Istituto Nazionale di Fisica Nucleare
DataGrid Project

INFN/TC-xx-yy
17th April 2001

Workload Management on a Data Grid: a review of current technology

Cosimo Anglano[1,2], Stefano Barale[2], Stefano Beco[3], Salvatore Cavalieri[4,5], Flavia Donno[6], Luciano Gaido[2], Antonia Ghiselli[7], Francesco Giacomini[7], Andrea Guarise[2], Stefano Lusso[2], Ludek Matyska[8,9], Salvatore Monforte[4,5], Francesco Pacini[3], Francesco Prelz[10], Miroslav Ruda[8,9], Massimo Sgaravatto[11], Albert Werbrouck[12,2], Zhen Xie[6]

[1)] *Università del Piemonte Orientale*. [2)] *INFN, Sezione di Torino*, [3)] *DATAMAT Ingegneria dei Sistemi S.p.A.*, [4)] *Universit'a degli Studi di Catania*, [5)] *INFN, Sezione di Catania*, [6)] *INFN, Sezione di Pisa*, [7)] *INFN, CNAF*, [8)] *Masaryk University*, [9)] *CESNET*, [10)] *INFN, Sezione di Milano*, [11)] *INFN, Sezione di Padova*, [12)] *Universit'a degli Studi di Torino*

Abstract

We report on the status of current technology in the fields of job submission and scheduling (workload management) in a world-wide data grid environment.

PACS:89.80

Published by **SIS-Pubblicazioni**
Laboratori Nazionali di Frascati

Draft

Contents

1 Introduction **4**

2 Scheduling technology **6**

2.1 Scheduling Approaches 6

2.1.1 Scheduler organization 7

2.1.2 Scheduling policy 8

2.1.3 State estimation technique 8

2.2 Evaluation 9

3 Globus **11**

3.1 Introduction 11

3.2 Security services 12

3.3 The Globus Security Infrastructure (GSI) 12

3.3.1 Operation of the GSI model 12

3.3.2 GSI advantages 14

3.3.3 GSI shortcomings and ways to address them 14

3.4 Information Services for the Grid 15

3.5 The Globus Grid Information Service 16

3.5.1 Data Design 16

3.5.2 Schema Design 17

3.5.3 Extending the GRIS 18

3.5.4 Service Reliability 18

3.5.5 Data Caching 18

3.6 Globus Services for Resource Management 22

3.6.1 Globus resource management architecture 22

3.7 GRAM 23

3.8 GRAM Client API 26

3.9 GARA 27

Draft 1.0

3.9.1 Overview 27
3.9.2 Network Reservations 28
3.9.3 Comments on Network Reservations 28
3.10 Globus Executable Management 29
3.11 Heartbeat Monitor 29

4 Condor **31**
4.1 Introduction 31
4.2 Overview of the Condor system 32
4.3 Condor Classified Advertisements 34
4.4 Remote system calls 36
4.5 Checkpointing 37
4.6 Vanilla Jobs 38
4.7 Condor Flocking 38
4.8 Parallel Applications in Condor 39
4.9 Inter-Job Dependencies (DAGMAN) 39
4.10 Experiences with Condor 40
4.11 Condor-G 42
4.12 Condor GlideIn 43

Draft

Chapter 1

Introduction

The resource management of processor time, memory, network, storage, and other resources in a computational grid is clearly fundamental. The Workload Management System (WMS) is the component of the Grid middleware that has the responsibility of managing the Grid resources in such a way that applications are conveniently, efficiently and effectively executed. To carry out its tasks, the RMS interacts with other components of the grid middleware, such as the security module to perform user authentication and authorization, and the resource access module that provides the interface to the local resource management systems (e.g., queueing systems and local O.S. schedulers). From the users' point of view, resource management is (or should be) completely transparent, in the sense that their interaction with the WMS should be limited just to the description, via a high-level, user-oriented specification language, of the resource requirements of the submitted job. It is responsibility of the WMS to translate these abstract resource requirements into a set of actual resources, taken from the overall grid resource pool, for which the user has access permission.

Typically, a WMS encompasses three main modules, namely a *user access* module, a *resource information* module, and a *resource handling* module. The resource handling module performs allocation, reclamation, and naming of resources. Allocation is the ability to identify the (set of) resource(s) that best match the job requirements without violating possible resource limits (scheduling), and to control job execution on these resources. Reclamation is the ability to reclaim a resource when the job that was using it terminates its execution. Finally, naming is the ability of assigning symbolic names to resources so that jobs and schedulers may identify the resources required to carry out the computation. To carry out its tasks, the resource handling module accesses individual grid resources by interacting with the local managers of these resources (possibly via a set of standardized interfaces).

The resource handling module strongly interacts with the *resource information*

module, whose purpose is to collect and publish the information concerning the identity, the characteristics, and the status of grid resources, so that meaningful scheduling decisions may be made. The resource information module typically provides both resource dissemination and discovery facilities. The resource dissemination service provides information about the various available resources, and is used by a resource to advertise its presence on the grid so that suitable applications may use it. Resource discovery is the mechanism by which the information concerning available Grid resources are sought (and hopefully found) when a scheduler has to find a suitable set of resources for the execution of a user application.

Finally, the user access module provides users (or applications) with basic access functionalities to the main WMS service, namely resource handling. Application resource requests are described using a *job description language* that is parsed by an interpreter into the internal format used by the other WMS components. This module usually provides a user interface that includes commands to start and terminate the execution of a job, and to inspect the status of its computation.

Draft

Chapter 2

Scheduling technology

The scheduler is one of the most critical components of the resource management systems, since it has the responsibility of assigning resources to jobs in such a way that the application requirements are met, and of ensuring that possible resource usage limits are not exceeded. Although scheduling is a traditional area of computer science research, and as such many scheduling techniques for various computing systems (ranging from uniprocessors [47] to multiprocessors [24] and distributed systems [37,38,9]) have been devised, the particular characteristics of Grids make traditional schedulers inappropriate. As a matter of fact, while in traditional computing systems all the resources and jobs in the system are under the direct control of the scheduler, Grid resources are geographically distributed, heterogeneous in nature, owned by different individuals or organizations with their own scheduling policies, have different access cost models with dynamically varying loads and availability conditions. The lack of centralized ownership and control, together with the presence of many competing users submitting jobs potentially very different from each other, make the scheduling task much harder than for traditional computing systems. Many ongoing research projects [16,20,32,14,18,35] are working on the development of Grid schedulers able to deal with the above problems, but a complete solution is still lacking. By looking at these partial solutions, however, it is possible identify those characteristics that should be present in a fully-functional Grid scheduler.

2.1 Scheduling Approaches

The importance of devising suitable scheduling techniques for computational grids has been known for a long time, as witnessed by the intensive research activity carried out in this field [16,20,32,14,18,35]. Although at the moment of this writing only partial solutions have been devised, the above research efforts have converged to a common model of grid schedulers (*SuperSchedulers*), that has been published [43] as a Grid Forum [2]

specification. The above specification is, of course, only abstract, but its presentation may help to clarify the issues involved in grid scheduling. According to this specification, the first step that has to be done when a scheduling decision needs to be made is resource discovery. As a matter of fact, as mentioned before, a grid scheduler does not have neither complete knowledge nor direct control over the available resources. Consequently, before choosing the set of resources that best satisfy the needs of the job at hand, the scheduler has to (a) discover which resources are available and their characteristics, and (b) verify whether the user submitting the job has access permission to the above resources or not. In the second step, the scheduler has to select, among the resources identified in the resource discovery step, those that best match the application performance requirements. In the third and final step, the scheduler has to interact with the local resource manager in order to start the execution of the job, or to plan for its future execution (e.g., it may use advance reservation facilities if available). All the existing Grid schedulers fit more or less in the model described before, and can be classified according to three factors, namely their organization (that may be centralized, hierarchical, or distributed), their scheduling policy (that may optimize either system or application performance), and the state estimation technique they use to construct predictive models of application performance. In the following subsections we review the schedulers developed as part of very influential Grid resource management systems, namely Condor [32], MSHN [27], AppLeS [14], Ninf [35], Javelin [36], Darwin [19], Nimrod/G [16], and NetSolve [18].

2.1.1 Scheduler organization

The schedulers used in Condor and MSHN adopt a *centralized* organization, that is scheduling is under the control of a single entity, that receives and processes all the allocation requests coming from the Grid users. However, while this approach has the advantage of providing the scheduler with a global system-wide view of the status of submitted jobs and available resources, so that optimal scheduling decisions are possible, it is poorly scalable and tolerant to failures, a centralized scheduler represents a performance bottleneck and a single point of failure.

In AppLeS, Ninf, Javelin, and NetSolve, a *distributed* organization has been adopted, that is scheduling decisions are delegated to individual applications and resources. In particular, each application is free to choose (according to suitable policies) the set of resources that better fit its needs, and each resource is free to decide the schedule of the applications submitted to it. In this approach there are no bottlenecks and single points of failure but, being the scheduling decisions based on local knowledge only, resource allocation is in general sub-optimal.

Finally, Darwin and Nimrod/G adopt a *hierarchical* approach where scheduling responsibilities are distributed across a hierarchy of schedulers. Schedulers belonging to the higher levels of the hierarchy make scheduling decisions concerning larger sets of resources (e.g., the resources in a given continent), while lower-level schedulers are responsible for smaller resource ensembles (e.g., the resources in a given state). Finally, at the bottom level of the hierarchy are the schedulers that schedule individual resources. The hierarchical approach tries to overcome the drawbacks of the centralized and the distributed approach, while at the same time keeping their advantages.

2.1.2 Scheduling policy

The other important feature of a grid scheduler is the scheduling policy it adopts. The schedulers of Condor, Darwin, and MSHN adopt a *system-oriented* policy, aimed at optimizing system performance metrics such as system throughput or resource utilization. Given the rather vast body of research on system-oriented scheduling policies for traditional (distributed) computing systems, it is natural to try to re-use some of these policies (e.g., FCSF, Random, or Backfilling as proposed in [26]). However, as discussed in [28], the need of dealing with resource co-allocation and advance reservation requires the development of new system-oriented scheduling policies. Recent research efforts in this direction [28] have shown promising results, but we believe that additional research is necessary in this field.

At the other end of the spectrum, we find systems like AppLes, Javelin, NetSolve, Nimrod/G, and Ninf, that adopt *application-oriented* scheduling policies. As a matter of fact, as discussed in [13], although in several application domains high-throughput schedulers are certainly necessary, there is a complementary need of scheduling techniques able to maximize *user performance*. That is, the machines used to execute a given applications should be chosen in such a way that its performance is maximized, possibly disregarding the overall system performance.

