Deliverable D2.2

Resource Management Systems for Distributed High Performance Computing

VERSION Version 1.1

DATE February 4, 2011

EDITORIAL MANAGER Yann Radenac (Yann.Radenac@inria.fr)

AUTHORS STAFF

Yann Radenac	IRISA/INRIA, Rennes (MYRIADS team)
Alexandre Denis	LaBRI/INRIA, Bordeaux (RUNTIME team)
Cristian Klein	LIP/INRIA, Lyon (GRAAL team)
Christian Perez	LIP/INRIA, Lyon (GRAAL team)

Resource Management Systems for Distributed High Performance Computing

Yann Radenac, Alexandre Denis, Cristian Klein and Christian Perez

Abstract

This report analyzes how computing systems manage pools of resources that execute high demanding applications. A pool of resources may contain heterogeneous resources where some of them are more efficient than others but use a special interface. Resource management systems provide services that simplify resource administration, resource programming and resource usage. Existing resource management systems implement these services using different approaches about machine organization, resource representation, resource discovery, scheduling, etc.

Especially, the resource management in the following systems are targeted: XtreemOS, DIET and Padico.

Contents

1 Introduction

High performance computers are now using many resources in parallel to answer the high demand of numerical applications. Instead of relying on the performance of always faster and larger resources to execute applications, a high performance computer is now based on a pool of resources. The more resources are added to the pool, the more powerful the computer is. For example, computing power is increased by using several processors or cores at the same time; storage capacity and data throughput is increased by using several disks in parallel; network bandwidth is increased by using several network links simultaneously; and the generic power is increased by using several generic interconnected machines in parallel.

However, setting up, programming and using pools of resources are complex tasks. Resources may be heterogeneous, numerous, and unreliable. Heterogeneous resources must be set up to be interoperable; and applications may include specific code to use these specific resources. A large scale pool of resources is difficult to administrate; and applications must implement scalable algorithms to be able to use a large number of resources without degrading performance. The larger the pool of resources is, the more likely some resources may fail, change or move during a computation. So, an application should take into account the dynamic changes in the pool of resources. Such powerful computers are also shared between several users, each running several applications. Managing all the users and their access to the shared resources of the pool is also a complex task, and must not be managed by applications themselves.

Resource management systems (RMS) have been designed to simplify the management of pools of resources. Let have a very general definition for RMS that includes batch systems, grid middleware systems, grid operating systems, clouds, etc. A RMS manages the pool of resources and provides some simplified resource services. Ultimately, a RMS should provide a transparent resource management and a very simple interface [24]; but most existing RMS only provide an intermediate interface where some resource management services are transparent and other services must still be directly managed by the administrator, the programmer or the user. For example, a RMS may provide a resource discovery service, but no automatic scheduler; in that case, the application programmer has still to code explicitly the deployment of the application.

Previous studies of RMS already provide generic descriptions of popular RMS. In [11], the authors describe the basic requirements that should be delivered by any RMS and provide a generic layered architecture. This work led to the implementation of the popular RMS Globus. In [17], some RMS are classified according to the different choices that have been made to built them, such as the resource model, the provided qualities of service, the resource discovery mechanism, the scheduler organization, etc.

Concerning the COOP project, the previous RMS studies are not enough. Even if the previous studies provide good overviews of RMS, they do not include some modern systems we are interested in, namely XtreemOS, DIET and Padico. Moreover, they do not focus on the main problem addressed by the COOP project which is the relation between RMS and programming model frameworks (PMF).[1] In this report, we describe RMS and the relation between RMS and PMF.

This report is organized as follows. Section 2 describes the interface provided by RMS according to the roles connected with RMS. Section 3 shows the most usual architecture choices made to build RMS in order to provide these services. Section 4 reviews some RMS: first resource management in XtreemOS, DIET and Padico systems are detailed, and then a few batch, grid and cloud RMS are reviewed.

2 Services provided by Resource Management Systems

Different roles interact with RMS. These roles have different objectives and needs regarding resource management. A RMS should provide services that satisfy those needs. There are three main classes of roles: the system administrators, the developers and the end-users [15]. The users of a RMS

[1] The COOP deliverable D2.1 provides an analysis of some PMF for high performance programming.

form a kind of chain: the administrator provides an infrastructure service, that is then used by the programmer to provide an application service, that is finally used by the end-user.

2.1 Services for the administrator's role

The administrator's goal is to set up a pool of resources and ensure their availability for programmers and end-users.

There may be so many resources in a pool that several administrators may have to manage one pool. Moreover, the resources that are part of the pool may be located in distant locations and so there should be at least one administrator per location.

Part of the work of a resource administrator is to set up the hardware. It consists in setting up a network, plugging and powering on the resources, ensuring the power supply, the cooling system, etc. But, in this report we just focus on the software that manages the pool of resources.

2.1.1 Add and remove resources

A basic administration operation consists in adding and removing resources from the pool. The RMS could provide a service to simplify the hot-plug and hot-unplug of resources. There should be no global interruption of the availability of other resources during that operation. The system implementing the RMS could be automatically installed on new resources added to the pool. For example, it could be unattended and through the network. It should also be easy to add a lot of resources at once; for example, merging pools of resources (like clusters) should be straightforward.

2.1.2 Configure resources

Resources that are part of the pool should be configured to specify how they are shared within the pool. The administrator should specify the description of the new resource, the access modes to that resource, the users that are allowed to use it, what services are provided by that resource, etc.

2.1.3 Detect and correct failed resources

To ensure the availability of resources, an RMS should assist the administrator to detect and correct abnormal behaviors of software or hardware resources. When a software failures is detected, the RMS should first try to self-repair if possible. For example, it could restart a non-responding service, reboot the system running locally on the resource, reinstall a local system, etc. A RMS should also provide to the administrator some sensors on the resources to predict and detect hardware failures, like temperature CPU monitoring, disk errors, disconnected cables, power supply sensors, etc.

2.1.4 Update resources

The administrator may need to update some resources because, for example, the some new applications require new version of services or new services, some services may become obsolete, a new version of the RMS system itself that fixes some bugs or add new functionalities may be available, etc. Thus an administrator has to update the software on the resources. But upgrades should be soft upgrades, i.e. without interrupting the whole pool and the availability of most resources.

2.1.5 Optimize resources

It is also part of the administration role to optimize the pool of resources. It consists in identifying the most solicited resources which are the bottlenecks of the pool. To optimize the pool, these resources could be replaced by more powerful resources, or the network configuration could be re-thought to better balance the load. A RMS should provide a service that monitors and profiles the resources usage and identifies those hot spots among the resources.

2.2 Services for the programmer's role

The programmer's goal is to write the program that will control the execution of a given applications for a given end-user over a pool of resources. That program describes how to map an application over some resources according to the application requirements and according to the end-user's rights. Moreover, the execution of applications should be efficient (i.e. scalable) since the main reason using a pool of resources is to gain on performance. And that program could also take as parameters some qualities of services requirements that must be satisfied by the execution.

A RMS provides some services that ease writing that execution program, i.e. it provides an execution programming interface. These services help the programmer to adapt some resources to match application's needs and end-user's rights. They also simplify the work of the programmer to ensure that the execution program satisfy the required qualities of services.

The programmer uses the programming interface provided by the RMS: an Application Programming Interface (API) that usually consists in a programming language and some libraries (i.e. services). It provides sometimes a toolkit and an Integrated Development Environment (IDE). The whole set of the interface, the tools and the environment is called a Programming Model Framework (PMF).

RMS may provide PMF of different levels. Some RMS provide a relative low level PMF where the programmer has to query some resources from the RMS, and then has to specify how to use them to run the application. Other RMS may provide higher level PMF which manage automatically the resources; they usually provide a configurable scheduler.

Several requirements have been identified [23] to help programming resources usage.

2.2.1 Discover resources

The resource discovery service (or resource brokering service) finds available resources which fulfills specific constraints or fits certain parameters.

A programmer may specify some required resources for several purposes. First, a programmer may have made some mandatory choices to write her application; so that the application will not execute if these choices are not met. For example: specific libraries, specific services, compilation towards a particular platform, data with restricted access rights, number of processes, etc. So she must precise some requirements on the resources that will be chosen to run the application, such as the type, number, size, network interconnection, available services, exclusive access, etc. Second, a programmer may also recommend some special resource configurations for better performance. Third, a programmer may also specify some specific resource configurations for debugging. The RMS should record a given set or scenario of resources that leads to an unwanted behavior, such that, that set of resources may be replayed to help find the bugs.

