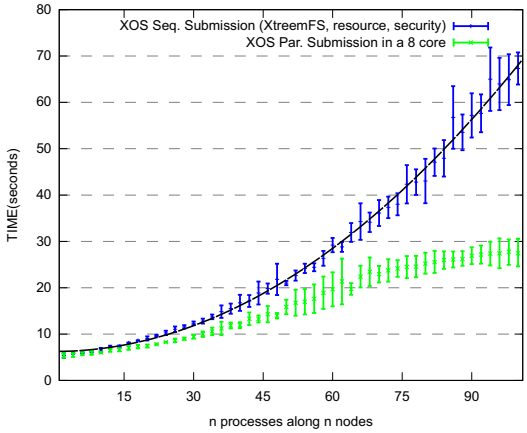

Fig. 5. Job Submission scalability (Parallel - Sequential).

1) Submission and Execution Results: The performance results in terms of job execution compared to Globus Toolkit 5 in the first environment are the following: the execution time is lower in XtreemOS for the same job. We have [0.9885, 0.9921] seconds for Globus and [0.5471, 0.5537] seconds for XtreemOS. Note that Globus Toolkit 5 already reduced by half the time of submission from version 4.2.

In Figure 5, we show the scalability results in the second environment. This test uses the automatic reservation mechanism. In these results we have the resource discovery and reservation overhead for the 100 nodes, plus XtreemFS mounting. When submitting a process, a suitable node from the reservation must be found and selected. We use a random scheduler but others, such as Round Robin or Least Used Resource, are also implemented. The lower line shows the results of the same test submitting the $n-1$ processes in parallel in an eight core XtreemOS node for reference. In summary, the cost of sequential job submission is $0.0062x^2$ for a random scheduler in the actual scenario (where x is the number of processes created). This x^2 constant cost will be reduced with some optimizations inside the code such as reducing credentials checking using Single Sign On technology. However it will be difficult to reduce the x^2 bound without reducing features. This cost is produced mainly by the resource timetable checking, as resources and reservations are dynamic and cannot be centralized for scalability reasons. Nevertheless, submitting jobs without processes reduces those checks and lowers x^2 bound. Scalability in the x^2 scenario is obtained distributing the jobs load between different jobMng,

for example a VO can have his own JobManager and still have a Global view of the system via the Job Directory (DHT). The cost is based on local network times, and is bounded by its latencies. To reproduce a similar scenario in Globus we would need a Job queueing system like Condor and multijobs (GT5 capability). But XtreemOS provides the Job-Process concept relating job to their processes, and Globus considers them different jobs. However we included Globus in the results subsection when submitting a single job.

C. Job Status experiments description

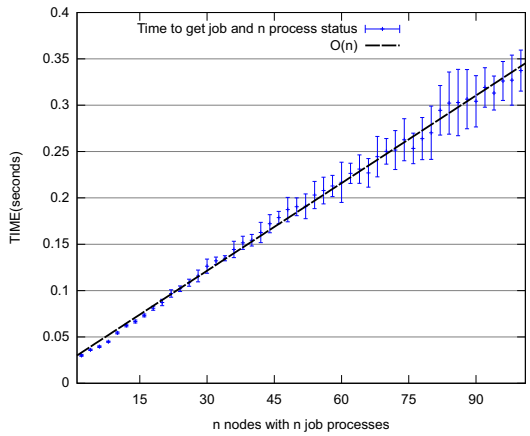

Fig. 6. Scalability of a status query.

The critical path for a workflow manager is to find, quickly, if the submitted job has finished to avoid a delay of the whole workflow manager. To test the features of XtreemOS in this part, we used the first environment to compare the performance of obtaining the job status of a large job (sleep) between Globus and XtreemOS and the second environment to measure the scalability of asking for the job and process status of a job (master process) and its $n-1$ processes distributed along n nodes. In the second test, the user request goes to the JobMng for the whole job status and through all the nodes in the system (ExecMng) for the process status. In this occasion again, Globus is not comparable in the second environment.

1) Job Status Results: The results of a single job status query in the first environment are the following: with GT5 we have [0.014006, 0.014041] seconds, in XtreemOS we have [0.024236, 0.024551] seconds. Means are, 0.014023 and 0.024393 respectively. GT5 also improved job status one order of magnitude over GT4.2. However, its job status reply is for one job and does not give more information than "ACTIVE". XtreemOS gives extra information such as submit time, process status, CPU time and user id in the same call. Without process status, we obtain mean times of 0.012 seconds.