2.1.3 State estimation technique

In order to obtain satisfactory performance, a scheduler must employ predictive models to evaluate the performance of the application or of the system, and uses this information to determine the allocation that results in best performance. Condor, Darwin, Nimrod/G, Ninf, and Javelin adopt *non-predictive* schedulers. These schedulers predict the application (or system) performance under a given schedule by assuming that the current resource status will not change during the execution of applications. Non-predictive technique, however, may result in performance much worse than expected because of the

Draft

possible presence of contention effects on the resources chosen for the execution of an application. AppLeS, NetSolve, and MSHN address this problem by adopting *predictive* schedulers, that use either heuristic or statistical techniques to predict possible changes in the resource status, so that pro-active scheduling decisions may be made. However, while predictive techniques have the potential of ensuring better application performance, they usually require a higher computational cost than their non-predictive counterparts.

2.2 Evaluation

By looking at the scheduling needs of typical Grid users, we observe that a suitable Grid scheduler should exhibit several properties not found in any of the currently available schedulers, and in particular:

- *Distributed organization*. If several user communities co-exist in a Grid, it is reasonable to assume that each of them will want to use a scheduler that better fits its particular needs (*community scheduler*). However, when a number of independent schedulers is operating simultaneously, a lack of coordination among their actions may result in conflicting and performance-hampering decisions. The need of coordination among these peer, independent schedulers naturally calls for a distributed organization.

- *Predictive state estimation*, in order to deliver adequate performance even in face of dynamic variation of the resource status.

- *Ability to interact with the resource information system.* At the moment, all the existing schedulers require that the user specifies the list of the machines that (s)he has permission to use. However, a fully functional grid scheduler should be able to autonomously find this information by interacting with the grid-wide information service.

- *Ability to optimize both system and application performance*, depending on the needs of Grid users. As a matter of fact, it is very likely that in a Grid users needing high-throughput for batches of independent jobs have to co-exist with users requiring low response times for individual applications. In this case, neither a system-oriented nor an application-oriented scheduling policy would be appropriate.

- *Submission reliability*: Grids are characterized by an extreme resource volatility, that is the set of available resource may dynamically change during the lifetime of an application. The scheduler should be able to resubmit, without requiring the user

Draft

intervention, an application whose execution cannot continue as consequence of the failure or unavailability of the machine(s) on which it is running.

- *Allocation fairness*. In a realistic system different users will have different priorities, that determine the amount of Grid resources allocated to their applications. At the moment, no Grid scheduler provides for mechanisms to specify the amount of resources assigned to users, and enforce such allocations, although some preliminary investigations on approaches based on *computational economy* [48,17] have shown promising results.

As consequence of the above considerations, it is our belief that a scheduler suitable to a realistic Grid, where different user communities compete for resources, is still lacking.

We will now move on to consider in higher detail some of the existing technologies on which a suitable grid scheduler can be built.

Draft

Chapter 3

Globus

3.1 Introduction

The software toolkit developed within the Globus ([3]) project comprises a set of components that implement basic services for security, resource location, resource management, data access, communication, etc., which facilitate the creation of usable Grids. Rather than providing a uniform, ”monolithic” programming model, the Globus toolkit provides a ”bag of services”, from which developers can select to build custom tailored tools or applications on top of them. These services are distinct and have well defined interfaces, and therefore can be integrated into applications/tools in an incremental fashion.

This layered Globus architecture is illustrated in Figure 3.1. The first layer, the Grid Fabric, comprises the underlying systems, computers, operating systems, local resource management systems, storage systems, etc. The components of the Grid fabric are integrated by Grid Services. These are a set of modules, providing specific services (information services, resource management services, remote data access services, etc.): each of these modules has a well-defined interface, which higher level services use to invoke the relevant mechanisms, and provides an implementation, which uses low-level operations to implement these functionalities. Examples of the services provided by the Globus toolkit are:

- The Grid Security Infrastructure, GSI, a library for providing generic security services for applications that will be run on the Grid resources, which might be governed by disparate security policies (see Section 3.2).
- The Grid Information Service, GIS, also known as the Metacomputing Directory Service, MDS, which provides middleware information in a common interface (see Section 3.4)

Draft 1.0

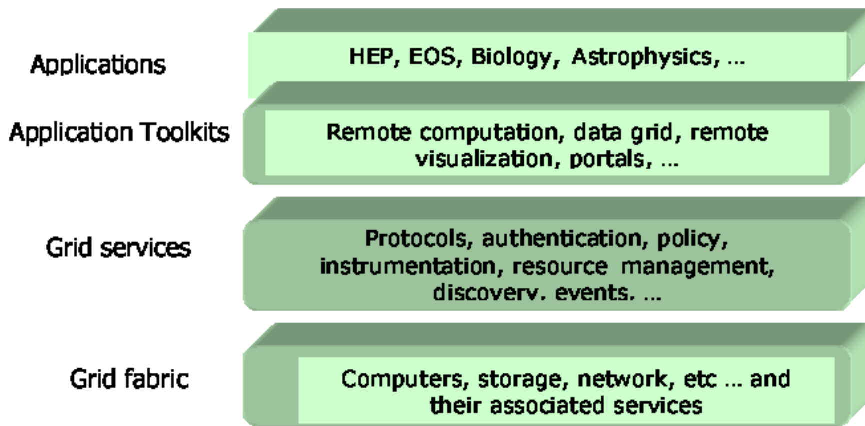

Figure 3.1: Layered structure of the Globus toolkit.

- The Globus Resource Allocation Manager (GRAM), a basic library service that provides a common user interfaces so that users can submit jobs to multiple machines on the Grid fabric (see Section 3.6).

Application toolkits use Grid services to provide higher-level capabilities, often targeted to specific classes of applications. Examples include remote job submission commands (built on top of the GRAM service), MPICH-G2, a Grid-enabled implementation of the Message Passing Interface (MPI), toolkits to support distributed management of large datasets, etc.

The fourth layer comprises the custom, tailored applications (e.g. HEP, EOS, biology applications, etc.) built on services provided by the other three layers. The Globus toolkit uses standard whenever possible, for both interfaces and implementations.

3.2 Security services

In this section we evaluate what is possibly the most mature part of the Globus package, namely the services that provide user authentication and authorization.

3.3 The Globus Security Infrastructure (GSI)

3.3.1 Operation of the GSI model

In order to describe the workings of GSI we will refer to Figure 3.2.

First of all, GSI is based on an implementation of the standard (RFC 2078/2743) Generic Security Service Application Program Interface (GSS-API). This implementation

Draft

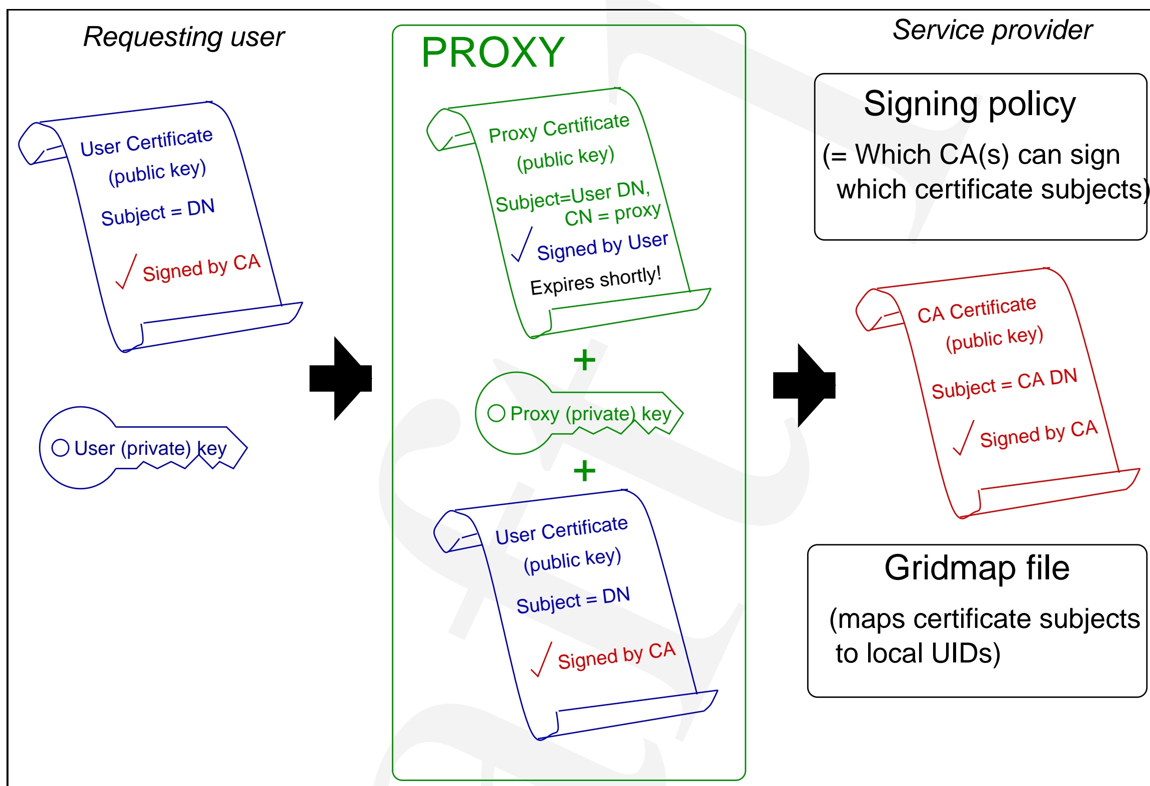

Figure 3.2: Basic elements of the GSI one-time-login model.

Draft

is based on X509 certificates and is implemented on top of the OpenSSL[4] library.

The GSS-API requires the ability to pass user authentication information to a remote site so that further authenticated connections can be established (one-time login). The entity that is empowered to act as the user at the remote site is called a ”proxy”. In the Globus implementation, a user ”proxy” is generated by first creating a fresh certificate/key pair, that are associated with the certificate of the user who is supposed to be represented by the proxy. The certificate has a short (default value of 1 day) expiration time and is signed by the user. The user certificate is signed by a Certificate Authority (CA) that is trusted at the remote end.

The remote end (usually at some service provider's site) is able to verify the proxy certificate by descending the certificate signature chain, and thus authenticate the certificate signer. The signer's identity is established by trusting the CA, and in particular by understanding and accepting its certificate issuing policy.

The last remaining step is user authorization: the requesting user is granted access and mapped to the appropriate resource provider's identifiers by checking the proxy (or user) certificate subject (X.500 DN) and looking it up in a list (the so-called *gridmap file*) that is maintained at the remote site. This list typically links a DN to a local resource username, so that the requesting user can inherit all the rights of the local user. Many DNs can be linked to the same local user.

3.3.2 GSI advantages

The GSI model provides (via the GSS-API compliance) a one-time login mechanism. This matches the ease of access requirement, and has already proven to be useful, for instance, in the existing applications of the Andrew File System (AFS).

An advantage of GSI over Kerberos authentication is the use of X509, which is by now a more widespread user identification and digital signature mechanism. X509 is also the only digital signature framework that is currently granted legal meaning in certain european countries.

