A Resource Management Architecture for Metacomputing Systems

Karl Czajkowski * Ian Foster † Carl Kesselman *
Stuart Martin † Warren Smith † Steven Tuecke †

* Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292-6695

† Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Abstract

Metacomputing systems are intended to support remote and/or concurrent use of geographically distributed computational resources. Resource management in such systems is complicated by five concerns that do not typically arise in other situations: site autonomy and heterogeneous substrates at the resources, and application requirements for policy extensibility, co-allocation, and online control. We describe a resource management architecture that addresses these concerns. This architecture distributes the resource management problem among distinct local manager, resource broker, and resource co-allocator components, and defines an extensible resource specification language to exchange information about requirements. We describe how these techniques have been implemented in the context of the Globus metacomputing toolkit and used to implement a variety of different resource management strategies. We report on our experiences applying our techniques in a large testbed, GUSTO, incorporating 15 sites, 330 computers, and 3600 processors.

1 Introduction

Metacomputing systems allow applications to assemble and use collections of computational resources on an as-needed basis, without regard to physical location. Various groups are implementing such systems and exploring applications in distributed supercomputing, high-throughput computing, smart instruments, collaborative environments, and data mining [10, 12, 18, 20, 22, 6, 25].

This paper is concerned with *resource management* for metacomputing: that is, with the problems of locating and allocating computational resources, and with authentication, process creation, and other activities required to prepare a resource for use. We do not address scheduling (such as decomposition, assignment, and execution ordering of tasks) or the management of other resources such as memory, disk, and networks.

The metacomputing environment introduces five challenging resource management problems: site autonomy, heterogeneous substrate, policy extensibility, co-allocation, and online control.

1. The *site autonomy* problem refers to the fact that resources are typically owned and operated by different organizations, in different administrative domains [5]. Hence, we cannot expect to see commonality in acceptable use policy, scheduling policies, security mechanisms, and the like.

2. The *heterogeneous substrate* problem derives from the site autonomy problem, and refers to the fact that different sites may use different local resource management systems [16], such as Condor [18], NQE [1], CODINE [11], EASY [17], Condor [18], LSF [28], PBS [14], and LoadLeveler [15]. Even when the same system is used at two sites, different configurations and local modifications often lead to significant differences in functionality.

3. The *policy extensibility* problem arises because metacomputing applications are drawn from a wide range of domains, each with their own requirements. A resource management solution must support the frequent development of new domain-specific management structures, without requiring changes to code installed at participating sites.

4. The *co-allocation* problem arises because many applications have resource requirements that can be satisfied only by using resources simultaneously at several sites. Site autonomy and the possibility of failure during allocation introduces a need for specialized mechanisms for allocating multiple resources, initiating computation on those resources, and monitoring and managing those computations.

5. The *online control* problem arises because substantial negotiation can be required to adapt application requirements to resource availability, particularly when requirements and resource characteristics change during execution. For example, a tele-immersive application that needs to simulate a new entity may prefer a lower-resolution rendering, if the alternative is for the entity not to be modeled at all. Resource management mechanisms must support such negotiation.

As we explain in Section 2 below, no existing resource management systems addresses all five problems. Some batch queuing systems support co-allocation, but not site autonomy, policy extensibility, and online control [16]. Condor supports site autonomy, but not co-allocation or online control [18]. Legion [12] and Gallop [26] address online control and policy extensibility, but not the heterogeneous substrate or co-allocation problems.

In this paper, we describe a resource management architecture that we have developed to address the five problems listed above. In this architecture, developed in the context of the Globus project [10], we address problems of site autonomy and heterogeneous substrate by introducing entities called *resource managers* to provide a well-defined interface to diverse local resource management tools, policies, and security mechanisms. To online control and policy extensibility, we define an extensible *resource specification language* that supports negotiation between different components of a resource management architecture, and introduce *resource brokers* to handle the mapping of high-level application requests into requests to individual managers. We address the problem of co-allocation by defining various co-allocation strategies, which we encapsulate in *resource co-allocators*.

One measure of success for an architecture such as this is its usability in a practical setting. To this end, we have implemented and deployed this architecture on GUSTO, a large computational grid testbed comprising 15 sites, 330 computers, and 3600 processors, using LSF, NQE, LoadLeveler, EASY, Fork, and Condor as local schedulers. To date, this architecture and testbed have been used by ourselves and others to implement numerous applications and half a dozen different higher-level resource management strategies. This successful experiment represents a significant step forward in terms of number of global metacomputing services implemented and number and variety of commercial and experimental local resource management systems employed.