The pool of known resources is either static or dynamic. For example, if the RMS manages some resources inside one machine (like several network cards), or a small cluster of machines, the discovery can be performed once when the RMS is started and then it is assumed that this pool of resources will never change. For pool of large scale, it is not realistic to consider the pool of resources to be static, but dynamic; so the discovery of resources should be permanent to detect new resources during computation.

The subset of resources used to execute a job may be fixed at invocation time, or may dynamically evolve. The discovery service may be called once when preparing the execution of a job which will then run on the found resources until the end (fix set of resources). Or the service may be called during the execution of an application to adapt to the new needs that appear during the execution (dynamic set of resources).

Being less demanding and leaving some features unspecified increase the chances to quickly find some resources and to start earlier the program (low latency), even if not optimal. Being less demanding also helps to adapt to the restrictions of the end-user (rights and cost). So, an application should be able to adapt to different sets of resources to increase its portability.

The discovery service could also generate events and notify running jobs of the availability of new resources. If the application of the job can be adapted, the new resources will be use to increase the performance during the execution of the job.

2.2.2 Deploy processes and data over some resources

The programmer has to describe how to deploy the processes and data of the distributed application on a set of resources. The aim is to find an efficient mapping so that the application runs as soon as possible and as fast as possible.

Different mapping are possible to increase the performance by overlapping communication and computation and avoid idle time. If the execution programmer knows how the application behaves, she can anticipate the resource needs and set them up before the application actually uses them.

For example, the programmer should bother about what data are required by what process and puts them together on the same resource or on close resources; what message quantity (number and size) is exchanged between what processes and puts highly communicating processes on fast networks; what processes involve a big amount of computing and puts them on the most powerful resources; trying to overlap communication and computation; managing failures; etc. Some choices may be contradictory and some tradeoffs must be reached.

2.2.3 Ensure some qualities of execution

There are jobs that require some guarantees like some real-time constraints. Some end-user may also ask for some additional properties like better performance, or security enforcement.

Resources should be tuned to ensure those qualities of execution services. When an application has been deployed overs some resources, the local execution of the different parts of the jobs depends on the local policies of each resource.

Jobs may be executed according to a best effort policy: jobs are executed on local resources as they arrive. However, the best effort policy does not ensure that the resources are equally shared between users, nor that the different parts of a distributed job are enabled at the same time to improve the performance if the job parts exchange many messages.

To ensure some qualities of execution, jobs should be scheduled on resources. The execution programmer should allocate a resource by reserving a time share on each resource that will run a part of the job. Each resource maintain a schedule of its usage. If possible, resources could be co-allocated to ensure that the different part of a same job are running at the same time.

It is essential for large scale pools of resources that a user does not need to manually coordinate the access to resources. To this end, efficient functions are required which automatically select and schedule resource allocation for jobs. A RMS should provide an automatic scheduler that will allocate resources automatically and try to ensure some qualities of execution. It should be based on a resource monitoring service that will notify the scheduler when some qualities of services are not met.

2.2.4 Account resource consumption

In many situations, accountability for the resource consumption is required. Especially the inclusion of cost considerations for establishing business models requires functions for economic services. This includes information about the resource consumptions as well as financial management of budgets, payments and refunding. An account service helps to keep a trace of the resource usage of a job. For example, it may be used to set up a bill, or used for debugging purposes.

2.2.5 Program with a high-level interface

Programming a pool of resources is a difficult task. There are many possible configuration of resources, there are many aspects to take into account such as failures, huge number of resources, data location, security, scheduling, etc. This leads to a huge number of scenarios where the programmer has to specify how to execute efficiently his application.

A RMS should provide a high level interface where the programmer and end-user do not have to bother about resource management and let the RMS manage them automatically. Programmers may use a high level Programming Model Framework (PMF). For example, some RMS are specialized towards a particular “software architecture”; like Google who has built a specialized pool of resources

managed by a specialized RMS that can be programed by a PMF based on the map-reduce software architecture [7]. And end-users may only specify a price, an application, its parameters and some qualities of services and never know about the resources which execute their jobs.

2.3 Services for the end-user's role

The end-user's goal is to run some applications on the pool of resources and to use the result of those applications. She needs to get access to the resources, to manage her jobs and to specify some qualities of services.

2.3.1 Get access to resources

To get access to resources, an end-user must be first identified (authenticated) and then she must gain the rights to use some specified resources (authorization). The end-user needs to register with the RMS to obtain an identifier. Then she needs to contact an administrator to obtain the rights to use some resources. For example, in grids RMS, user definitions and resource accesses are managed through the notion of Virtual Organizations (VOs) [13]. The grid RMS provides a registration service, an authentication service and an authorization service. Ideally, the end-user should only log in once to access all the resources she is allowed to access; she should not log in explicitly on each resources individually.

2.3.2 Execute applications

A job is an instance of the execution of one application in the RMS. An end-user starts a job by running the execution command, which is also called a "job submission". The execution command takes a job description as a parameter.

A job description may be contained in a file, or specified in the command line of the execution command, or stored in an object. It contains the following information: the application description (at least a binary file), the parameters of the application, the end-user identity to check her rights to access some data and some resources, and the required qualities of execution (c.f. next section). For example, such job descriptions include the OGF recommended Job Submission Description Language (JSDL), or the job description part of the Globus Extended Resource Specification Language (xRSL), or the parameters of the mpirun command for MPI programs (which may also be stored in a so-called "application file"), or an object which implements the OGF recommended DRMAA job template interface, etc.

A job has a state; for example, it can be "queued", "running", "suspended", "failed", "terminated", etc. A state model describes how a job passes from one state to another. Each RMS comes with its own state model: their own states and own transitions. For example, the Basic Execution Service (BES) has its own state model, and the DRMAA interface provides another state model. At any time, an end-user may monitor the current state of her jobs.

An end-user may also control her jobs by sending them signals, to suspend them, or stop them, etc.

2.3.3 Specify some quality of services

For each execution of an application, the RMS may allow the end-user to specify some Qualities of Services (QoS) like execution time, low latency, reliability, security, compromise between price and performance, real-time constraints, etc. These QoS are non-functional properties on resource usage. Concerning reliability, the end-user may specify a resistance to a number of crashes during the execution, or using a checkpoint service to save an ongoing computation to continue the execution later with a different set of resources if possible. Another QoS could concern the cost of the execution of a job. On one side, by paying more, an end-user could get more resources and get them with a higher priority for her jobs. On the other side, an end-user may want to spare her money and wait longer for her jobs to complete. A RMS may also provide static QoS, which are decided at the

creation of a job and fixed for the whole execution of the job, but it may also provide dynamic QoS, which can be changed during the execution of a job.

All the QoS requirements may be gathered into a document called a Service Level Agreement (SLA) contract, and provided to the RMS when submitting a job.

3 Architectures of resource management systems

Most RMS designers make some recurrent choices regarding the architecture of RMS. RMS architectures are usually organized in layers: the lowest layer managing individual resources and the highest layer managing sets of resources. RMS may have different resource organizations, different resource interfaces, and provide different scheduling strategies.

3.1 Generic layered architecture

A widely accepted and promoted generic layered architecture for a RMS consists in five layers [11, 13]: the fabric layer, the connectivity layer, the resource layer, the collective layer and the application layer (c.f. Figure 1). Programs in each layer provide some common services that may be implemented using some services provided by programs of lower layers.

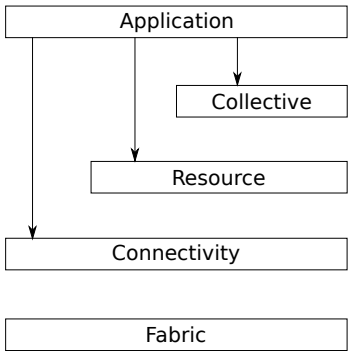

Figure 1: The layered architecture.