GSI also provides a scheme (via the Globus Authentication and Authorization library - GAA) for extending relations of trust to multiple CAs without having to interfere with their X.500 naming scheme.

3.3.3 GSI shortcomings and ways to address them

1. The fact that authorization is based only on certificate subjects allows for the existance of multiple valid certificates for the same subject. This means that care must be taken in making sure that revoked certificates (certificates that were compro-

Draft

mised and revoked by the CA) are not accepted. The Globus GSS-API explicitly tries to find and check a CA Certificate Revocation List (CRL) when verifying proxy certificates. This means that the CRL must be present and up-to-date at every GSI-capable site. There are no specific tools in the current Globus toolkit to handle this.

2. The model where generally valid (even if for a limited time only) private keys are available on remote hosts fits a world where all system administrators are honest and able to implement a seamless security model. Good security practices call for limited scope proxy certificates. The only current limitation is in the ability to further delegate a proxy (which already is a delegated form of user credential). There is an ongoing Globus development activity (in the CAS - Community Authorization Services framework) to provide limited "capability" certificates.

3. The GSI infrastructure doesn't provide any tool to handle the association of user identities (certificate DNs) to activity (experiment) specific groups. This is an important requirement in the grid context, so some appropriate solution has to be provided, possibly also in the framework of the CAS development.

3.4 Information Services for the Grid

The Information Service (IS) plays a fundamental role in the Grid environment, since resource discovery and decision making is based upon the information service infrastructure.

Basically an information service is needed to collect and organize, in a coherent manner, information about grid resources and status and make it available to consumer entities.

Areas where information services are used for the grid are:

- **Static and dynamic information about computing resources**: this includes *host configuration and status*, a list of services, protocols and devices that can be found on hosts, access policies and other resource related information.
- **User Information**: User descriptions and groupings need to be available to client systems for authentication purposes. In the Grid environment, this is mostly a service provided by the Public Key Infrastructure of every participating institution.
- **Network status and forecasting**: Network Monitoring and forecasting software will use an information system to publish network forecasting data.

Draft

- **Software repositories**: Information on where software packages are located, what they provide, what they need etc. needs to be available on the net.

Hierarchical directory services are widely used for this kind of applications, due to their well defined APIs and protocols.

Anyway, the main issue with using the existing directory services is that they are not designed to store dynamic information such as the status of computing (or networking) resources.

3.5 The Globus Grid Information Service

The Globus *Grid Information Service* provides an infrastructure for storing and managing static and dynamic information about the status and the components of computational grids. The Grid Information Service implemented by INFN is based on the Globus package, so here we'll refer to the *Globus* GIS software.

The Globus Grid Information Service is based on LDAP directory services. As we pointed out, LDAP may not be the best choice for dynamic data storage, but provides a number of useful features:

- A well defined data model and a standard and consolidated way to describe data (ASN.1).
- A standardized API to access data on the directory servers.
- A distributed topological model that allows data distribution and delegation of access policies among institutions.

In the next sections we'll discuss directory services related topics and how they apply to the Globus grid information service.

3.5.1 Data Design

There are many issues on describing the data that the information system must contain. Basic guidelines for the GIS are:

- The information system must contain useful data for real applications (not too much, not too few).
- It is important to make a distinction between static and dynamic data so that they are treated in different ways by the various servers in the hierarchy. This can be done with the Globus package assigning different time to live to entries.

Unfortunately there are some lacking features in the Globus data representation: though LDAP represents data using ASN.1 with BER encoding (a good reference book for ASN.1 is [22]), the Globus v1.1.3 GIS implementation has no data definition types tied to attributes: all the information is represented in text format and the proper handling of attribute types is left to the backend.

As a consequence of this, it is not straightforward to apply search filters to data (for example numerical values are compared as they were strings).

3.5.2 Schema Design

LDAP schemas are meant to make data useful to data consumer entities. The schema specifies:

- An unique naming schema for objectclasses (not to be confused with the namespace).
- Which attributes the objectclass must and may contain.
- Hierarchical relationships between objectclasses.
- The data type of attributes.
- Matching rules for attributes.

The schema is then what makes data valuable to applications. Globus implemented its own schema for describing computing resources.

As with other LDAP servers, the Globus schema can be easily modified. Anyway this process must be lead by an authority, in order to make schema modifications consistent among various GIS's infrastructures.

From our perspective the Globus schema should become a standard schema to describe computing resources with the aim to make it worth for a wide range of applications and to allow an easier application development (applications must refer to a schema to know how to treat data).

3.5.2.1 The Globus Schema

The Globus schema is used to describe computing resources on a Globus host.

The current schema may need to be integrated at least with a description of:

- Devices (and file systems)

Draft

- Services (other than the services provided by Globus)

The definition and standardization of information about computing resources is largely work in progress ([5]) and the upcoming grid development efforts will have to make sure to feed back their proposals for extensions to the information schema into the standards process.

3.5.3 Extending the GRIS

A good flexibility in extending the set of data returned by each GRIS in the GIS architecture is a definite requirement for a production grid system.

The GRIS currently uses *trigger* programs to fetch data from the system. Such data is then cached for a configurable amount of time. The output of trigger programs must be in LDIF format and must respect the Globus schema. The schema can be extended to represent information other the those provided by default. In some case we concluded that a different definition of some standard information "keys" was needed. In the short term, we hope that such changes can be negotiated in the framework of our collaboration with Globus. In the long term the entire information modeling schema will probably evolve.

The Globus LDAP schema, at the moment, is not checked at runtime by the GRIS (LDAP server) to ensure data consistency.

3.5.4 Service Reliability

When a geographically distributed Information Service is implemented (see Figure 3.3 for an example), it is important, in terms or service reliability, that each site GIIS be able to operate independently from other sites and from the root GIIS server. This means that GIS clients at each site must be able to rely only on the local GIIS (as it happens, for example, with DNS clients, that don't lose the local visibility of the DNS when the root name servers are not reachable).

In the LDAP standard way to obtain this, the site GIIS server can hold a superior knowledge reference to its ancestor (a default referral to the root GIIS) and return it to the clients (that can point themselves to the local GIIS) if they wish to know about global resources (compare Figure 3.4).

3.5.5 Data Caching

At the moment the root GIIS server contains the same set of information available at lower levels, but the caching of dynamic data at the top level is often useless. Furthermore, there

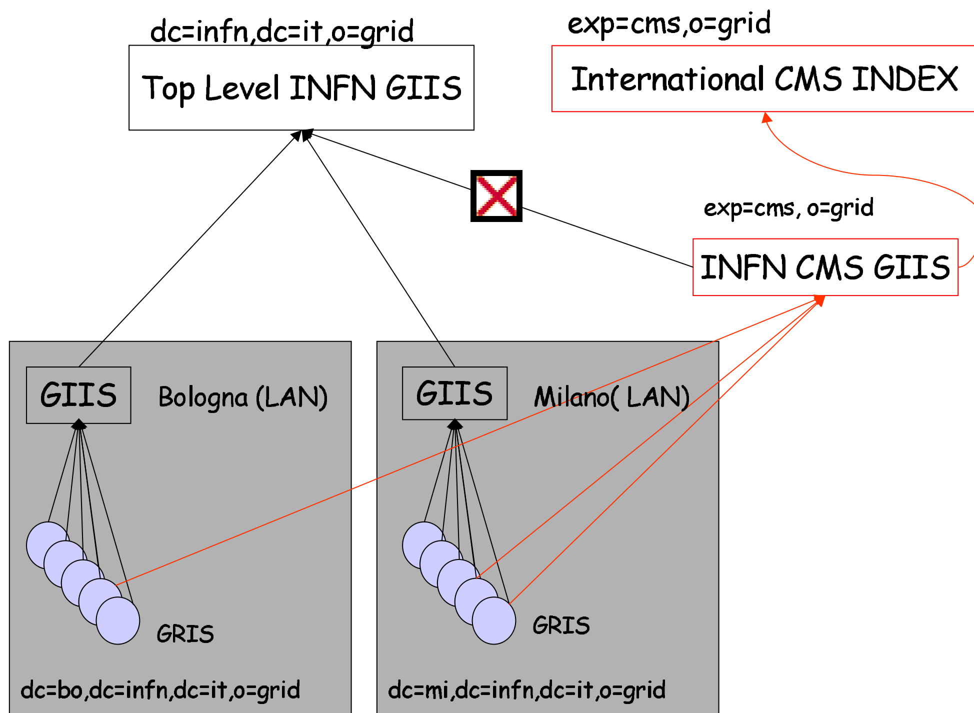

Figure 3.3: Hierarchy of Globus v1.1.3 (GIS) information services within the INFN-GRID test organization.

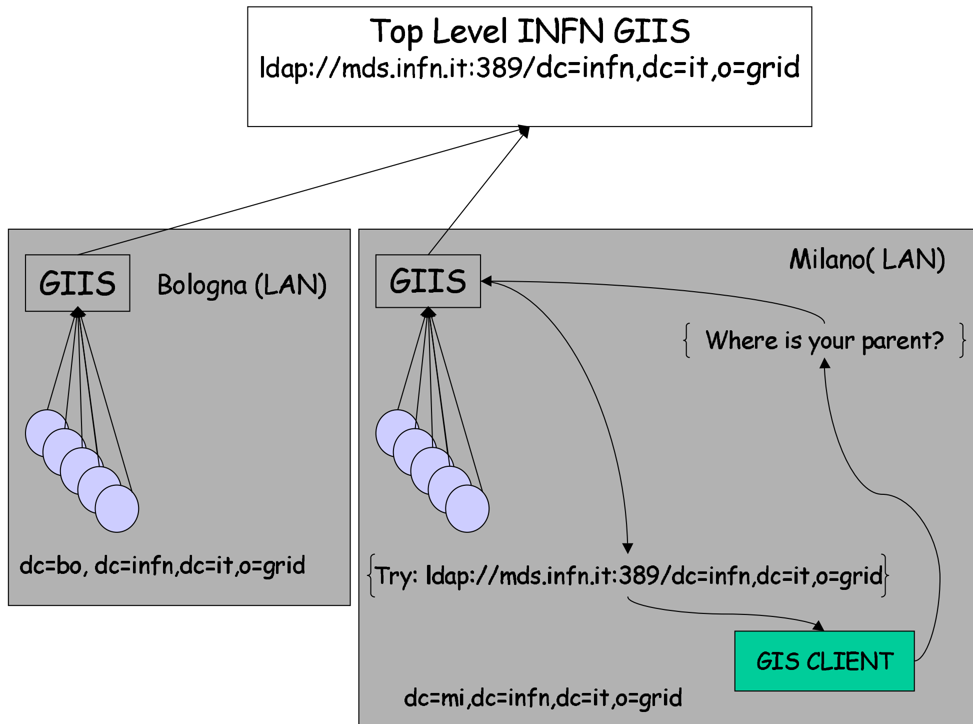

Figure 3.4: When building a geographically distributed (directory based) information tree, it is important to make sure that each site can get a reliable and consistent view of the local information in case the geographical links fail.