The rest of this paper is structured as follows. In the next section, we review current distributed resource management solutions. In subsequent sections we first outline our architecture and then examine each major function in detail: the resource specification language, local resource managers, resource brokers, and resource co-allocators. We summarize the paper and discuss future work in Section 8.

2 Resource Management Approaches

Previous work on resource management for metacomputing systems can be broken into two broad classes:

- *Network batch queuing systems.* These systems focus strictly on resource management issues for a set of networked computers. These systems do not address policy extensibility and provide only limited support for online control and co-allocation.
- *Wide-area scheduling systems.* Here, resource management is performed as a component of mapping application components to resources and scheduling their execution. To date, these systems do not address issues of heterogeneous substrates, site autonomy, and co-allocation.

In the following, we use representative examples of these two types of system to illustrate the strengths and weaknesses of current approaches.

2.1 Networked Batch Queuing Systems

Networked batch queuing systems, such as NQE [1], CODINE [11], LSF [28], PBS [14], and LoadLeveler [15], handle user-submitted jobs by allocating resources from a networked pool of computers. The user characterizes application resource requirements either explicitly, by some type of job control language, or implicitly, by selecting the queue to which a request is submitted to. Networked batch queuing systems are typically designed for single administrative domains, making site autonomy difficult to achieve. Likewise, the heterogenous substrate problem is also an issue due to the fact that these systems generally assume that they are the only resource management system in operation. One exception is the CODINE system which introduces the concept of a *transfer queue* to allow jobs submitted to CODINE to be allocated by some other resource management system, at a reduced level of functionality. An alternative approach to supporting substrate heterogeneity is being explored by the PSCHED [13] initiative. This project is attempting to define a uniform API through which a variety of batch scheduling systems may be controlled. The goals of PSCHED are similar in many ways to those of the Globus Resource Allocation Manager described in Section 5 below.

Batch scheduling systems provide a limited form of policy extensibility in that resource management policy is set by either the system or the system administrator, by the creation of scheduling policy or batch queues. However, this capability is not available to the end user, who has little control over how the batch scheduling system interprets their resource requirements.

Finally, we observe that batch queuing systems have limited support for on-line allocation, as these systems are designed to support applications in which the requirements specifications are in the form "get X done soon" where X is precisely defined but "soon" is not. In metacomputing applications, we have more complex, fluid constraints, in which we will want to make tradeoffs between time (when) and space (physical characteristics). This leads to a need for the resource management system to provide capabilities such as negotiation, inquiry interfaces, information-based control, co-allocation, etc., none of which are provided in these systems.

In summary, batch scheduling systems do not provide in themselves a complete solution to metacomputing resource management problems. However, clearly some of the mechanisms developed for resource location, distributed process control, remote file access, to name a few, can be applied to wide area systems as well. Furthermore, we note that network batch queuing systems will necessarily be part of the local resource management solution. Hence, any metacomputing resource management architecture must be able to interface to these systems.

2.2 Wide-Area Scheduling Systems

We now examine how resource management is addressed within systems developed specifically to schedule metacomputing applications. In order to gain a good perspective on the range of possibilities, we discuss four different schedulers, designed variously to support specific classes of applications (Gallop [26]), an extensible object-oriented system (Legion [12]), general classes of parallel programs (PRM [22]), and high-throughput computation (Condor [18]).

The **Gallop** [26] system allocates and schedules tasks defined by a static task graph onto a set of networked computational resources. (A similar mechanism has been used in Legion [27].) Resource allocation is implemented by a scheduling manager, which coordinates scheduling requests and a local manager, which manages the resources at a local site, potentially interfacing to site-specific scheduling and resource allocation services. This decomposition, which we also adopt, separates local resource management operations from global resource management policy and hence facilitates solutions to the problems of site autonomy, heterogenous substrates, and policy extensibility. However, Gallop does not appear to handle authentication to local resource management services, limiting the level of site autonomy that can be achieved.

The use of a static task-graph model makes online control in Gallop difficult. Resource selection is performed by attempting to minimize the execution time of task graph as predicted by a performance model for the application and the prospective resource. However, because the minimization procedure and the cost model is fixed, there is no support for policy extensibility. **Legion** [12] overcomes this limitation by leveraging its object-oriented model. Two specialized objects, an application-specific `Scheduler` and a resource-specific `Enactor` negotiate with one another to make allocation decisions.

Gallop supports co-allocation for resources maintained within an administrative domain, but depends for this purpose on the ability to reserve resources. Unfortunately, reservation is not currently supported by most local resource management systems. For this reason, our architecture does not rely on reservation to perform co-allocation, but rather uses a separate co-allocation management service to perform this function.