Fabric layer A RMS is based on available programs that manage each resource locally. These programs belong to the Fabric layer. Usually, a RMS may call query functions on each resource of the pool to get a description of each resource (their structure, state and capabilities). Depending on their type, resources may provide different capabilities. Computational resources must offer at least an interface to start programs and to monitor and control the execution of the resulting processes. Enquiry functions are needed for determining hardware and software characteristics as well as relevant state information such as current load and queue state. Storage resources must offer an interface to put and get files from them. Some management mechanisms are useful such as control to data transfers (space, disk bandwidth, network bandwidth). Enquiry functions on storage resource should return the available space and bandwidth utilization. Network resources can provide some control over the network transfers (like prioritization, reservation). Enquiry functions should provide the network characteristics and load.

Connectivity layer The Connectivity layer defines core communication protocols (e.g. TCP/IP, DNS, etc.) and authentication protocols (e.g. Kerberos, etc.) required for specific network transactions. Core communication protocols include standard communication mechanisms to transfer a message from one resource to another. Authentication protocols should provide single sign on: a user logs on just once and then has access to multiple resources without any further explicit user interaction. They should also allow the user to delegate her rights to a program such that this

program may use the resources that this user can access. Programs of the Connectivity layer interact with the Fabric layer by using the local network capabilities of each resource and by mapping to the local authentication mechanism of each resource.

Resource layer The Resource layer groups functions related to the individual management of a resource within the pool. Programs of this layer uses the Connectivity layer to access programs provided by the Fabric layer to define protocols for the secure negotiation, initiation, monitoring, control, accounting and payment of sharing operations on individual resources. Two primary classes of Resource layer protocols can be distinguished: information protocols and management protocols. Information protocols are used to obtain information about the structure and state of an individual resource. Management protocols are used to negotiate access to a resource by specifying, for example, resource requirements and the operations to be performed (process creation or data access).

Collective layer The Collective layer groups programs that deal with the coordination of multiple resources. These programs may offer various collective services such as directory services that allow their users to query for resources; resource discovery to update the directory services; co-allocation, scheduling and brokering services; monitoring services; grid-enabled programming framework (for example grid implementation of the Message Passing Interface); community authorization servers that enforce a global policy on resource access; community accounting and payment services; etc.

Application layer The Application layer comprises the user applications. They are constructed by calling upon services defined at any layer. "Applications" here may be, in practice, complex languages and frameworks (components, workflows, etc.) and feature much internal structure than displayed in this figure. For example, these frameworks may provide checkpointing, job management, distributed data archiver, etc.

3.2 Resource management system taxonomy

The taxonomy [17] classifies RMS by characterizing different attributes: their organization, their resource interface and their resource scheduling.

3.2.1 Machine organization

The organization of the machines in the pool of resources affects the communication patterns of the RMS and thus determines the scalability of the resultant architecture. A centralized organization is enough for small pool of resources, however centralized architectures do not scale to large pools.

In large scale pool, the architecture can use a flat, hierarchical or cell-based organization. In a flat organization all machines can directly communicate with each other without going through an intermediary. In a hierarchical organization machines in same level can directly communicate with the machines directly above them or below them, or peer to them in the hierarchy. In a cell structure, the machines within the cell communicate between themselves using flat organization. Designated machines within the cell function acts as boundary elements that are responsible for all communication outside the cell.

3.2.2 Resources interface

The resource interface is defined by the resource model, the resource namespace organization, the resource QoS definition, the resource information store organization, and the resource discovery mechanism.

The resource model determines how applications and the RMS describe and manage resources. In a schema-based approach the data that comprises a resource is described in a description language along with some integrity constraints. In some systems the schema language is integrated with a

query language. In an object model scheme approach the operations on the resources are defined as part of the resource model. In both approaches, the resource model can be fixed or extensible.

A RMS provides a global namespace for resources required by network applications. The organization of the resource namespace influences the design of the resource management protocols and affects the discovery methods. A relational namespace divides the resources into relations and uses concepts from relational databases to indicate relationships between tuples in different relations. A hierarchical namespace divides the resources into hierarchies. Relational namespace or hierarchical namespace are typically used with a schema-based resource model. A graph-based namespace uses nodes and pointers where the nodes may or may not be complex entities. Namespaces that are implemented using an object-oriented paradigm typically use graph namespaces with object as nodes and inter-object references being the pointers.

Individual resource must support a QoS interface to provide QoS globally. There are two parts to QoS, admission control and policing. Admission control determines if the requested level of service can be given by a resource, and policing ensures that a job does not violate its agreed upon level of service. A resource may not provide any QoS support, or just the admission control (soft QoS support), or both admission control and policing (hard QoS support).

In a RMS architecture, the resource description, resource status and resource reservation is managed by the resource information store organization. That store can be a network directory (e.g. X.500/LDAP) or a distributed objects directory. The important difference between distributed object and network directory approaches is that in network directories the schema and operations are separated with the operations defined externally to the data store schema. In an object oriented approach the schema defines the data and the operations.

Resource discovery and dissemination provide complementary functions. Discovery is initiated by a network application to find suitable resources within the pool. In a query-based resource discovery, the application sends queries to network directories. In a agent-based resource discovery, fragments of code are sent across the resources to collect resources. The major difference between a query-based approach and an agent-based approach is that agent-based systems allow the agent to control the query process and make resource discovery decisions based on its own internal logic rather than rely on an a fixed function query engine.

Resource dissemination is initiated by a resource trying to find a suitable application that can utilize it. It is categorized by the approach taken to updating the resource information. In a batch or periodic approach, resource information is batched up on each machine and then periodically disseminated through the pool. In an online or on demand approach information is disseminated from the originating machine immediately. In this case, the information is pushed to other machines in the pool.

3.2.3 Resource and job scheduling

The scheduling components of a RMS decide which resources are assigned to which jobs (and vice versa). This mapping of resources to jobs is a policy decision. The policy can be explicitly specified via an external rule base or a programmable interface. The policy can be implicitly implemented by choice of state estimation algorithms and rescheduling approach as well.

RMS make different choices concerning the organization of scheduling components, the degree of extensibility, rescheduling approaches and state estimation.

A RMS scheduler organization may be centralized, hierarchical or decentralized. In a centralized controller scheme there are one or more machines that provide the resource scheduling function for resource requestors and resource providers. All requests are routed to the resource scheduling machines that schedule the resource provider machines. Logically there is one central request and one job queue that is distributed among the designated resource schedulers. In a hierarchical controller approach, the resource schedulers are organized as a hierarchy. Resource schedulers in higher levels of the hierarchy schedule bigger units of resource providers, and lower level resource schedulers schedule smaller units of resource providers until individual resource providers are scheduled. Logically there is one request and one job queue per hierarchy level. In a fully decentralized approach there are no dedicated resource for scheduling: the resource requestors and resource providers directly determine

the allocation and scheduling of resources. Logically there are as many different requests and job queues as there are resource requestors and resource providers in the pool.

A resource state estimation service can be used to anticipate scheduling decisions. It can be non-predictive or predictive. Non-predictive state estimation uses only the current job and resource status information but do not take into account historical information. This approach uses either heuristics based on job and resource characteristics, or a probability distribution model based on an off-line statistical analysis of expected job characteristics. Predictive state estimation takes current and historical information such as previous runs of an application into account in order to estimate state. Predictive models use either heuristic, pricing model or machine learning approaches. In a heuristic approach, predefined rules are used to guide state estimation based on some expected behavior of applications. In a pricing model approach, resources are bought and sold using market dynamics that take into account resource availability and resource demand. In machine learning, online or off-line learning schemes are used to estimate the state using potentially unknown distributions.

The rescheduling characteristic of a RMS determines when the current schedule is re-examined and the job executions reordered. The jobs can be reordered to maximize resource utilization, job throughput, or other metrics depending on the scheduling policy. The rescheduling can be periodic or event-driven. Periodic or batch rescheduling approaches group resource requests and system events and process them at intervals. This interval may be periodic or may be triggered by certain system events. The key point is that rescheduling is done to batches instead of individual requests or events. Event driven online rescheduling performs rescheduling as soon the RMS receives the resource request or system event.

4 Resource management in some existing distributed systems

In this section, we review a few RMS. We first focus on the three RMS developed by partners of the COOP project: XtreemOS, DIET and Padico. Then we review some other RMS for clusters, grids and clouds systems.