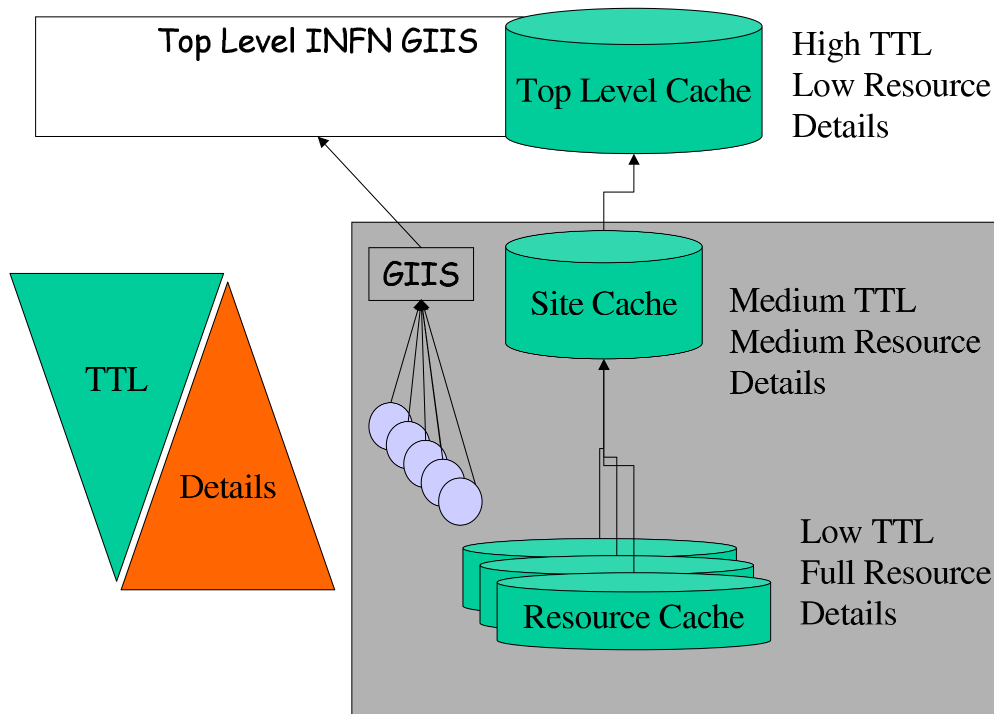

Figure 3.5: Detailed view of the GIS cache.

is some information that is convenient not to publish for security reasons: users on a host, software installed etc.

The simplest way to limit the propagation of this "classified" (and useless, if it is too dynamic) information on higher hierarchy levels is to implement access control on GIIS's in such a way that superior GIIS servers can access only a portion of the information, while authorized hosts can access the whole information. At the moment, the GIS does not implement any security mechanism, but GSI support is included in the current development code.

As stated earlier, the time to live of static information that should be available at higher levels has to be greater than the time to live of dynamic information.

As of now, the data provided by the root server is the same that inferior servers can provide, but expires less often (1 hour versus 5 minutes).

Draft

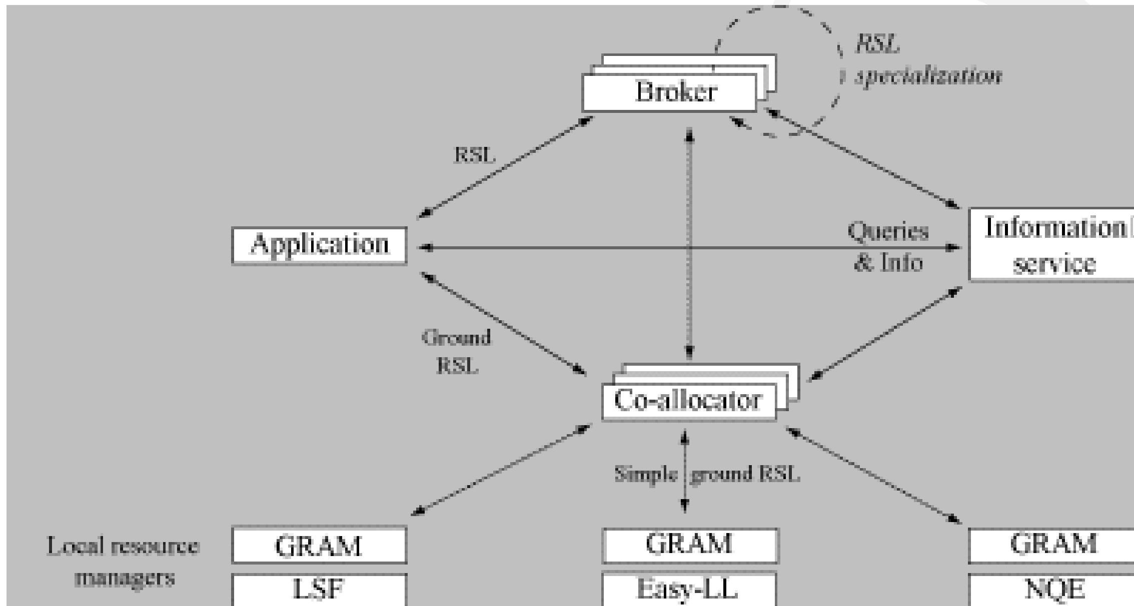

Figure 3.6: The Globus resource management architecture

3.6 Globus Services for Resource Management

3.6.1 Globus resource management architecture

The Globus resource management architecture [21][25], illustrated in Figure 3.6.1, is a layered system, in which high level services are built on top of a set of local services.

The applications express their resource requirements using a high-level RSL (Resource Specification Language) expression. One (or more) broker(s) are then responsible for taking this abstract resource specification, and translating it into more concrete specifications, using information maintained locally, and/or obtained from an Information Service (responsible for providing efficient and pervasive access to information about the current status and capability of resources), and/or contained in the high level specification. The result of this specialization is a request that can be passed to the appropriate GRAM or, in the case of a multirequest (when it is important to ensure that a given set of resources is available simultaneously), to a specific resource co-allocator. Globus doesn't provide any generic, general-purpose broker: it has to be developed for the particular applications that must be considered. The GRAM (Globus Resource Allocation Manager) is the component at the bottom of this architecture: it processes the requests for resources from remote application execution, allocates the required resources, and manages the active jobs. The current Globus implementation focuses on the management of computational

resources only.

3.7 GRAM

The Globus Resource Allocation Manager (GRAM) [21] is responsible for a set of resources operating under the same site-specific allocation policy, often implemented by a local resource management system (such as LSF, PBS, Condor). Therefore a specific GRAM doesn't need to correspond to a single host, but rather represents a service, and can provide access for example to the nodes of a parallel computer, to a cluster of PCs, to a Condor pool. In the Globus architecture the GRAM service is therefore the standard interface to "local" resources: grid tools and applications can express resource allocation and management requests in terms of a standard interface, while individual sites are not constrained in their choice of resource management tools.

The GRAM is responsible for:

- Processing RSL specifications representing resource requests, by either creating the process(es) that satisfy the request, or by denying that request;
- Enabling remote job monitoring and management;
- Periodically updating the GIS (MDS) information service with information about the current status and characteristics of the resources that it manages.

The architecture of the GRAM service is shown in Figure 3.7.

The GRAM client library is used by the application: it interacts with the GRAM gatekeeper at a remote site to perform mutual authentication and transfer the request.

The gatekeeper responds to the request by performing mutual authentication of user and resource (using the GSI service), determining the local user name for the remote user, and starting a job manager, which is executed as the selected local user and actually handles the request.

The job manager is responsible for creating the actual process(es) requested. This task typically involves submitting a request to an underlying resource management system (GRAM can operate in conjunction with several resource management systems: Condor, LSF, PBS, NQE, etc...), although if no such system exists, the fork system call may be used. Once the process(es) are created, the job manager is also responsible for monitoring their state, notifying a callback function at any state transition (the possible state transitions for a job are illustrated in Figure 3.8). The job manager terminates once the job(s) for which it is responsible have terminated. The GRAM reporter is responsible for storing into GIS (MDS) various information about the status and the characteristics of resources and jobs.