The **Prospero Resource Manager** [22] (PRM) provides resource management functions for parallel programs written using the PVM message passing library. PRM consists of three components: a system manager, a job manager, and a node manager. The job manager makes allocation decisions, while the system and node manager actually allocate resources. The node manager is solely responsible for implementing resource allocation functions, thus PRM does not address issues of site autonomy or substrate heterogeneity. A variety of job managers can be constructed, allowing for policy extensibility, although there is no provision for composing job managers so as to extend an existing management policy. As in our architecture, PRM has both an information infrastructure (Prospero [21]) and a management API, providing the infrastructure needed to perform online control. However, unlike our architecture, PRM does not support co-allocation of resources.

Condor [18] is a resource management system designed to support high-throughput computations by discovering idle resources on a network and allocating those resources to application tasks. While Condor does not interface with existing resource management systems, resources controlled by Condor are deallocated as soon as the "rightful" owner starts to use them. In this sense, Condor supports site autonomy and heterogeneous substrates. However, Condor currently does not interoperate with local resource authentication, limiting the degree of autonomy a site can assert. Condor provides an extensible resource description language, called *classified ads* which provides limited control over resource selection to both the application and resource. However, the matching of application component to resource is performed by a system *classifier*, which defines how matches—and consequently resource management—takes place, limiting the extensibility of this selection policy. Finally, Condor provides no support for co-allocation or online control.

In summary, our review of current resource management approaches revealed a range of valuable services, but no single system provides solutions to all five metacomputing resource management problems posed in the introduction.

3 Our Resource Management Architecture

Our approach to the metacomputing resource management problem is illustrated in Figure 1. In this architecture, an extensible *resource specification language* (RSL), discussed in Section 4 below, is used to communicate requests for resources between components: from applications to resource brokers, resource co-allocators and resource managers. At each stage in this process, information about resource requirements is coded an an RSL expression by the application or by refined by one or more resource brokers and co-allocators; information about resource availability and characteristics can be is obtained from an an information service.

Resource brokers are responsible for taking high-level RSL specifications and transforming them into more concrete specifications through a process we call *specialization*. As illustrated in Figure 2, multiple brokers may be involved in servicing a single request, with application-specific brokers translating application requirements into more concrete resource requirements, and different resource brokers being used to locate available resources that meet those requirements.

Transformations effected by resource brokers generate a specification in which the locations of the required resources are completely specified. Such a *ground request* can be passed to a *co-allocator*, which is responsible for coordinating the allocation and management of resources at multiple sites. As we describe in Section 7 below, a variety of co-allocators will be required in a metacomputing system, providing different co-allocation semantics.

Resource co-allocators break a multirequest—that is, a request involving resources at multiple sites—into its constituent elements and pass each component to the appropriate *resource*

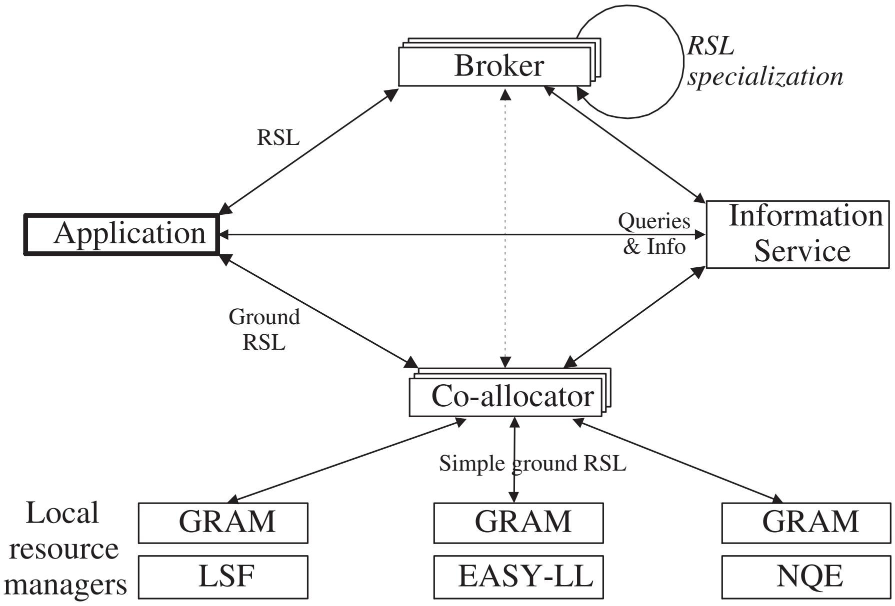


Figure 1: The Globus resource management architecture, showing how RSL specifications pass between application, resource brokers, resource co-allocators, and local managers (GRAMs). Notice the central role of the information service.

manager. As discussed in Section 5, each resource manager in the system is responsible for taking a RSL request and translating it into operations in the local, site-specific resource management system.