4.1 Resource management in XtreemOS

XtreemOS [6] is a grid operating system based on Linux. The main particularity of XtreemOS is that it provides for Grids what a traditional operating system offers for a single computer: hardware transparency and secure resource sharing between different users. It thus simplifies the work of users by giving them the illusion of using a traditional computer while removing the burden of complex resource management issues of a typical Grid environment. When a user runs an application on XtreemOS, the operating system automatically finds all resources necessary for the execution, configures user's credentials on the selected resources and starts the application.

XtreemOS is developed by the XtreemOS Project that gathers 19 partners from Europe and China, from academic and industrial backgrounds. The XtreemOS project is a silver organizational member of the OGF community, and is funded by the European Commission through the Information Society Technology.

4.1.1 XtreemOS fundamental properties

XtreemOS is based on an underlying operating system which is extended for enabling and facilitating distributed computing. The XtreemOS operating system is based on the Linux traditional general-purpose OS, extended as needed to support Virtual Organizations (VOs), and to provide appropriate interfaces to grid OS services.

XtreemOS is an operating system able to execute any kind of application, including legacy applications since it provides a POSIX interface for grid-unaware applications. XtreemOS aims at providing transparency for users and application developers, scalability, manageability, security,

trust. Moreover, XtreemOS can also run grid-aware application using the Simple API for Grid Applications (SAGA) since it provides an implementation of SAGA called XOSAGA.

XtreemOS targets dynamic grids with high node churn. Grid nodes may join or leave the system at anytime based on decisions of their administrator or user. This may happen with a prior announcement, via a sign on/off, or without, for example in the event of a crash. A given grid node may be temporarily disconnected as network failures may occur at any time.

4.1.2 XtreemOS's architecture

Figure 2 depicts the XtreemOS software architecture at a high level of abstraction. Internally, XtreemOS is composed of two parts: the XtreemOS foundation called XtreemOS-F and the high-level services called XtreemOS-G. XtreemOS-F is a modified Linux kernel embedding VO support mechanisms and providing kernel level process checkpoint and restart functionalities. XtreemOS-G is implemented on top of XtreemOS-F at user level. It consists of several distributed services to deal with resource and application management in VOs.

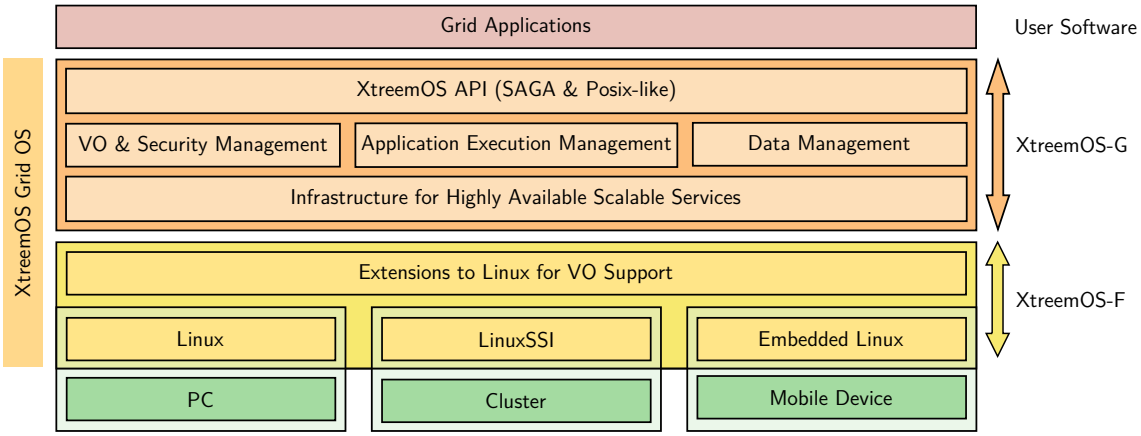

Figure 2: XtreemOS's architecture.

VO management and security XtreemOS targets scalable and flexible management of dynamic VOs. In XtreemOS, a VO-frame spans multiple administrative domains on different sites and is comprised of the set of resources (computers, VO-frame members (human beings)) shared by the participating organizations. A VO-frame member can create a VO, for which she becomes the owner. Any VO-frame member may request her registration in a given VO, subject to the VO owner approval. Resources can be registered as well in VOs. The VO owner defines policies, stating permission and usage rules for VO resources. The VO-frame administrator also defines policies regulating what a VO-frame member can do (for example, permission to create a VO). Resource owners in the different administrative domains may also define local policies for resource usage. VO-frame, VO and local policies are enforced by the XtreemOS system. A VO-frame member or a resource may belong to multiple VOs and to multiple VO-frames. XtreemOS implements a set of security and VO management services to manage authentication, authorization, single-sign-on and accounting. XtreemOS relies on node level system mechanisms in order to dynamically map global identities of VO members into local Linux accounts.

Data management with XtreemFS XtreemFS is the XtreemOS distributed file system. It allows the federation of multiple data stores located in different administrative domains and provides secure access to stored files to VO members, whatever their location. It provides the POSIX file system interface and consistent data sharing. Moreover, XtreemFS allows automatic data replication management for data access efficiency and availability. Files are stored in XtreemOS

volumes. Each VO user gets an XtreemFS home volume, which is automatically mounted by the XtreemOS system as needed.

Application execution management The Application Execution Management (AEM) service of XtreemOS is in charge on the one hand of discovering, selecting and allocating resources for job execution and on the other hand of starting, controlling and monitoring jobs. In XtreemOS, jobs are self-scheduled, meaning that there is no assumption that a batch scheduler exists on grid nodes. Each computational resource is managed by an independent scheduler; and the job dispatcher and local schedulers interact to find the best scheduling. In AEM, when a user launches a job described by a standard JSDL 1.0 file, the AEM service dynamically discovers resources based on an overlay linking all VO resources. Resources may dynamically join and leave the overlay. The AEM provides some advanced job control features that can be used by workflow managers. One of these features is the concept of job dependencies that can relate two or more jobs with a tag or label that may be used by other tools. The AEM also provides a checkpointing service that is able to stop, restart and migrate jobs to another resource. It is important to allow features as fault tolerance and load balancing. If a resource fails the user/system could migrate and restart the job in another resource without losing the work done. The XATI is the AEM programming interface used to communicate with services from the AEM. There is a Java and C version that can either be used by applications directly, or via XOSAGA (which will in turn use XATI to communicate with the AEM services). For example, an application may call AEM services during its execution through the XATI to ask for more resources or free some resources.

Infrastructure The infrastructure for highly available services provides a set of building blocks (such as overlays, publish/subscribe, replication and distributed directory services) that allows the upper services (AEM, XtreemFS, VO and security management services) to cope with the large scale and dynamicity of grids. In particular, the infrastructure provides the Resource Selection Service (RSS) and the Application Directory Service (ADS) which form together the Scalable Resource Discovery System (SRDS). The RSS takes care of performing a preliminary selection of nodes to allocate to an application, according to range queries upon static attributes. It exploits a fully decentralized approach, based on an overlay network which is built and maintained through epidemic protocols. This allows to scale up to hundred thousands, if not millions, of nodes and to be extremely resilient to churn and catastrophic failures. The ADS handles the second level of resource discovery, answering queries expressed as predicates over the dynamic attributes of the resources. ADS creates an application-specific "directory service" using the NodeIDs received by the RSS, related to the resources involved in the application execution. To provide scalability and reliability, DHT techniques and their extensions to dynamic and complex queries are used.

XtreemOS-F foundation layer There are three flavors of the XtreemOS-F system, one for each kind of grid node supported by XtreemOS: PC, cluster and mobile device. LinuxSSI, the cluster flavor of XtreemOS-F, leverages the Kerrighed Linux-based single system image (SSI). A cluster appears as a single powerful node in the grid, its individual nodes being invisible at grid level. The XtreemOS mobile flavor, called XtreemOS-MD, provides a set of open source modules, easily integrable in any mobile Linux OS, to enable mobile devices (e.g. PDAs, smartphones, etc.) for grid operation in VOs, in an efficient and transparent way.

Resource discovery and allocation with XtreemOS To execute an application in XtreemOS, a user logs into a VO she belongs to and gets XtreemOS credentials from the Credential Distribution Authority service. These credentials contain her public key and VO attributes and are used by other XtreemOS services (AEM, XtreemFS, etc.) to authenticate the user and check her authorizations. The AEM service launches a resource discovery over the VO-frame overlay. Suitable resources registered in the user VO reply to this discovery. Then AEM selects and allocates resources for the application execution, that it monitors until termination. Access to files stored in XtreemOS volumes is granted based on VO user credentials.