Draft

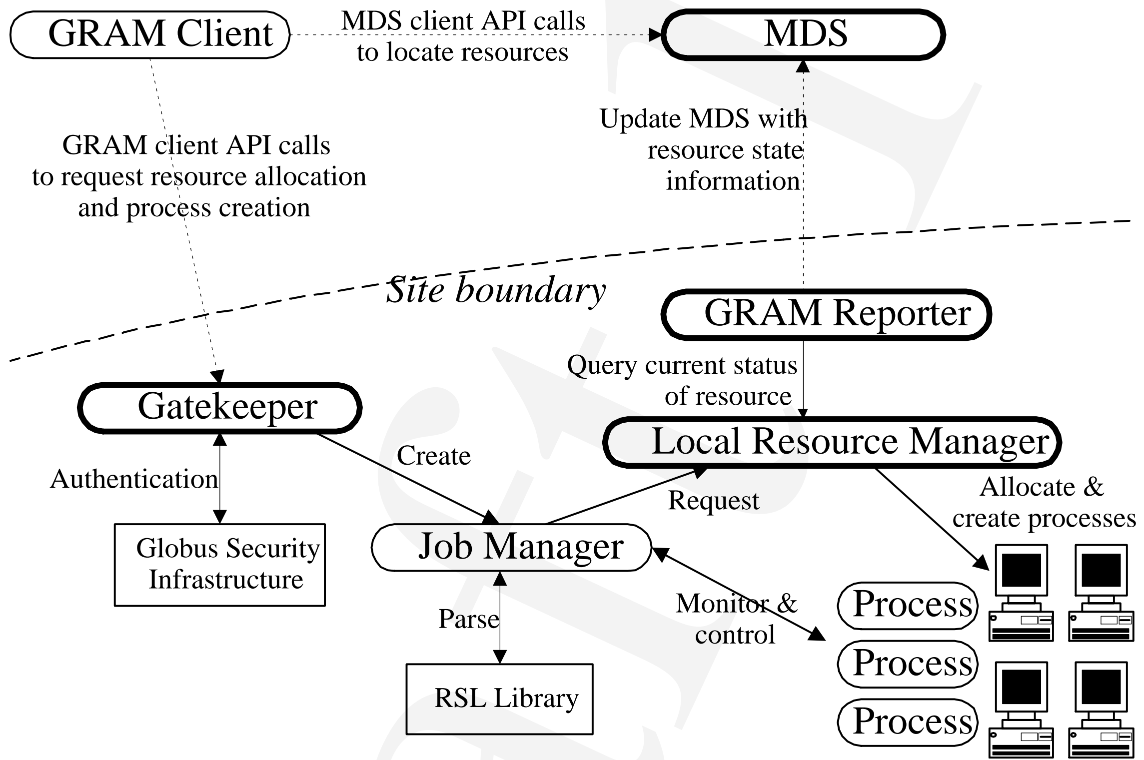

Figure 3.7: GRAM architecture

Draft

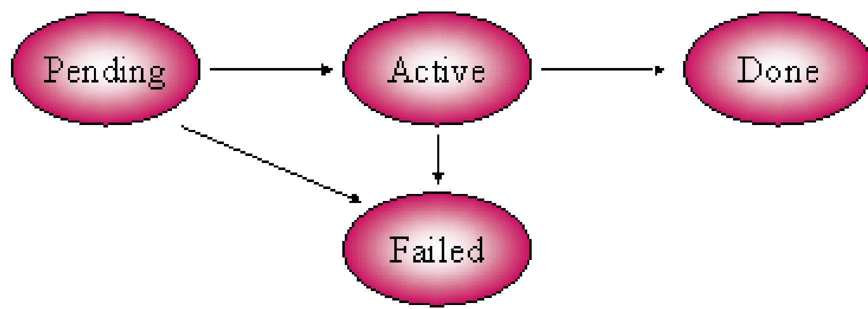

Figure 3.8: State transition diagram for a job

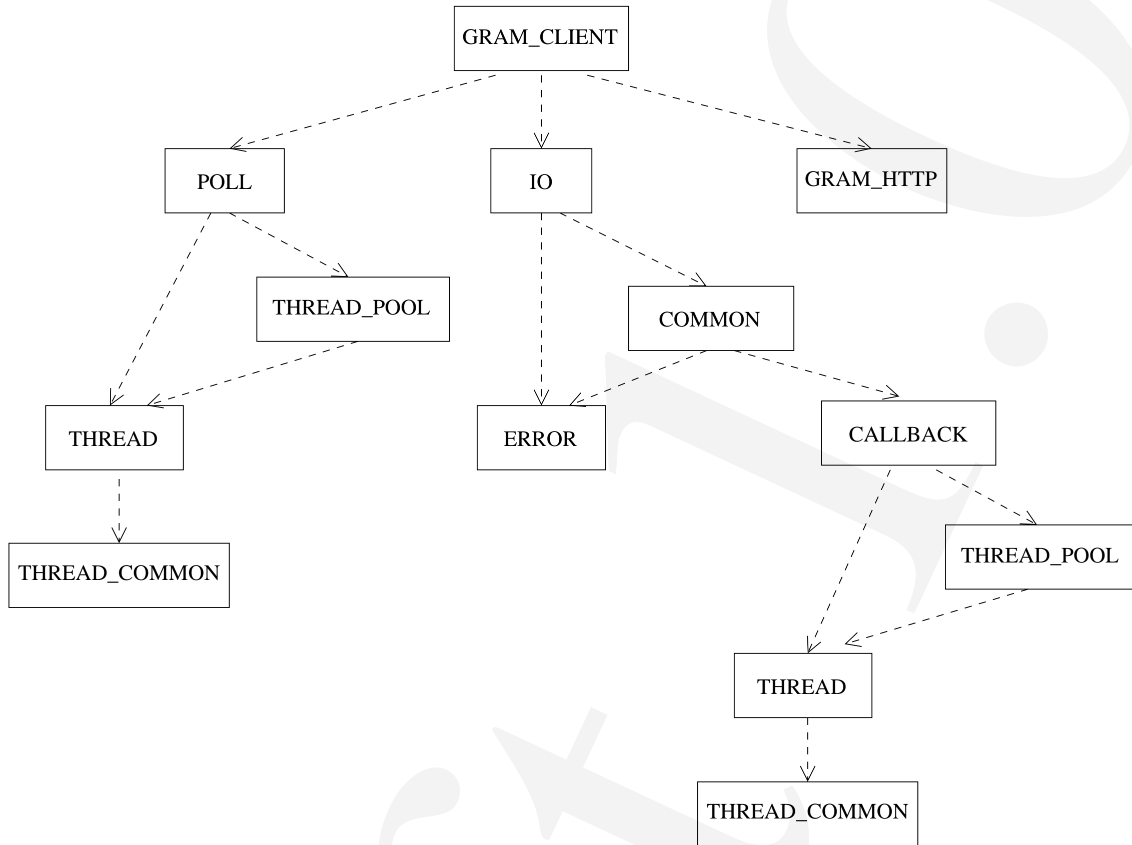

Figure 3.9: Graph of dependencies for the GRAM client module

3.8 GRAM Client API

The GRAM Client API includes eleven functions. Before any of these functions is called, a software module, which is specific of the GRAM client, has to be loaded. Module activation automatically triggers the activation of possibly other modules, which the first one relies on. In the case of the GRAM client also the POLL, IO, GRAM_HTTP modules are loaded together with all the other modules that these need. In particular during the GRAM_HTTP activation the client credentials are acquired, allowing to run in a secure environment.

The full graph of dependencies for the GRAM client module is shown in Figure 3.9.

Once the GRAM client module is not needed any more, it can be deactivated.

A job, in the form of an RSL description, can be submitted through a call to the `globus_gram_client_job_submit` function. The function returns as output a unique job handle, which can then be used for several other functions, in particular to monitor the

Draft

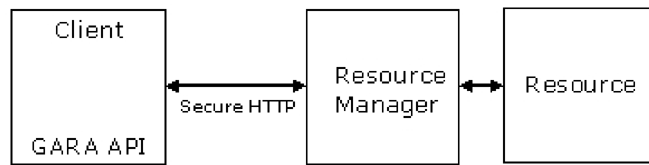

Figure 3.10: GARA basic architecture

status of the job (through the `globus_gram_client_job_status` function), or to kill it (through the `globus_gram_client_job_cancel` function).

In addition, the callback mechanism provided by the GRAM client API can be used to allow the job submitter to be asynchronously notified of a job state change.

Two functions of the API (one for jobs already submitted, and another one for not-yet-submitted jobs) allow obtaining an estimate of the time a certain job would start running. Unfortunately these two functions are not implemented yet. They would be extremely useful in the implementation of a grid scheduler, because the scheduler, if needed, could delegate the estimation of a job start time to the resource, which knows its current state better than what would be possible considering the information published in an Information Service.

From the preliminary tests done so far the GRAM client API seems quite complete and correctly implemented. Also the documentation, which happens to be quite poor for other Globus modules, is accurate enough.

3.9 GARA

In this section we provide a simple overview on GARA based on [8] and we express some comments on the network reservation part of GARA.

3.9.1 Overview

The goal of the General-purpose Architecture for Reservation and Allocation (GARA) is to provide applications with a method to reserve resources like disk space, CPU cycles and network bandwidth for end-to-end Quality of Service (QoS). GARA provides a single interface for reservation of diverse resources.

GARA has a hierarchical structure ([8], see Figure 3.10). At the lower layer the resource manager performs resource admission control (to make sure that only entitled customers actually get access to the grid resources) and reservation enforcement. Communi-

Draft

cation with the resource manager is through the Globus Gatekeeper, which authenticates and authorizes the resource requester. At layer 2 the Local Reservation implements an API for reservation request in a single trust domain. Reservation authentication through GSI is supported at layer 3, so that reservations can be requested remotely. The higher level supports mechanisms for end-to-end reservation.

Reservations can be made in advance or immediately when needed by the application itself. Transmission Quality of Service is implemented according to the Differentiated Services architecture, which provides traffic aggregates with differentiation through marking, policing, scheduling and shaping. Packets are marked at the ingress point of the network with a code called the Differentiated Services CodePoint (DSCP). Packets generated by different application sessions can share the same codepoint. Then, in each congestion point packets are placed in different dedicated queues, so that depending on the priority, they will experience different treatment. Quality of Service can be quantified through several performance metrics like: one-way delay, one-way delay variation, packet loss probability, throughput, etc.

CPU reservation is implemented through a mechanism for process scheduling called Dynamic Soft Real-Time (DSRT), while disk space reservation is based on DPSS.

3.9.2 Network Reservations

Quality of Service configuration in GARA requires three fundamental building blocks: marking, policing and scheduling. Each time a new reservation request is received, the edge router configuration has to be modified. The prototype is designed to work with CISCO routers only and it uses the Cisco Command Line Interface. Scheduling pre-configuration at the egress interface of the router is required. The mechanism requires configuration privileges on the router to proceed with router configuration every time a reservation request is received.

3.9.3 Comments on Network Reservations

We believe that the approach to network reservation adopted by GARA is of great interest, since it addresses the problem of end-to-end Quality of Service, a fundamental requirement for networked applications. However, we think that some aspects may need investigation.

The change in router configuration every time a new reservation is received is a viable solution only if the number of reservations performed locally is not frequent. The alternative approach would be to adopt static configuration, which is possible when the source/destination IP address of GRID hosts or the corresponding subnets is known in

Draft

advance.

A second issue is related to per-flow policing. The number of policing/marking instances, which have to be enabled on the input interface of the router, is a critical parameter. Performance of small edge routers is greatly dependent on the number of traffic filters (access-lists) enabled at one time for traffic policing. Per-microflow policing offers better traffic isolation at the expense of additional CPU overhead.

A third potential weakness of the architecture depends on the fact that resource reservation does not automatically recognize the ingress interface to which a policer/marker has to be associated, i.e. it does not rely on routing information, but rather requires that for each host allowed to reserve bandwidth, the corresponding input interface on the router is known. This is specified in a configuration file which has to be manually updated every time the set of local hosts varies. This approach is human-error prone.

3.10 Globus Executable Management

According to the Globus documentation [25][1], the Globus Executable Management (GEM) service should provide mechanisms to implement different distributed code management strategies, providing services for the identification, location and instantiation of executables and run time libraries, for the creation of executables in heterogeneous environments, etc.

Actually, we found that GEM doesn't exist as a package, and Globus can only provide some functionalities, to do just executable staging, that is transfer the application (i.e. the executable file) to a remote machine immediately prior to execution. This is possible if the executable file is accessible via HTTP or HTTPS, or present on the machine on which the *globusrun* command is issued.

This executable staging does not do anything with regard to moving shared libraries with the executable and setting the `LD_LIBRARY_PATH` environment variable, so if shared libraries are in non-standard places on the target machine, or if the application uses non-standard shared libraries, then this application will probably fail.

Nothing exists in the Globus toolkit about the packaging and portability issues that would allow new executables to be automatically built for a new architecture from some portable source packages.

3.11 Heartbeat Monitor

The Heartbeat Monitor (HBM) service [25][44] should provide mechanisms for monitoring the status of a distributed set of processes. Through a client interface, a process should

Draft

be allowed to register itself with the HBM service, and sending regular heartbeats to it. Moreover, a data collector API should allow a process to obtain information related to the status of other processes registered with the HBM service, thus allowing to implement, for example, fault recovery mechanisms. Unfortunately this service is not seeing active development: an HBM package, implementing some very preliminary and incomplete functionalities, has been included in the early Globus releases, but now it is not supported anymore, and has been dropped from the distribution.