Figure 6 shows the time of getting the process status as the number of processes increases. In Globus we would have to make one job status query to each node with a different JobID. Additionally, in XtreemOS, as Execution Manager requests are done in parallel, the time to ask all involved Execution Manager, in order to gather results from each individual process, is reduced. This is why the time is lower than $O(n)$. Checking job status in a loaded system does not imply an overhead as far as we distribute the load between different Job Managers. The result does not depend on the number of jobs running in the system but only on the number of nodes used for each job. It's worth noting that we can also ask only for job status. To do so, we select a smaller metrics set as explained in IV-E. Removing the communication with Execution Manager, the line is constant and independent on the number of processes.

VI. Applications

In this section we will introduce a couple of applications which have been ported to XtreemOS and AEM.

A. COMP Superscalar (COMPSs) - hmmpfam

COMPSs [24] is a framework that facilitates the development and execution of Grid-unaware Java applications. It is composed of a programming model and a runtime.

In the COMPSs programming model, the user selects a set of methods of a sequential Java application for them to be run on the Grid. At execution time, COMPSs instruments the application and automatically replaces the local invocations to these methods by the creation of remote tasks.

The COMPSs runtime is in charge of optimizing the perfomance of the application by exploiting its inherent concurrency. It receives the tasks from the application, checks the data dependencies between them and decides which ones can be run at every moment and where, considering task constraints and performance aspects.

In order to test COMPSs on top of XtreemOS we chose hmmpfam, a bioinformatics application for protein sequence analysis. Hmmpfam is computationally intensive and embarassingly parallel, which makes it a good candidate.

1) Test Results: In order to evaluate COMPSs-hmmpfam on top of XtreemOS-AEM, we conducted some tests to measure the execution time of the application. For the executions, we used one machine as master node (hosting the COMPSs runtime) and a variable number of worker nodes (resources that run the application tasks). Besides, to compare the XtreemOS performance, we ran the same series of tests using two different configurations: first, the COMPSs runtime ported to XtreemOS, making use of the AEM Java-XATI API and, second, the original COMPSs runtime, using the JavaGAT interface and its SSH adaptor to submit jobs and transfer files, thus obtaining a lower bound.

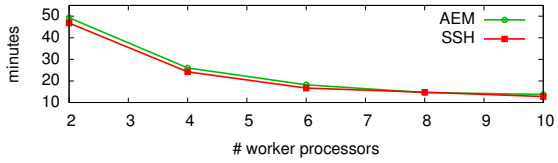

Fig. 7. Execution times of the hmmpfam application on top of COMPSs, both with the XtreemOS and the SSH flavours.

Figure 7 shows the execution times of running COMPSs-hmmpfam, both using XtreemOS and JavaGAT-SSH, with

a range of two to ten worker resources. We see how the results are quite similar for both configurations. For two processors, there is a small difference in the execution time which is progressively reduced for higher numbers of worker processors. Given that in all the executions the number of submitted jobs is the same, the number of job-to-job transitions that occur at each worker resource decreases as the number of workers increases. Since the slight difference in performance between SSH and AEM takes place at job transitions, the difference in execution time is more noticeable when the number of workers is low.

In light of these results, we can conclude that AEM is able to keep up with SSH job submission, while offering more functionality like resource management.

B. On-demand performance scalability of web servers

The job management and monitoring infrastructures inside AEM offer a spectrum of possibilities regarding web server management, operation and performance. As a matter of fact, on-demand scalability of those servers, according to typical time-varying workloads of web applications, is a research challenge that needs to be addressed by the community. For instance, Fito et al. [25] address web servers elasticity in a Cloud environment. In this sense, we contemplate XtreemOS as a distributed and scalable Grid operating system which allows us to scale the performance offered by web servers according to the changing demands of web applications deployed on them. The approach presented leads to the elasticity enactment of web servers into a Grid environment. Figure 8 shows the architecture of the system proposed here. Mainly, we present an intermediate layer between clients and web servers running on a pool of Grid resources. This in-between layer is composed by two interconnected components: *Proxy* and *Scheduler*. On one hand, the proxy server is in charge of balancing the clients' requests between the available web servers. On the other hand, the scheduler has a key role in the system, *i.e.*, implements the dynamic resource management policies and communicates with *AEM-XOSD* component to: ask for monitoring information, request for resources reservations needed, and update these reservations according to the requirements of the web applications deployed on the servers. For our purposes, we take advantage of AEM monitoring capability through the definition of the following user metrics: utilization of the CPU, memory, network and I/O devices of each web server. The description of these high-level metrics allows the scheduler to be aware of resources constraints and to properly act to solve the undesired situations of web servers overload that affect the performance growing. Furthermore, by using the buffering mechanism of AEM we do not lose any monitoring event.