4.2 Resource management in DIET

DIET (Distributed Interactive Engineering Toolbox) is a scalable distributed middleware for accessing transparently and efficiently heterogeneous and highly distributed machines.

4.2.1 The DIET architecture

The DIET architecture is structured hierarchically for improved scalability. Such an architecture is flexible and can be adapted to diverse environments, including arbitrary heterogeneous computing platforms. The DIET toolkit [4, 1] is implemented in CORBA and thus benefits from the many standardized, stable services provided by freely-available and high performance CORBA implementations. CORBA systems provide a remote method invocation facility with a high level of transparency. This transparency should not substantially affect the performance, as the communication layers in most CORBA implementations is highly optimized.

The DIET framework comprises several components. A Client is an application that uses the DIET infrastructure to solve problems using a remote procedure call (RPC) approach. Clients access DIET via various interfaces: web portals programmatically using published C or C++ APIs. A SED, or server daemon, acts as the service provider, exporting functionality via a standardized computational service interface; a single SED can offer any number of computational services. A SED can also serve as the interface and execution mechanism for either a stand-alone interactive machine or a parallel supercomputer, by interfacing with its batch scheduling facility. Agents are the third component of the DIET architecture. They facilitate the service location and invocation interactions of clients and SEDs. Collectively, a hierarchy of agents provides higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single Master Agent (MA) and several Local Agents (LA). Figure 3 shows an example of a DIET hierarchy.

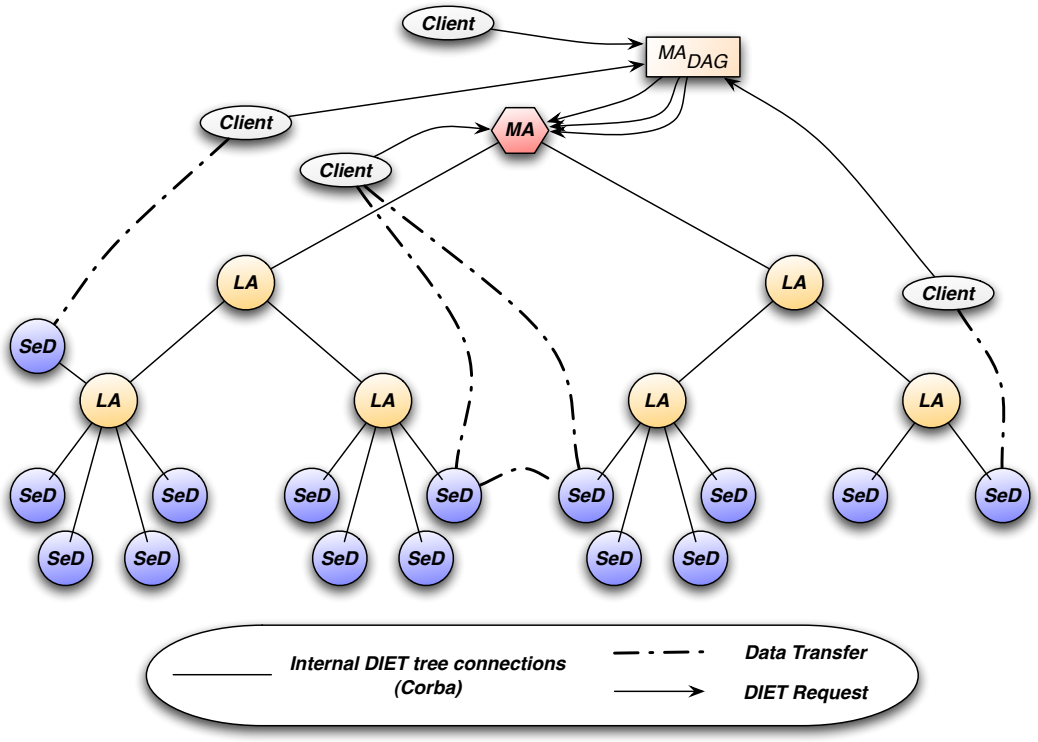

Figure 3: DIET hierarchical organization. Plug-in scheduler are available in each MA and LA. Clients can submit requests directly to the MA or submit workflows to the MA_{DAG}.

The Master Agent of a DIET hierarchy serves as the distinguished entry point from which the services contained within the hierarchy can be logically accessed. Clients identify the DIET hierarchy using a standard CORBA naming service. Clients submit requests — composed of the name of the specific computational service they require and the necessary arguments for that service — to the MA. The MA then forwards the request to its children, who subsequently forward the request to their children, such that the request is eventually received by all SEDs in the hierarchy. SEDs then evaluate their own capacity to perform the requested service; capacity can be measured in a variety of ways including an application-specific performance prediction, general server load, or local availability of datasets specifically needed by the application. SEDs forward this capacity information back up the agent hierarchy. Based on the capacities of individual SEDs to service the request at hand, agents at each level of the hierarchy reduce the set of server responses to a manageable list of server choices with the greatest potential. The server choice can be made specific to any kind of application using plug-in schedulers at each level of the hierarchy.

4.2.2 Data management

Data management is handled by DAGDA [5] components (Data Arrangement for Grid and Distributed Application); it allows data explicit or implicit replications and advanced data management on a grid such as data backup and restoration, persistence, data replacement algorithm. A DAGDA component is attached to each DIET element and follows the same hierarchical distribution. However, whereas DIET elements can only communicate following the hierarchy order (those communications appear when searching a service, and responding to a request), DAGDA components will use the tree to find data, but once the data is found, direct communications will be made between the owner of the data and the one which requested it. A DAGDA component associates an ID to each stored data, manages the transfers by choosing the "best" data source according to statistics about the previous transfers time and performs data research among the hierarchy. Just like DIET, DAGDA uses the CORBA interface for inter-nodes communications.

4.2.3 Workflows

A large number of scientific applications are represented by graphs of tasks which are connected based on their control and data dependencies. The most commonly used model is the graph and especially the Directed Acyclic Graph (DAG). DIET introduces a new kind of agent: the MA_{DAG}. This agent is connected to the MA as can be seen in Figure 3. Instead of submitting requests directly to the MA, a client can submit a workflow to the MA_{DAG}, i.e., an XML file containing the whole workflow description. The MA_{DAG} will then take care of the workflow execution, and it will schedule it along with all the other workflows present in the system, hence the system can manage multiple workflows concurrently. Thus, a client only needs to describe the workflow in an XML format, then feeds in the input data, and finally retrieves the output data when the workflow has finished its execution.

4.2.4 Web Portal

In order to ease job submissions, a web portal has been developed[2]. It hides the complexity of writing workflow XML files. The user describes astrophysical parameters, and provides the filters list file, the observation cones description file, and a tarball containing one or many treefiles, then clicks on submit. The system takes care of the communications with DIET: it creates the corresponding workflow descriptions, submits them to DIET, retrieves the final results and makes them available via a web page. In order to explore the parameter space, the submission web page allows for astrophysical and cone parameters to define three values: the minimum and maximum values, and the increase step that needs to be applied between these two values. Thus, with the help of a single web page, the user is able to submit a lot of workflows.

[2]Portal for cosmological simulations submission: http://graal.ens-lyon.fr:5544/Cosmo/
Description of the available features: http://graal.ens-lyon.fr/DIET/dietwebboard.html

In case a submission fails, the job is submitted again after a fixed period, this until the job finishes properly, or is canceled by the user. Once finished, a tarball file containing all result files can be downloaded.

Access to this system is protected by a login/password authentication, so as to restrict access to applications and produced data. This system also provides independent parameter sweep jobs submissions for both applications.

4.3 Resource management in Padico

Padico is a Software Environment for Computational Grids. It is designed for high performance parallel computing, distributed computing, and software components. In particular, it targets complex applications such as code coupling applications. It contains two particular entities for COOP with respect to resource management:

PadicoTM — the Padico runtime layer to manage multi-threading and networking.

Adage — the tool for automatic deployment of applications on distributed systems.