Draft 1.0

Chapter 4

Condor

4.1 Introduction

Many of the typical computing problems that are planned to be submitted to computational grids require long periods (days, weeks, months) of computation to solve. Examples include different kinds of simulations, parametric studies (where many jobs must be executed to explore the entire parameter space), parallel processing of many independent data sets, etc. This user community is interested in maximizing the number of computational requests that can be satisfied over long periods of time, rather than improving the performance over short periods of time. For these users, High Throughput Computing (HTC) environments [10] [34], able to deliver large amounts of computational power over long periods of times (in contrast with classical High Performance Computing (HPC) environments, that bring enormous amounts of computing power to bear over relatively short periods of time) must be considered.

To create and deploy a usable HTC environment, the key is the effective management and exploitation of all available resources, rather than maximizing the efficiency of the existing computing systems. The distributed ownerships of the resources is probably the major obstacle: the migration from powerful, expensive central mainframe systems to commodity workstations or personal computers with better performance/price ratio has tremendously increased the overall amount of computer power, but, just because of this distributed ownership, there has not been a proportionate increase in the number of computing power available to any individual users.

For many years the Condor team at the University of Wisconsin/Madison has been designing and developing tools and mechanisms to implement a HTC environment, able to manage large collections of distributively owned computing resources: the Condor system [45] [31] [30] [33] [46] is the result of these activities.

Draft 1.0

4.2 Overview of the Condor system

A resource in the Condor system (typically a node of the distributed computing environment), is represented by a Resource-owner Agent (RA), implemented by a specific daemon (the *startd* daemon) running on this node. The resource-owner agent is responsible for enforcing and implementing the policies, which specify when a job may begin using the resource, when the job will be suspended, etc., depending on possible different factors: CPU load average, keyboard and mouse activities, attributes of the customer making the resource request, etc. (Condor is popular for harnessing idle computers CPU cycles, but it can be configured according to different policies as well). These policies are distributively and dynamically defined by the resource owners, who have complete control over their resources, and can therefore decide when, to what extent and by whom his resource can be used. The resource-owner agents periodically probe the resources to determine their current state, and report this information, together with the owners policies, to a collector, running on one well-defined machine of the Condor pool, called central manager. Customers of Condor are represented by Customer Agents (CAs), which maintain queues of submitted jobs. Like the resource-owner agents, each customer agent (implemented by the so-called *schedd* daemon) periodically sends the information concerning the job queues to the collector. For this purpose the RAs and the CAs use the same language, called Condor Classified Advertisement (ClassAd) language, to describe resource requests and resource offers (see Section 4.3). Periodically a negotiation cycle occurs (see Figure 4.1): a negotiator (matchmaker), running on the central manager, takes information from the collector, and invokes a matchmaking algorithm, which finds compatible resource requests and offers, and notifxiiesy these agents of their compatibility. Then it is up to the compatible agents to contact each other directly, using a claiming protocol, to verify if they are still "compatible" with the updated state of the resource and the request. Therefore the matching and the claiming are two distinct operations: a match is an introduction between two compatible entities, whereas a claim is the establishment of a working relationship between the entities.

If in the claiming phase the two agents agree, then the computation can start (see Figure 4.2): the *schedd* on the submitting machine starts a shadow process,which acts as connection for the job running on the remote machine, and the *startd* on the executing machine creates a starter process, responsible to stage the executable file from the submitting machine, to start the job, to monitor it, and in case to vacate it (if, for example, the resource is reclaimed by its owner).

If necessary, the matchmaking algorithm can break an existing match involving a specific resource, and create a new match between the resource and a job with a better

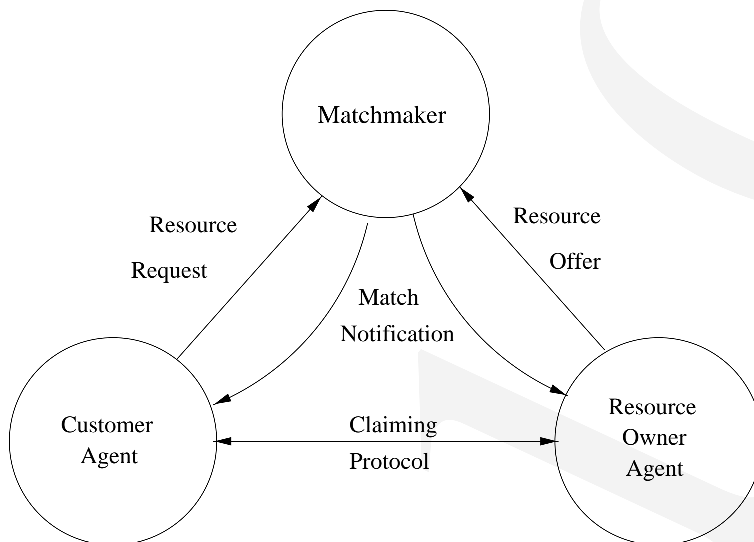

Figure 4.1: Condor matchmaking

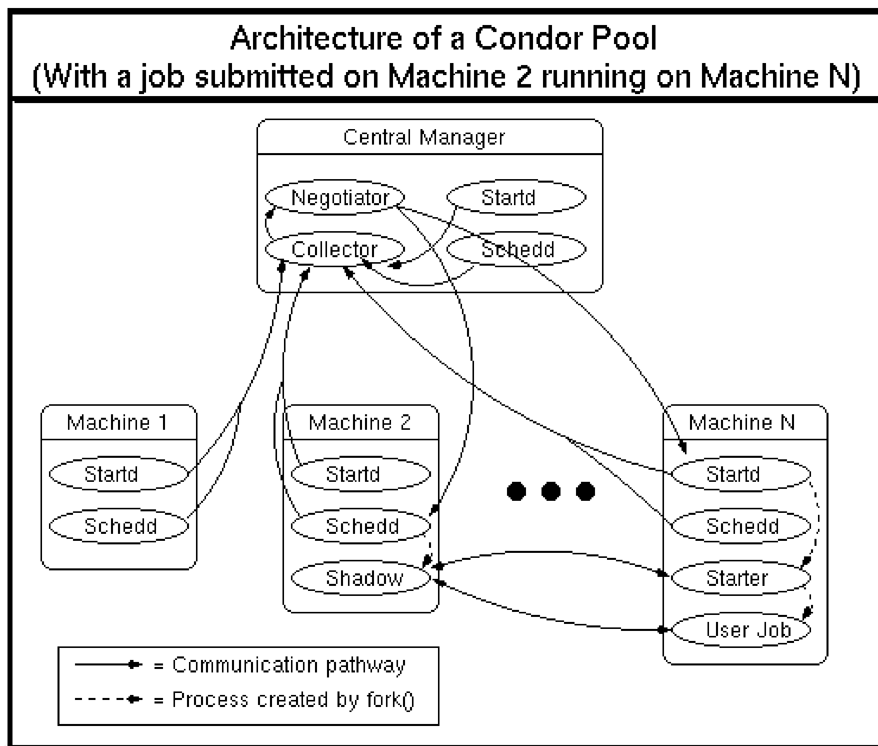

Figure 4.2: Architecture of a Condor pool

priority. This preempts the job associated with the broken match, possibly resulting in application migration.

Just re-linking the application with a specific Condor library, it is possible to exploit two distinguishing features of the Condor system, which are useful services for a HTC environment: remote system calls (see Section 4.4) and job checkpointing (see Section 4.5). Besides these so-called standard Condor jobs (job re-linked with the Condor library), Condor also provides mechanisms to support the execution of applications that haven't been re-linked with the Condor library, and therefore can't exploit the remote system call and checkpointing capabilities. These so-called *vanilla jobs* are discussed in section 4.6.

4.3 Condor Classified Advertisements

As introduced in Section 4.2, in the Condor system the resource offer and the resource request entities advertise their characteristics, requirements and preferences using the classified advertisements (ClassAds) mechanism [33] [40] [41]. A ClassAd is a mapping from attribute names to expressions. Attributes may be integer, real or string constants, or they may be more complicated expressions (constructed with arithmetic and logical operators, and record and list constructors). The ClassAd language includes a query language as part of the data model, so advertising agents can specify their compatibility by including constraints in their resource offers and requests. The ClassAd mechanisms is very flexible: it doesn't constraint which entities take part in the matchmaking process, their requirements, and how they wish to describe themselves. Moreover ClassAds use a semi-structured data model, so the matchmaker doesn't have to consider a well-defined schema.

In Condor the resource offer ClassAds and the resource request ClassAds conform to an advertising protocol that states that every ClassAd should include expressions named *Requirement* and *Rank*: in the matchmaking a pair of ClassAds are incompatible unless their *Requirement* expressions both evaluate to "true". To choose the *best* pair among compatible matches, the *Rank* attribute is then considered: among provider ClassAds matching a given customer ClassAd, the one with the highest *Rank* value, breaking ties according to the provider's *Rank*, is chosen. Figure 4.3 shows an example of a ClassAd representing a Condor resource (with a sophisticated resource usage policy), while Figure 4.4 represents a ClassAd for a job submitted for execution.

The ClassAds paradigm is very flexible, and can be generalized to include resources other than computing systems and customers other than applications. However this bilateral match can't be applied to a certain class of problems; let's consider, for example, a job, which requires for its execution a software license, besides a computing resource.

Draft

```
[
Type          = "Machine"
Activity      = "Idle";
DayTime       = 36107; // current time in seconds since midnight
KeyboardIdle  = 1432;// seconds
Disk          = 323496; //kbytes
Memory        = 64; // megabytes
State         = "Unclaimed";
LoadAvg       = 0.042969;
Mips          = 104;
Arch          = "INTEL";
OpSys         = "LINUX";
KFlops        = 21893
Name          = "lxde01.pd.infn.it";
ResearchGroup = {"user1", "user2", "user3", "user4"};
Friends       ={"frienda", "friendb"};
Untrusted     = {"rival1", "rival2"};
Rank          = member(other.Owner, ResearchGroup)*10 + (member(other.Owner, Friends);
Constraint    = !member(other.Owner, Untrusted) && Rank>= 10 ? true : Rank>0 ?
                         LoadAvg<0.3 && KeyboardIdle>15*60 : DayTime<8*60*60 ||
                        DayTime>18*60*60;
]
```

Figure 4.3: A ClassAd describing a Condor resource

```
[
Type               = "Job";
QDate              = 886799469; // Submit time secs. past 1/1/1970
CompletionDate     = 0;
Owner              = "userx";
Cmd                = "run_sim";
WantRemoteSyscalls = 1;
WantCheckpoint     = 1;
Iwd                = "/usr/userx/sim2";
Args               = "-Q 17 3200 10";
Memory             = 31;
Rank               = KFlops/1E3 + other.Memory/32;
Constraint         = other.Type=="Machine" && Arch == "INTEL" &&
                          OpSys=="LINUX" && Disk>=10000 &&
                          other.Memory>=self.Memory;
]
```

Figure 4.4: A ClassAd describing a submitted job

In this case the classical bilateral match cannot be applied, since three entities must be considered in the match: the job, the computing resource and the software license. The Condor team is addressing this problem with the so-called Gang-Matching [42], which replaces the single implicit bilateral match with an explicit list of required bilateral matches.