With this information, the scheduler is able to take scalability decisions, *i.e.*, scale up and down the total number of web servers in order to meet the web applications' demand. This scaling operation is done by updating the resource reservation. In fact, when the scheduler asks for more or less resources, the AEM-XOSD returns a list of available resources. Then, the web server, as a new process running inside the job, will

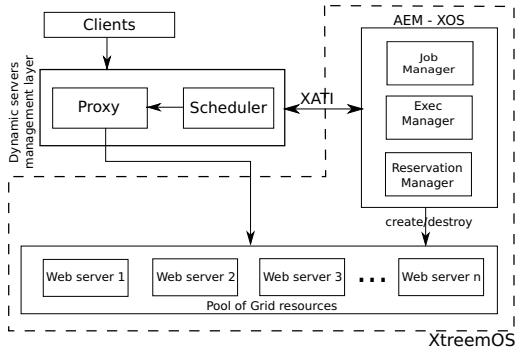

Fig. 8. Dynamic resource management layer architecture

automatically run on this new reserved resource due to the usage of the scheduling hint *ONE_PER_NODE*. Therefore, by using XtreemOS capabilities we are able to overcome a current web applications' limitation: non-scalability.

1) Results: We perform a vertical scalability of the server when reserving resources with different number of processors, thereby obtaining the most representative web server's performance metrics, *i.e.*, throughput and response time. The testbed is a set of 12 nodes with XtreemOS.

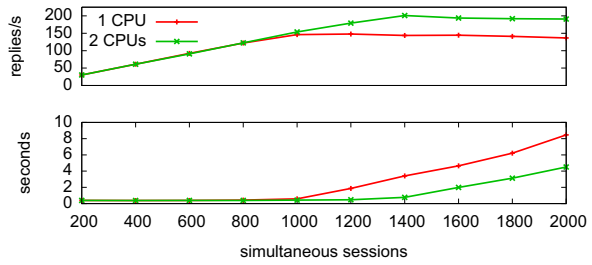

Fig. 9. Banking throughput (request per second) / response time (in seconds) when running with different number of processors

Figure 9 illustrates the servers throughput and response time if we scale vertically the number of available processor units. Note that the results were extracted from a server which runs in previously reserved XtreemOS resources.

Throughput and response time metrics are regarding the banking application of the SPECweb benchmark [26], and are expressed as a function of simultaneous user sessions emulated by the clients of the benchmark. The bottleneck resource in this application is the processor unit. It is noteworthy that the throughput of the server increases proportionally to the number of user sessions until a saturation point is reached. The same pattern of quality of service degradation is observed in the response time metric: it remains more or less linear, but increases sharply when the server becomes overloaded. Moreover, there is another possible scenario: the horizontal resource scalability, *i.e.*, replication of load balanced web servers. In this case, the performance of a single web server is multiplied by the total amount of servers minus the overhead introduced by the proxy to make the load balancing between them. Anyway, the results show the performance scalability that could be achieved by

using the dynamic resource management approach presented.

Conclusively, we demonstrate how the proposed system is able to provide web servers' performance scalability by using both job management and monitoring mechanisms of AEM. This preliminary results demonstrate the motivation and feasibility of the use-case scenario described above.

VII. Conclusion

In this paper we presented AEM, a job management infrastructure designed to be highly scalable and with features not found in other similar middlewares, like including user metrics in the job scope in a simple way. We can retrieve all job and their processes information faster.

XtreemOS is using this Application Execution Management component to offer a seamless experience to the user. This Grid system behaves at user level as a Linux box. In particular the Job - Process management provides a powerful set of capabilities, like monitoring all job processes at once, enabling more performance and lowering the load in the system.

We presented a preliminary evaluation of the system. We achieve more performance than Globus with a similar set of features except for Job status in one node, although we offer also process information. The additionally included features scale in a controlled way.