The *information service* is responsible for providing efficient and pervasive access to information about the current availability and capability of resources. This information is used to locate resources with particular characteristics, to identify the resource manager associated with a resource, to determine properties of that resource, and for numerous other purposes during the process of translating high-level resource specifications into requests to specific managers. We use the Globus system's Metacomputing Directory Service (MDS) [8] as our information service. MDS uses the data representation and application programming interface (API) defined on the Lightweight Directory Access Protocol (LDAP) to meet requirements for uniformity, extensibility, and distributed maintenance. It defines a data model suitable for distributed computing applications, able to represent computers and networks of interest, and provides tools for populating this data model. LDAP defines a hierarchical, tree-structured name space called a *directory information tree* (DIT). Fields within the namespace are identified by a unique *distinguished name* (DN). LDAP supports both distribution and replication. Hence, the local service associated with MDS is exactly an LDAP server (or a gateway to another LDAP server, if multiple sites share a server), plus the utilities used to populate this server with up-to-date information about the structure and state of the resources within that site. The global MDS service is simply the ensemble of all these servers. An advantage of using MDS as our information service is that resource management information can be used by other tools, as illustrated in Figure 3.

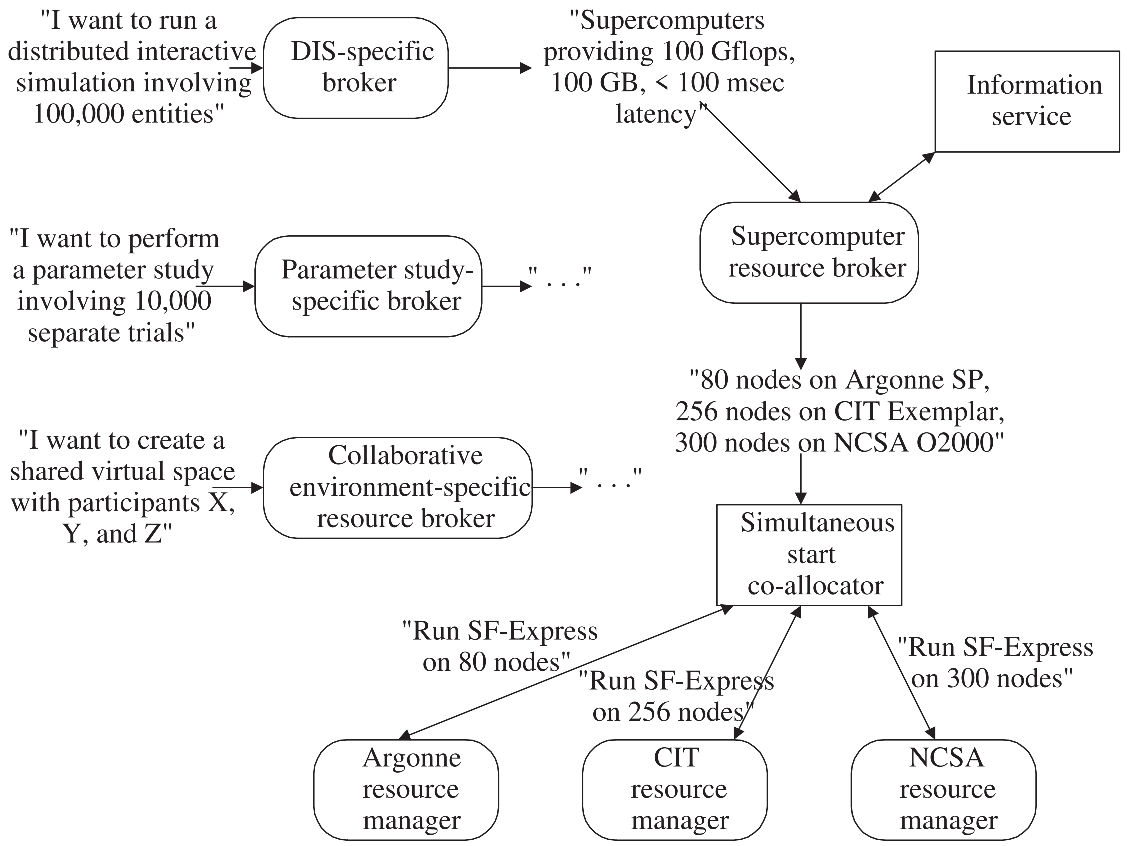
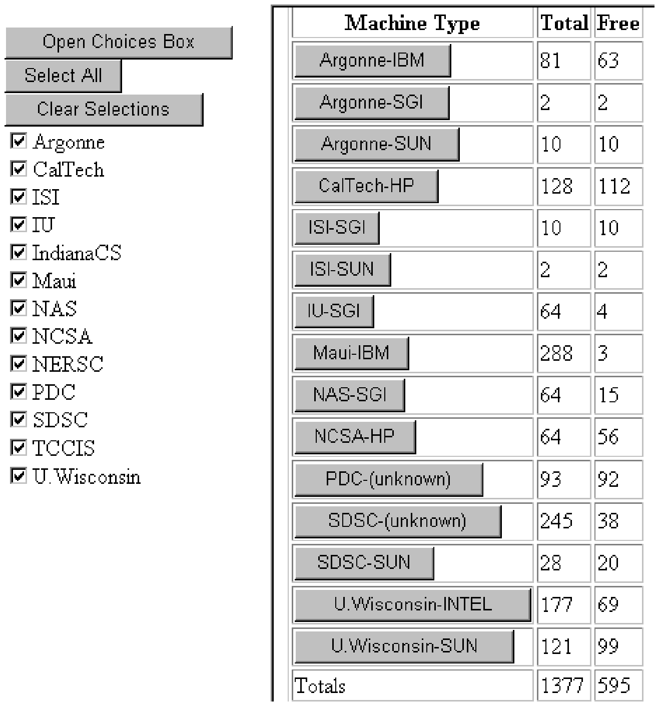