4.4 Remote system calls

A big issue that must be faced in overcoming the problem of distributed ownership is data access, since typically a job, placed on a remote, foreign computing resource, requires to read from and write to files on the submitting machine. Imposing a uniform, distributed file system (such as NFS or AFS) between all the machines, is a burdensome requirement, which could significantly decrease the number of accessible resources.

This problem has been addressed in the Condor system with the remote system calls mechanisms [33]. Linking the job against a specific Condor library, instead of the standard C library, nearly every system call a job performs is caught by Condor. As shown in Figure 4.5, the Condor library contains function stubs for all the system calls: these stubs send a message to the shadow, running on the submitting machine, asking it to perform the requested system call. The shadow executes the system call on the submitting machine, and sends the result back to the job. This way all I/O operations performed by

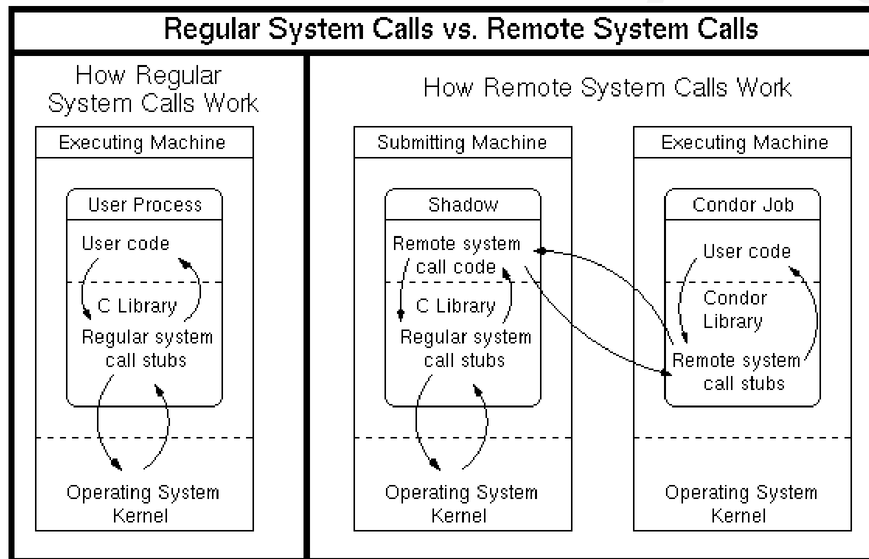

Figure 4.5: Remote system calls in Condor

the job are done on the submitting machine.

This is transparent to the job, which has no hint that the system that performed the call was actually the submitting machine, instead of the machine where it is running.

4.5 Checkpointing

Checkpointing [33] [29] a running program means taking a snapshot of its current state in such a way that the program can be restarted from that state at a later time. Since most operating systems do not provide kernel-level checkpointing services, Condor employs a user-level checkpointing capability, available for many Unix platforms. When a job must be checkpointed, its state (which includes the contents of the processs stack and data segments, all shared library code and data mapped into the processs address space, all CPU state including register values, the state of all open files, and any signal handlers and pending signals) is written to a file. To enable checkpointing, the program must be re-linked with a specific Condor library. Checkpointing is used in the Condor system when the matchmaker decides to no longer allocate a machine to a job (for example because the owner reclaims this resource): the job is checkpointed, and when a suitable replacement machine is found, the process is restored from this checkpoint, resuming the computation from where it left off, without losing the work already accomplished. Moreover, it is possible to configure Condor to perform periodic checkpoints of jobs, to

improve the fault tolerance of the system: even in spite of failures (for example a crash of the execution machine) the job can restart later from its last checkpoint, without losing the work done so far. Checkpoint files are written on the file system of the submitting machine, or it is possible to implement specialized checkpoint servers, with dedicated disk space for storing checkpoints, and with good network connection to the machines of the Condor pool.

It must be noted that checkpointing can be expensive and time consuming, since the checkpoint file can be very big, and it must be written (possibly over the network) to disk [29]. Therefore Condor foresees also other mechanisms, besides checkpointing, to preempt a running job. A job can be suspended (it keeps staying in the allocated resource, but the execution is suspended) or it may be even killed, without saving any intermediate results. For example, a possible configuration that can be implemented in the Condor system to preempt a job is by suspending it as a first step: this is a useful mechanism when the owner reclaims his resource for only a short time. If the owner keeps using the resource, the job can then be checkpointed, but if this checkpointing can't be accomplished in a relative short period (and therefore the owners can't promptly access their resources), the job is killed and must then be restarted from the beginning.

4.6 Vanilla Jobs

Some applications cannot be re-linked with the Condor library, for example because the object code is not available (this is often true for some commercial software binaries), or for the limitations of Condor for the jobs that can be checkpointed and migrated: for example IPC is not supported, only sigle process jobs are allowed, etc. ([46]). *Vanilla jobs* are intended for these kinds of jobs: they can't checkpoint or perform remote system calls, but they are still scheduled by the matchmaking system. Since vanilla jobs can't exploit the checkpointing mechanisms, when a job must be preempted from a machine, it can be suspended (and completed at a later time), or killed (and then restarted from the beginning on an other available resource). Since remote system calls are not supported, a vanilla job must run on a machine that shares the same filesystem with the submitting machine. The job must also run on a machine where the user has the same UID as on the submitting machine.

4.7 Condor Flocking

The flocking mechanism [46] [23] allows linking together different Condor pools. In the standard Condor configuration the *schedd* daemon contacts the central manager of the

Draft

local Condor pool to locate executing machines available to run jobs in its queue. In the flocking arrangement, additional central managers of remote Condor pools can be specified as configuration parameter of the *schedd* daemon. When the local pool doesn't satisfy all its job requests, the *schedd* daemon will try these remote pools in turn (so the central managers in the list should be ordered in order of preference) until all jobs are satisfied. The schedd will only send a request to a remote central manager only if the local pool and pools earlier in the list are not satisfying all the job requests. Obviously the machines of the remote pools must be configured to allow the execution from the remote machine, and the central managers must be configured to listen to requests from this remote *schedd* process.

4.8 Parallel Applications in Condor

Condor provides a framework, called Condor-PVM [39], which allows running PVM applications in the Condor environment. PVM applications which follow the master-worker paradigm are supported by Condor PVM: this model foresees that one node act as controlling master for the parallel applications, and sends pieces of work out to the worker nodes. The worker nodes do some computation, and send the result back to the master node. Condor PVM doesn't define a new API, but the existing PVM calls are used. What happens is that whenever a PVM application asks for a node, the request is re-mapped to Condor, which finds a suitable CPU in the pool using the usual Condor mechanisms, and adds it to the PVM virtual machine. If a machine needs to leave the pool, the PVM program is notified of that as well via the normal PVM mechanisms. Therefore Condor acts as e resource manager for the PVM daemon: the master is executed on the machine where the job has been submitted from, while workers are pulled in from the Condor pool as they become available.

The Condor team is currently working to include support for MPI as well.

4.9 Inter-Job Dependencies (DAGMAN)

Solving a problem may require multiple jobs that need data from each other. These problems are best represented using Directed Acyclic Graphs (DAGs), which represent the flow of control from one node to another (i.e. from one job to another). To manage this kind of problems, the Directed Acyclic Graph Manager (DAGMan) [46] can be used together with a Condor pool. The DAGMan is a meta-scheduler for Condor jobs, responsible to submit batch jobs in a predefined order and process the results: DAGMan is responsible for all the scheduling, recovery and reporting activities of the submitted job

Draft

system. To submit a DAG job, a DAG input file must be defined. In this file all the jobs that will appear in the DAG must be specified: a DAG can contain a mixture of standard and vanilla jobs, or even other DAG jobs. Then the dependencies between these jobs must be defined, specifying the *parent* and the *child* job: a child job is one whose input is taken from one or more parent jobs, and therefore it can't run until all of its parents have successfully terminated. Moreover, for each job of the DAG it is possible to specify a *pre* and/or *post* script, that are executed before/after the job is run. For example a *pre* script could be useful to put the required files into a staging area, while a *post* script could be used to copy the output files to another storage system, and then delete the staged input and output files.

4.10 Experiences with Condor

The need to share the distributely-owned computing resources at the various sites across Italy has always been a peculiar problem for INFN. Condor was identified as a possible candidate to implement such a HTC system, and therefore, beginning in 1997, it was decided to investigate its suitability for the computing needs of the INFN community [6] [15].

In collaboration with the Condor team, a single Condor pool (to optimize the CPU usage of all INFN resources) on wide area network was deployed, as a general purpose computing resource available to all INFN users.

During the test phase of this project we found that some customizations and tailoring were needed. Foe example the resource owners felt the necessity to guarantee priorities on their resource usage for particular applications (i.e. local or collaboration jobs). This requirement has been addressed by configuring sub-pools: a sub pool is a set of collaborative machine (i.e. workstations belonging to the same research group, not necessarily local to a single site), configured to prioritize collaboration user jobs. Using sub-pools it is possible to define and implement different policies and priorities on resource usage. For example, most of the sub-pools of the INFN WAN Condor pool are now configured to give highest priority to jobs of a specific group, and then prioritize jobs submitted by local users, while jobs submitted by remote users have lower priority.

A problem that came to light in the test phase is related with checkpointing. As reported in Section 4.5, checkpointing is an *expensive* operation, in particular when big checkpoint files (very common for most applications in the high energy/nuclear physics field) must be written and read across a network environment. This can significantly reduce the so-called *goodput* [11]: the allocation time when a remotely execution application uses the CPU to make forward progress, that is the true throughput obtained by the

Draft

application. We soon realized that having a single, central checkpoint server for all the hosts of the INFN WAN Condor pool was not an appropriate choice. What was needed was a proper checkpoint server topology, able to limit the checkpoint file transfers over the network, and that allowed the accomplishment of checkpoints in short time, to let the owners to access their resources without delay, and without losing checkpoint files due to network timeouts, without reducing the computing throughput. To meet these requirements, the Condor pool was partitioned into different checkpoint domains: a dedicated checkpoint server was deployed on each checkpoint domain and used by all the executing machines of the Condor pool belonging to that checkpoint domain. The definition of the checkpoint domains has been done taking into account the presence of set of machines with efficient network connectivity to the checkpoint server, the presence of a sufficiently large CPU capacity inside a domain, and the topology of sub-pools.