4.3.1 Resource services

PadicoTM is the Padico runtime layer. PadicoTM provides the user with network communication and multi-threading. Two levels of services are available:

- standard APIs such as Posix sockets, MPI, CORBA, and pthreads;
- specific APIs for networking and multi-threading.

Standard APIs make legacy codes run seamlessly atop PadicoTM with no change in the code. They may also be used by new applications whose developers do not want to be tied to a specific framework. On the other hand, some advanced features may not be exposed through existing standard API and require thus a specific API.

Adage stands for Automatic Deployment of Applications in a Grid Environment. It is a prototype middleware designed to automatically launch distributed (CCM, JXTA, DIET, etc.) or parallel (MPI) applications on the resources of a computational grid.

4.3.2 Resource management

Component-based communication framework. The most challenging part to manage communication on distributed systems such as grids for example is the heterogeneity of the resources and the variety of applications — and thus their requirements for a communication sub-system. The networks are ranged from high-performance networks in cluster to wide-area networks between sites. Not only their properties are different, but their protocol, communication methods and programming interface are different. Moreover, the requirements for the communication sub-system depends on the application; the performance versus security tradeoff largely depends on the nature of the application and should not be hard-coded in a communication framework.

Such a needed flexibility may hardly be reached by the usual two-layer portability model based on an abstraction layer and drivers for each supported resource. Considering the variety of cases to deal with, it would be highly welcome to have communication methods be *assembled* by the users depending on the application and the kind of networks and protocols involved. For example, a user may want to add compression or encryption on the fly to any communication method; another user may want no encryption at all to get the best achievable performance with non-critical data.

To reach such a flexibility, [9] proposes to manage communications with a freely and dynamically assembled protocol stack made of several simple *building blocks*. Such a technique is nowadays commonly used in all fields of software development and is known under the name of *software component*. PadicoTM [9] is a component-based communication framework for grids implementing this proposition. It is designed to be as flexible as possible. It supports a wide range of networks and communication methods, from high-performance networks (Infiniband, Myrinet, Quadrics QsNet) to

wide area networks (routing, compression, authentication, proxying, tunneling). Moreover, several middleware systems — MPI, various CORBA implementations, Java RMI, SOAP implementations, HLA, ICE, DSM systems, JXTA — have been ported on top of PadicoTM thanks to its flexible personality layer that enables a seamless integration of existing code.

PadicoTM is based on a three-layer approach [10]: the lowest layer does multiplexing and arbitration between concurrent accesses to a given network, and between accesses to different networks (e.g. TCP/Ethernet and Myrinet) on the same machine; the middle layer is the abstraction layer, based on dynamically assembled components; the higher layer, or personality layer, adapts the API to the expectations of applications.

Mixing threads and communications. The low layer of PadicoTM relies on the PM2 software suite, composed of the NewMadeleine communication scheduler, the Marcel multi-threading library, and an I/O event manager called PIOMan.

NewMadeleine has a 3-layer architecture with its activity driven by the underlying NICs, in contrast with the usual behavior of most communication libraries driven by the send/recv from the application. The core layer applies dynamic scheduling optimizations on multiple communication flows such as packet reordering, coalescing, multirail distribution, etc. NewMadeleine implements both a specific API and a MPI interface called Mad-MPI. The application provides messages to the collect layer. When a NIC becomes idle, the optimization layer is invoked so as to compute the best message arrangement (by aggregating messages, splitting messages, etc.) and to submit the next packet to send to the transfer layer that uses the NIC drivers to transmit the arranged packet.

The Marcel multi-threading library features a two-level thread scheduler that achieves the performance of a user-level thread package while being able to exploit SMP machines. It is highly tunable and includes hooks usable for asynchronous communication progression. The communication engine of the PM2 software suite is called PIOMan [26]. It handles polling in behalf of the communication library and works closely with the thread scheduler. It is able to perform polling inside Marcel hooks (when a core is idle, on context switch, on timer interrupts) or within tasklets in order to exploit any core of the machine.

ADAGE ADAGE (*Automatic Deployment of Applications in a Grid Environment*) [8] is a research prototype that aims at studying the deployment issues related to multi-middleware applications. Its original contribution is to use a *generic* application description model [18] (*GADe*) to transparently handle various middleware systems.

With respect to application submission, ADAGE requires an application description, which is specific to a programming model, a reference to a resource information service (MDS2, or an XML file), and a control parameter file. The application description is internally translated into a generic description, so as to support multi-middleware applications. The control parameter file allows a user to express constraints on the placement policy, which is specific to an execution. For example, a constraint may specify the latency and the bandwidth between a computational component and a visualization component.

The support of multi-middleware applications is based on a plug-in mechanism. The plug-in is involved in the conversion from the specific to the generic application description, but also during the execution phase so as to deal with specific middleware configuration actions.

ADAGE currently deploys static applications and also pseudo-dynamic applications by re-deploying new application elements. It supports standard programming models like MPI (MPICH1, MPICH2 and OpenMPI), CCM, DIET, JXTA, P2P, and Gfarm.

4.4 Resource management in some batch systems

4.4.1 OAR

OAR [2] is an open-source batch scheduler that can be used to manage multiple clusters. It uses the Conservative Back-Filling scheduling algorithm (CBF) [19]. Its main strengths lie in the wide range of job types it supports and the flexibility of specifying resource requirements.

Among others, OAR supports the following types of jobs. *Batch jobs* are queued until the requested resources become available; upon start a command is launched on the target resources. *Interactive jobs* automatically spawns a user shell on the first allocated host. *Advance reservations* allow a user to reserve resources with a fixed start time. *Deploy* is a job type designed to install a custom, bare-metal image on the target resources. *Best-effort jobs* are low-priority batch jobs. They are automatically killed when another job needs resources.

For submitting jobs, one has to write a resource specification (which includes a wall-time), whose ultimate purpose is to select computation cores. OAR has a hierarchical representation of the resources: core, CPU, node, switch. Cores can be assigned an arbitrary number of attributes, such as the amount of RAM, scratch space, the cluster it takes part in etc. The user may enter complex SQL-like queries, which select both the number and the attributes of the requested resources. The user may submit multiple resource specifications, in which case OAR chooses the one that ends earliest, according to the provided wall-times. This allows the user to improve the back-filling capabilities of her *moldable* application, potentially improving response time and resource usage.

Due to its scalability, flexibility and support for deploy jobs, OAR is used to manage the resources of the French National experimental platform Grid'5000 [3].

4.4.2 SLURM

Simple Linux Utility for Resource Management (SLURM)[3] is a popular, open-source, batch scheduler. It distinguishes itself by its jobs pre-emption capabilities, which allow it to implement a wide range of scheduling algorithms.

Besides the classical first-come first-serve [22] and conservative back-filling [19], SLURM implements other scheduling algorithms which require job pre-emption. SLURM has the build-in ability to suspend and resume jobs, provided enough swap and disk space is available to accommodate all jobs allocated to a node, either running or suspended. For example, it is able to suspend low-priority jobs, to allow a high-priority job to launch, or it can do time-sharing of resources and gang-schedule multiple jobs.

Job resource specifications are relatively simple, allowing the user to specify them in terms of a wall-time and minimum/maximum nodes, CPUs, cores, RAM.

SLURM is very scalable and is in production use on multiple super-computing sites.

4.4.3 TORQUE

TORQUE[4] (previously known as PBS) is an open-source, batch scheduler with commercial support. In contrast to traditional batch schedulers, which only support rigid jobs (i.e., jobs that require a unique fix number of nodes and walltime), TORQUE supports two additional types of job: moldable and dynamic[5].

Moldable jobs (incorrectly called "malleable" in the TORQUE documentation) allow the user to specify a *task request list* which consists in a sequence of number of nodes and wall-times. The user can for example specify that her job runs on 4 nodes for 3 hours, 8 nodes for 1.5 hours, or 16 nodes for 1 hour. TORQUE selects the configuration which both minimizes job completion time and optimizes resource utilisation.

Dynamic jobs are jobs to which the RMS may add or remove nodes, based on a user-desired target response time or target load (a metric declared by the job). Such a job must provide TORQUE with two scripts: WorkloadQuery and JobModify. WorkloadQuery allows TORQUE to query the job about its current load or estimated response time. JobModify is called to inform the job that the nodes allocated to it have changed. At each scheduling iteration, TORQUE allocates or deallocates nodes, so that the estimated response time or load meets the user-specified targets. Dynamic jobs can be used to run transactional workloads (such as web servers) on the same infrastructure batch workloads run on, thus simplifying management and reducing resource under-utilisation.