The different tests done in this paper, and the real use of the system during the implementation phase, marks a tendency that scalability will be as good as was designed for. More precisely the decentralized design of job metrics inside JobMng and processes of a job unit inside its ExecMng node, decouples information providing a higher throughput. Only jobs with a higher number of processes, distributed among a large number of nodes, may be affected. When this situation is produced, the user can reduce the depth of the information and obtain job status without process status; this will cut the utilization from n nodes (ExecMng) to 1 (JobMng) and get faster if the job is running or not. This decentralization is followed in every layer of the AEM, avoiding scalability problems. Features like monitorization from job-level to thread-level plus buffering allow more flexibility to the users while keeping the complexity and requirements of their software low as they are integrated in XtreemOS. In particular, buffering and callbacks implementation cut the system overload in general and increase the scalability of the system. Also collaboration with the kernel via kernel connectors gives a valuable information to track processes inside jobs, even when they are created via a fork. Advanced features like applying Linux signals to jobs, tagging jobs with user dependencies or creating user metrics that can be checked and updated by the user, are integrated in AEM. This integration gives a seamless Linux-box experience for the novice user, but working on the Grid.

Acknowledgment

This work was partially supported by the EU IST program as part of the XtreemOS project (contract FP6-033576), by the Spanish Ministry of Science and Technology under the TIN2007-60625 grant, and by the Catalan Government under the 2009-SGR-980 grant. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. P. Prieto, and A. Reinefeld, "XtreemOS: a Vision for a Grid Operating System," 2008. [Online]. Available: http://xtreemos.eu/publications/research-papers/xtreemos-cacm.pdf/download

[2] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," in *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, 2005, pp. 2–13.

[3] UNICORE - Uniform Interface to Computing Resources. [Online]. Available: www.unicore.eu

[4] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 323–356, 2005.

[5] G. Staples, "Torque resource manager," in *SC '06*. New York, NY, USA: ACM, 2006, p. 8.

[6] W. G. S. Microsystems), "Sun grid engine: Towards creating a compute power grid," in *CCGRID '01*. Washington, DC, USA: IEEE Computer Society, 2001, p. 35.

[7] A. Chervenak, J. M. Schopf, L. Pearlman, M.-H. Su, S. Bharathi, L. Cinquini, M. D'Arcy, N. Miller, and D. Bernholdt, "Monitoring the Earth System Grid with MDS4," in *E-SCIENCE '06*, 2006, p. 69.

[8] FP 6 EU Project-HPC-Europa. [Online]. Available: www.hpc-europa.eu

[9] SAGA Web Page. [Online]. Available: saga.cct.lsu.edu

[10] /proc information. [Online]. Available: propcs.sourceforce.net/

[11] loadLeveler job queue. [Online]. Available: www.ibm.com/systems/software/loadleveler

[12] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson, "The portable batch scheduler and the maui scheduler on linux clusters," in *ALS'00*, 2000, pp. 27–27.

[13] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "The XtreemFS architecture—a case for object-based file systems in Grids," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 17, pp. 2049–2060, 2008.

[14] J. Mehnert-Spahn, T. Ropars, M. Schoettner, and C. Morin, "The Architecture of the XtreemOS Grid Checkpointing Service," in *Euro-Par '09*, 2009, pp. 429–441.

[15] P. Costa, J. Napper, G. Pierre, and M. van Steen, "Autonomous Resource Selection for Decentralized Utility Computing," in *ICDCS '09*, Washington, DC, USA, 2009, pp. 561–570.

[16] T. Schütt, F. Schintke, and A. Reinefeld, "Scalaris: reliable transactional p2p key/value store," in *ERLANG '08*, 2008, pp. 41–48.

[17] E. Carlini, M. Coppola, P. Dazzi, D. Laforenza, S. Martinelli, and L. Ricci, "Service and Resource Discovery Supports over P2P Overlays," in *IEEE International Conference on Ultra Modern Communications*, 2009.

[18] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. V. Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf, "SAGA: A Simple API for Grid Applications. High-level application programming on the Grid," 2008.

[19] A. Martnez, S. Prieto, N. Gallego, R. Nou, J. Giralt, and T. Cortes., "XtreemOS-MD: Grid computing from mobile devices," in *Mobilware 2010*, 2010.

[20] J. Dongarra, J. London, S. Browne, N. Garner, K. London, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," 2000.

[21] Linux trace toolkit manual. [Online]. Available: lttng.org

[22] Paraver - trace visualization and analysis tool. [Online]. Available: www.bsc.es

[23] X. Zhang, J. L. Freschl, and J. M. Schopf, "A Performance Study of Monitoring and Information Services for Distributed Systems," in *HPDC '03*, 2003, p. 270.

[24] E. Tejedor and R. Badia, "COMP Superscalar: Bringing GRID superscalar and GCM Together," in *CCGRID'08*, May 2008.

[25] J. O. Fitó, I. Goiri, and J. Guitart, "SLA-driven Elastic Cloud Hosting Provider," in *PDP'10*, February 17–19 2010.

[26] SPECweb2005 benchmark. [Online]. Available: www.spec.org/web2005