We realized that in a distributed environment the network must be considered a resource [15], [12], and therefore the ClassAds describing a resource of the Condor pool have been augmented, including also the bandwidth (dynamically updated) between this machine and its checkpoint server, profiting from the flexibility of the matchmaking framework, that doesn't entail a well-defined, predefined schema. When the matchmaker must find a suitable computing resource for a job, the checkpointing characteristics of this job is taken into account. The job checkpointing policies are defined by the user when submitting a job, who for example can decide that his job prefers to stay within a checkpoint domain, that a machine with a given bandwidth value to its checkpoint server must be selected, that a job can't move between different checkpoint domains (suitable for very large jobs), etc. In the long term the goal is to have a dynamic, "network aware" checkpointing system, where the association between executing machines and checkpoint servers are dynamically decided according to the network status.

At present the INFN WAN Condor pool is composed by about 230 machines (mainly Linux PCs and Digital Unix workstations) scattered in different INFN sites. Usually on these machines Condor has been configured to harness idle computer CPU cycles. The pool is currently partitioned in 7 checkpoint domains.

The total allocation time for jobs submitted to the Condor pool in the period January 2000-December 2000 is about 400000 hours (45 years). Many applications, in particular CPU intensive jobs, have succesfully exploited the INFN WAN Condor pool facility, since for these applications very good workload can be achieved running in the Condor environment; examples include Monte Carlo event simulation, simulation of Cherenkov light in the atmosphere, MC integration in perturbative QCD, dynamic chaotic systems, stochastic differential equations.

People within the INFN community that are using Condor are essentially quite

happy, since they have been able to substantially increase the throughput of computational requests that can be satisfied. The robustness and the reliability of the system are highly appreciated: in fact, besides the checkpointing mechanism (that provides fault tolerance in spite of crashes of the executing machine), Condor maintains persistent queues of the submitted jobs, so if the submitting machine crashes, Condor will be able to recover. The flexibility of the ClassAd matchmaking framework is appreciated as well, since, for example, resource owners can define new ClassAds describing particular characteristics of their resources, and users are then allowed to use these new attributes in their job request expressions.

Other users, on the contrary, reported poor performance running their I/O intensive applications in the Condor pool: we found that in this case performance improves if these jobs are forced to run in a sub-pool with a uniform filesystem. Other Condor customers found difficulties in using the Condor facility. Problems managing Condor have been reported by some Condor administrators as well, since for example it is not trivial to define particular usage policies. Also, troubleshooting is not that easy, since it is difficult to interpret the Condor system log files, primarily useful only to the Condor developers. There is also a security concern, since the authentication/authorization mechanisms in the current Condor implementation are quite primitive.

Finally its worthwhile to remark that in some INFN sites Condor is used also a local resource management system for local dedicated farms.

4.11 Condor-G

Condor-G allows submitting jobs to Globus resources, profiting from some capabilities, features and mechanisms of the Condor system. In particular, since in Condor the queue of the submitted jobs is saved in a persistent way, using Condor-G it could be possible to implement a reliable, crash-proof, checkpointable job submission service. Also, the Condor tools for job management (job submission, job removal, job status monitoring) and logging can be exploited.

The current implementation of Condor-G, which relies upon the Condor *schedd* service, runs the Globus *globusrun* command behind the scenes: a *condor_submit* command, used to submit a job to a Globus resource via Condor-G, is simply translated to the submission library equivalent of a *globusrun* command.

Condor-G doesn't provide any brokering/matchmaking functionalities (this means that the Globus resource where the job must run has to be explicitly specified), and there's no plans to provide a way to plug in application specific resource choice policies in Condor-G: the place to implement this resource choice is within a component that sits

Draft

on top of Condor-G itself.

4.12 Condor GlideIn

With GlideIn, what happens is that the Condor daemons (specifically, the *master* and the *startd*) are effectively run on Globus resources (machines that use the fork system call as job manager, or clusters managed by a resource management system). These resources then temporarily (the Condor daemons exit gracefully when no jobs run for a configurable period of time) become part of a given Condor pool, which can then be used to submit any kind (standard or vanilla) of Condor jobs. GlideIn is a particular implementation of the Condor-G mechanism: the Condor master is submitted as a Condor-G job.

The GlideIn procedure operates in two steps, after acquiring a valid user proxy. In the first step, that must be considered only once, the Condor executables and configuration files are downloaded from a server in Wisconsin, while in the second phase the Condor daemons are executed in the remote Globus resource.

Draft

Bibliography

[1] http://www.globus.org/hbm/heartbeat_spec.html.

[2] The global grid forum home page. http://www.gridforum.org.

[3] Home page for the globus project. http://www.globus.org.

[4] Home page for the openssl project. http://www.openssl.org/.

[5] Home page of the global grid forum information services working group. http://www-unix.mcs.anl.gov/gridforum/gis.

[6] The infn condor on wan project. http://www.infn.it/condor.

[7] Notes on extending the gris information schema. http://www.globus.org/mds/extending-gris.html.

[8] Administrator guide to gara. http://www-fp.mcs.anl.gov/qos/papers/gara_admin_guide.pdf, March 2000.

[9] C. Anglano. A Fair and Effective Scheduling Strategy for Workstation Clust ers. In *Proc. of the IEEE Int. Conference on Cluster Computing*. IEEE-CS Press, December 2000.

[10] J. Basney and M. Livny. Deploying a high throughput computing cluster. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1. Prentice Hall, 1999.

[11] J. Basney and M. Livny. Improving goodput by co-scheduling cpu and network capacity. *Int'l Journal of High Performance Computing Applications*, 13(3), 1999.

[12] J. Basney and M. Livny. Managing network resources in condor. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), Pittsburgh, Pennsylvania*, pages 298–299, August 2000.

Draft

[13] F. Berman. High-Performance Schedulers. In *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[14] F. Berman and R. Wolski. The AppLeS Project: A Status Report. In *Proc. of the 8th NEC Research Symposium*, Berlin, Germany, May 1997.

[15] D. Bortolotti, T. Ferrari, A. Ghiselli, P. Mazzanti, F. Prelz, M. Sgaravatto, and C. Vistoli. Condor on wan. In *Proceedings of the CHEP 2000 conference*, 2000.

[16] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *Proc. of Int. Conf. on High Performance Computing in Asia-Pacific Region*, Beijing, China, 2000. IEEE-CS Press.

[17] R. Buyya, D. Abramson, and J. Giddy. An Economy Grid Architecture for Service-Oriented Grid Computing. In *Proc. of the 10th Int. Workshop on Heterogeneous Computing*. IEEE-CS Press, 2001.

[18] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *Intl. Journal of Supercomputing Applications and High Performance Computing*, 11(3), 1997.

[19] P. Chandra, A. Fisher, and C. Kosak et al. Darwin: Customizable Resource Management for Value-Added Network Services. In *Proc. of the 6th Int. Conf. on Network Protocols*. IEEE, 1988.

[20] S. Chapin, J. Karpovich, and A. Grimshaw. The Legion Resource Management System. In *Proc. of the 5th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*. Springer, 1999.

[21] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[22] Olivier Dubuisson. Asn.1 - communication between heterogeneous systems.

[23] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. In *Journal of Future Generations of Computer Systems*, volume 12, 1996.

Draft

[24] D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevci k, and P. Wong. Theory and Practice in Parallel Job Scheduling. In *Proc. of IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science. Springer, 1997.

[25] I. Foster and C. Kesselman. The globus project: A status report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[26] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of Job-Scheduling Strategies for Grid Computing. In *Proc. of 1st ACM/IEEE Int. Workshop on Grid Computing*, number 1971 in Lecture Notes in Computer Science. Springer, 2000.

[27] D. Hensgen and T. Kidd. An Overview of MSHN: The Management System for Heterogeneous Networks. In *Proc. of the 8th Workshop on Heterogeneous Computing*. IEEE-CS Press, April 1999.

[28] J. Hollingsworth and S. Maneewongvatana. Imprecise Calendars: an Approach to Scheduling Computational Grids. In *Proc. of the 19th Int. Conf. on Distributed Computing Systems*. IEEE-CS Press, 1999.

[29] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system, April 1997.

[30] M. J. Litzkow and M. Livny. Experience with the condor distributed batch system. In *Proc. of the IEEE Workshop on Experimental Distributed Systems, Huntsville, Alabama*, October 1990.

[31] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. In *Proc. of the 8th Int'l Conf. On Distributed Computing Systems*, pages 104–111, 1998.

[32] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th Int. Conf. of Distributed Computing Systems*, 1988.

[33] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP Journal*, 11(1):36–40, 1997.

[34] M. Livny and R. Raman. High-throughput resource management. In *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[35] H. Nakada, M. Sato, and S. Sekiguchi. Design and Implementation of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems*, October 1999. Special Issue on Metacomputing.

[36] M. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0: Java-Based Parallel Computing in the Internet. In *Proc. of the European Parallel Computing Conference (EUROPAR 2000)*, 2000.

[37] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. of 3rd Int. Conf. on Distributed Computing Systems*, pages 22–30, May 1982.

[38] F. Petrini and W. Feng. Scheduling with Global Information in Distributed Systems. In *Proc. of 20th Int. Conf. on Distributed Computing Systems*. IEEE-CS Press, 2000.

[39] J. Pruyne and M. Livny. Interfacing condor and pvm to harness the cycles of workstation clusters. 12, 1996.

[40] R. Raman, M. Livny, and M. Salomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing, Chicago, Illinois*, July 1998.

[41] R. Raman, M. Livny, and M. Solomon. Matchmaking: an extensible framework for distributed resource management. *Cluster: Journal of Software, Networks and Applications*, 2(2), 1999.

[42] R. Raman, M. Livny, and M. Solomon. Gang-matchmaking: Advanced resource management through multilateral matchmaking. In *Proceedings of the 9th IEEE Int'l Symposium on High Performance Distributed Computing*, August 2000.

[43] J. Schopf. Ten Steps for SuperScheduling. http://www.cs.nwu.edu/ jms/sched-wg/WD/schedwd.8.3.pdf, February 2000. Working Draft no. 8.3.

[44] P. Stelling, I. Foster, C. Kesselman, C.Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, pages 268–278, 1998.

[45] Condor Team. The condor high throughput computing environment. `http://www.cs.wisc.edu/condor`.

[46] Condor team. Condor manual. `http://www.cs.wisc.edu/condor.manual`.

Draft

[47] U. Vahalia. *Unix Internals: The New Frontiers*. Prentice Hall, 1996.

[48] R. Wolski, J. Plank, J. Brevik, and T. Bryan. G-Commerce: Market Formulations Controlling Resource Allocation on the Computational Grid. In *Proc. of the Int. Parallel and Distributed Processing Symposium (IPDPS 2001)*. IEEE-CS Press, 2001.

Draft 1.0