[3] https://computing.llnl.gov/linux/slurm/slurm.html
[4] http://www.clusterresources.com/products/torque-resource-manager.php
[5] http://www.clusterresources.com/products/moab/21.3dynamicjobs.shtml

Due to its scalability and fault-tolerance, TORQUE is used to manage large infrastructures, such as TeraGrid[6].

4.5 Resource management in some grid systems

4.5.1 Condor

Condor [25] is a middleware designed for high-throughput computing, allowing users to access a tremendous amount of resources. The Condor project started in 1984 to allow researchers to access idle cycles on an institution's workstations, also known as cycle-scavenging, desktop computing or desktop grid. Throughout the years, as cheap, commodity hardware was regrouped into clusters and later grids, the focus of Condor shifted to exploit these resources, while at the same time offering the same ease of access and transparency.

In its early days, Condor focused on providing easy access to the idle resources of workstations, having two main goals. From the resource owner's perspective, Condor should not interfere with her work, i.e. it should not compromise the security of her system, slow down her machine or force her to leave her machine running. From the scientist's perspective, applications should run transparently (without requiring modifications) and should finish successfully, even if resources become unavailable.

To achieve these goals, Condor proposed an architecture in which three types of actors participate: *resource* (R), *match-maker* (M) and *agent* (A). Executing a job occurs in three steps: first, both the resource daemon (running on the idle workstation) and the agent advertise themselves to the match-maker, using a language called ClassAds. Second, the match-maker informs both parties that they potentially match. Third, the agent contacts the resource directly to attempt to execute the job.

Jobs are executed inside a sandbox. Besides protecting the owner's workstation, this allow Condor to *checkpoint* and *migrate* jobs for reliable execution and do *system calls translation*, which allows transparent access to remote files.

As the world shifted to the paradigm of grid computing, the Condor project developed Condor-G, which interfaces Globus' GRAM. To access resources behind batch scheduler, whose queues behave unpredictably, Condor-G developed a *gliding in* mechanism, also known as pilot job [14]. The idea is that, instead of submitting jobs directly to the batch scheduler, Condor-G uses the batch scheduler to launch a resource daemon on the target machines and then uses the usual Condor mechanism to launch jobs.

Condor proved to be a very good solution for high-throughput computing. It is actively maintained and in use by a large community.

4.5.2 Globus Toolkit

The Globus Toolkit [12] is a set of services to build grids and grid applications. It includes software services and libraries for distributed security, resource management, monitoring and discovery, and data management. The GT4 version includes components for building systems that follow the Open Grid Services Architecture (OGSA) framework defined by the Global Grid Forum (GGF). GT4's components and software development tools also comply with the Web Services Resource Framework (WSRF), a set of standard web services.

The toolkit components that are most relevant to OGSA are the Grid Resource Allocation and Management (GRAM) protocol, the Meta Directory Service (MDS-2), and the Grid Security Infrastructure (GSI). The GRAM protocol provides for the reliable, secure remote creation and management of arbitrary computations. GSI mechanisms are used for authentication, authorization, and credential delegation to remote computations. A two-phase commit protocol is used for reliable invocation, based on techniques used in the Condor system. Service creation is handled by a small, trusted "gatekeeper" process, while a GRAM reporter monitors and publishes information about the identity and state of local computations. MDS-2 provides a uniform LDAP-based framework

[6] https://www.teragrid.org/

for discovering and accessing system configuration and status information such as compute server configuration, network status, or the locations of replicated datasets. MDS-2 uses a soft-state protocol, the Grid Notification Protocol, for lifetime management of published information. The public-key-based Grid Security Infrastructure (GSI) protocol provides single sign-on authentication, communication protection, and some initial support for restricted delegation. Single sign-on allows a user to authenticate once and thus create a proxy credential that a program can use to authenticate with any remote service on the user's behalf. Delegation allows for the creation and communication to a remote service of delegated proxy credentials that the remote service can use to act on the user's behalf, perhaps with various restrictions; this capability is important for nested operations.

The Global Access to Secondary Storage (GASS) of Globus helps to program data movement and management. GASS simplifies the porting and running of applications that use file I/O to the Globus environment. Libraries and utilities are provided to eliminate the need to manually login to sites and FTP files, or install a distributed file system. The APIs are designed to allow reuse of programs that use Unix or standard C I/O with little or no modification. Globus includes also an implementation of the GridFTP protocol which provides a reliable and high performance file transfer for grid computing applications to transmit very large files. GridFTP the following alterations to normal FTP: security with GSI, third party transfers, parallel and striped transfer, partial file transfer, fault tolerance and restart and automatic TCP optimization for large files and large groups of files.

Globus offers QoS in the form of resource reservation. It allows application level schedulers such as the Nimrod/G resource broker to extend scheduling capabilities. The resource brokers can use heuristics for state estimation while performing scheduling or re-scheduling whenever the status of the Grid changes. Globus is a Grid toolkit and thus does not supply scheduling policies, instead it allows third party resource brokers. Globus services have been used in developing several resource brokers (global schedulers) such as Nimrod/G, AppLeS and Condor/G.

4.5.3 Legion

Legion [16] is an object-based wide-area operating system. Users working on their home machines see the illusion of a single computer, with access to all kinds of data and physical resources, such as digital libraries, physical simulations, cameras, linear accelerators, and video streams. Groups of users can construct shared virtual work spaces, to collaborate research and exchange information. This abstraction springs from Legion's transparent scheduling, data management, fault tolerance, site autonomy, and a wide range of security options. Legion executes as middleware on top of individual resource operating systems but provides a uniform API and object space to users and to developers. Legion provides a large number of key requirements: security, a global name space, programming ease, interactivity, fault tolerance, persistence, dynamicity, scalability, and site autonomy. Being run as a middleware, Legion can manage heterogeneous resources running different operating systems. But, in return, Legion provides a specific API to users (in order to manage the global user namespace) and to developers. A Legion domain is autonomous and manages its local users and resources. But multiple domains can be combined in order to form larger systems: objects created in one domain can communicate with and use the services of other objects in connected domains.

4.5.4 Vigne

Vigne [21] is a grid middleware that abstracts for the user the distributed physical resources. The main aim of Vigne is to provide scalable and reliable resource and job services. It is composed of a set of high-level services like application management, application monitoring, resource discovery, persistent data management, volatile data management, and low-level services like system membership. The application management service controls application execution. The application monitoring service offers information about the execution of applications. The volatile data management service offers distributed shared memory and the persistent data management service is used for storing and transferring files. The resource discovery service is in charge of finding a set of resources corresponding to the job specification. It uses an unstructured peer-to-peer overlay for distributed

attribute searching. System membership service connects the nodes of Vigne through a structured peer-to-peer overlay and is the base for the other services.

Each application is run under the supervision of the application management service. The application management service is a distributed service, with one instance per application. An instance of the service is called an application manager. The application manager has the task of running the application and ensuring that it terminates correctly despite nodes failing. For this, it interacts with other components like the client, the resource discovery service and the resource on which the application tasks are run. The application managers are distributed over the grid nodes using the structured peer-to-peer overlay. This provides fault tolerant message routing between applications managers and clients or grid applications. The structured overlay is based on Pastry and uses Bamboo's maintenance algorithms.

To launch an application, a client contacts one node of Vigne and sends it the job specification. The specification is described in a JSDL (Job Submission Description Language) file and includes the architecture requirements, number of nodes, memory, scheduler, etc. When the application is launched, a random key is associated to it, and an application manager is created for this application on the node whose id is the closest to the key at that moment. Afterward, a client can control the application by sending commands to the application manager, through the peer-to-peer overlay. In order to run an application, an application manager has to execute the following steps. First, it needs to select the nodes on which the application will run. For doing so, the application manager asks the resource discovery service to obtain the necessary resources. Then, if any input files are needed, the application manager asks the file transfer service to transfer them to the nodes on which the application will run. Finally, the tasks are started on the selected nodes and are monitored during their running to ensure their successful execution. If the task fails then the application manager restarts it from the beginning, or from a checkpoint if some checkpointing mechanisms are provided with the application. At the end the application manager cleans the task files, gets the output files and frees the allocated resources.

4.5.5 KOALA

KOALA [20] is a grid scheduler developed at the University of Delft in order to support co-allocation and advanced job placement policies in multi-cluster systems. For easing the task of launching applications on grid, *runners* have been developed.

Co-allocation is the simultaneous allocation of resources in multiple clusters. This is required for applications whose resource requirements are so large, that no single cluster can satisfy them. Co-allocation in grid systems is difficult, as the underlying resources are managed by uncoordinated local resource managers (LRMSs), whose job queues are not synchronised. In the presence of advance reservations, a grid scheduler could ask all concerned LRMSs for the list of free time slots and find one that fits the application. Unfortunately, most of the currently used LRMSs do not support advance reservations, or advance reservation require special privileges.

KOALA allows the user to specify multiple job components using an extension to Globus' Resource Specification Language (RSL), which it co-allocates even in the absence of advance reservations. How the job is split into job components and how components are associated to clusters can either be specified by the user or done by KOALA.

Malleability support has been added to KOALA, allowing to run jobs which can adapt to an increasing / decreasing number of hosts, as driven by the RMS. KOALA includes several malleable policies and automatically sends grow / shrink to the runners depending on system load. Malleable support has been proposed as a container to running low-priority, high-throughput, parameter-sweep applications, similarly to Condor's gliding-in.

KOALA is operational on the DAS-3 system, the Dutch national research grid.

4.6 Resource management in some cloud systems

Cloud computing is a paradigm shift in which computing resources are consumed from 3rd party providers, outside the direct control of the consuming organization. Cloud computing appeared

due to the need of businesses to diminish management costs of in-house computing resources and simplify planning for peak periods. Cloud providers aid by allowing dynamic provisioning of resources at various levels of abstraction (from lowest to highest):

Infrastructure as a Service (IaaS) provides hardware resources, such as virtual machines, storage and network. Popular examples include Amazon's EC2 and S3, as well as Rackspace Cloud.

Platform as a Service (PaaS) provides a platforms to develop, manage and host applications. In contrast to IaaS, it offers higher-level abstractions, such as Content Delivery Networks (CDN), Key-Value Databases and frameworks to quickly develop (usually) web-based applications. Popular examples include Google App Engine and Microsoft's Windows Azure Platform.

Software as a Service (SaaS) offers ready-to-use software to the end-user. Popular examples include GMail and Salesforce.com.

Higher abstraction levels take more management burden from the user, at the expense of less control. Since the scope of the COOP project is to improve collaboration between resource management and programming model (which is hidden from the user in PaaS and SaaS), this deliverable shall focus on IaaS.

While deviating from the most common cloud definitions, the term *private cloud* has been coined to denote a data-center which is managed using cloud-technologies, as opposed to *public clouds* which are outside the control of the user. Private clouds allow a company to address issues such as privacy, while at the same time being prepared for a transition to public clouds. *Hybrid cloud* is employed when private clouds and public clouds are combined, so as to combine the advantages of both.

4.6.1 Amazon Elastic Compute Cloud (EC2)

EC2 is Amazon's IaaS service which allows a user to reserve virtual machines (called *instances* in EC2 terminology) and deploy custom images on them. Currently the user can select among 9 instance types in 4 locations with varying hourly prices[7].

Instances can be reserved by interacting with a Web Service. Due to its popularity, the Amazon EC2 WSDL has become a de-facto standard in IaaS. Its implementation in popular open-source middle-wares (such as Eucalyptus[8] and Nimbus[9]) allows for easy creation of hybrid clouds.

5 Conclusion

Managing resources is a complex and difficult task, in particular with respect to scalability, heterogeneity, and security. Traditionally, each kind of architecture has its own kind of RMS: operating systems for single machines, batches for homogeneous sets of machines (clusters/supercomputers), grid systems for multiple administrative domains, desktop grids for heterogeneous and volunteer machines and recently cloud API for IaaS. Each of these infrastructures has particularities that motivate this plethora of systems. However, it turns out to be quite difficult for an application to be able to efficiently run on several of these systems. Either the user has to have a good knowledge of the resources so as to specifically configure its application, or the system keeps the control of the application and thus forces its behavior. For example, most grid systems only supports JSDL-like applications that are not able to efficiently support applications with complex structure like code coupling.

This report has presented various RMS developed by the partners of COOP: XtreemOS, DIET and PadicoTM/Adage. They represent the targeted RMS to study how to efficiently run applications with complex structures programmed with advanced programming models.

[7] http://aws.amazon.com/ec2/
[8] http://www.eucalyptus.com/
[9] http://www.nimbusproject.org/

References

[1] Abelkader Amar, Raphaël Bolze, Yves Caniou, Eddy Caron, Benjamin Depardon, Jean-Sébastien Gay, Gaël Le Mahec, and David Loureiro. Tunable scheduling in a GridRPC framework. *Concurrency and Computation: Practice and Experience*, 20(9):1051–1069, 2008.

[2] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. *CoRR*, abs/cs/0506006, 2005.

[3] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, pages 99–106, Seattle, USA, November 2005. IEEE/ACM.

[4] Eddy Caron and Frédéric Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[5] Eddy Caron, Frédéric Desprez, and Gaël Le Mahec. Dagda : An advanced data manager for the diet middleware. In *AHEMA 2008: Proceedings of the Advances in High-Performance E-Science Middleware and Applications Workshop*, Indianapolis, Indiana USA, December 2008.

[6] Toni Cortes, Carsten Franke, Yvon Jégou, Thilo Kielmann, Brian Matthews Domenico Laforenza, Christine Morin, Luis Pablo Prieto, and Alexander Reinefeld. XtreemOS: a vision for a grid operating system. Technical Report TR-4, European Integrated Project, 2008.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Alexandre Denis, Sébastien Lacour, Christian Pérez, Thierry Priol, and André Ribes. Programming the grid with components: models and runtime issues. In Beniamino Di Martino, Jack Dongarra, Adolfy Hoisie, Laurence T. Yang, and Hans Zima, editors, *Engineering The Grid: Status and Perspective*. American Scientific Publishers, 01 2006.

[9] Alexandre Denis, Christian Pérez, and Thierry Priol. PadicoTM: An open integration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 19(4):575–585, May 2003.

[10] Alexandre Denis, Christian Pérez, and Thierry Priol. Network communications in grid computing: At a crossroads between parallel and distributed worlds. In *18th International Parallel and Distributed Processing Symposium (IPDPS2004)*, page 95a, Santa Fe, New Mexico, April 2004. IEEE Computer Society.

[11] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a new Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[12] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.

[13] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3):2001, 2001.

[14] G. Graciani Diaz, A. Tsaregorodtsev, and A. Casajus Ramo. Pilot Framework and the DIRAC WMS. In *CHEP'09*, Prague Tchèque, République, 2009.

[15] Andrew Grimshaw. What is a Grid? *Grid Today*, 1(26), 2002.

[16] Andrew S. Grimshaw, Wm. A. Wulf, and CORPORATE The Legion Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[17] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems. *Software Practice and Experience*, 32:135–164, 2002.

[18] Sébastien Lacour, Christian Pérez, and Thierry Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, Seattle, WA, USA, November 2005. Springer-Verlag.

[19] D. Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995.

[20] Hashim Mohamed and Dick Epema. Koala: a co-allocating grid scheduler. *Concurrency and Computation: Practice and Experience*, 20(16):1851–1876, 2008.

[21] Louis Rilling. Vigne: Towards a self-healing grid operating system. In *Proceedings of Euro-Par 2006*, volume 4128 of *Lecture Notes in Computer Science*, pages 437–447, Dresden, Germany, August 2006. Springer.

[22] U. Schwiegelshohn and R. Yahyapour. Analysis of first-come-first-serve parallel job scheduling. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 629–638, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[23] U. Schwiegelshohn, R. Yahyapour, and P. Wieder. Resource management for future generation grids. Technical Report TR-0005, CoreGrid Network of Excellence – Institute on Resource Management and Scheduling, 2005.

[24] Andrew Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1994.

[25] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[26] Francois Trahay, Alexandre Denis, Olivier Aumage, and Raymond Namyst. Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager. In *EuroPVM/MPI*. Springer, 2007.