Figure 2: This view of the Globus resource management architecture shows how different types of broker can participate in a single resource request



Machine Type	Total	Free
Argonne-IBM	81	63
Argonne-SGI	2	2
Argonne-SUN	10	10
CalTech-HP	128	112
ISI-SGI	10	10
ISI-SUN	2	2
IU-SGI	64	4
Maui-IBM	288	3
NAS-SGI	64	15
NCSA-HP	64	56
PDC-(unknown)	93	92
SDSC-(unknown)	245	38
SDSC-SUN	28	20
U.Wisconsin-INTEL	177	69
U.Wisconsin-SUN	121	99
Totals	1377	595

Figure 3: The GlobusView tool uses MDS information about resource manager status to present information about the current status of a metacomputing testbed

```
specification    := request
request          := multirequest | conjunction | disjunction | parameter
multirequest     := + request-list
conjunction      := & request-list
disjunction      := | request-list
request-list     := ( request ) request-list | ( request )
parameter        := parameter-name op value
op               := = | > | < | >= | <= | !=
value            := ([a..Z][0..9][_])+
```

Figure 4: BNF grammar describing the syntax of a RSL request

4 Resource Specification Language

We now discuss the resource specification language itself. The syntax of an RSL specification, summarized in Figure 4, is based on the syntax for filter specifications in the Lightweight Directory Access Protocol (LDAP) and MDS. An RSL specification is constructed by combining simple parameter specifications and conditions with the operators `&`, to specify conjunction of parameter specifications, `|`, to express the disjunction of parameter specifications, or `+`, to combine two or more requests into a single compound request, or multirequest.

The set of `parameter-name` terminal symbols is extensible: resource brokers, co-allocators, and resource managers can each define a set of parameter names that they will recognize. For example, a resource broker that is specialized for tele-immersive applications might accept as input specification containing a `frames-per-second` parameter and generate as output a specification containing a `mflops-per-second` parameter, to be passed to a broker that deals with computational resources. Resource managers, the system components that actually talk to local scheduling systems, recognize two types of `parameter-name` terminal symbols:

- *MDS attribute names*, used to express constraints on resources: e.g., `memory>=64`, `network=atm`. In this case, the parameter name refers to a field defined in the MDS entry for the resource being allocated. The truth of the parameter specification is determined by comparing the value provided with the specification with the current value associated with the corresponding field in the MDS. Arbitrary MDS fields can be specified by providing their full distinguished name.
- *Scheduler parameters*, used to communicate information regarding the job, such as `count` (number of nodes required), `max_time` (maximum time required), `executable`, `arguments`, `directory`, and `environment` (environment variables). Schedule parameters are interpreted directly by the resource manager.

For example, the following specification:

```
&(executable=myprog)
 (|(&(count=5)(memory>=64))(&(count=10)(memory>=32)))
```

requests 5 nodes with at least 64 MB memory, or 10 nodes with at least 32 MB. In this request, `executable` and `count` are scheduler attribute names, while `memory` is an MDS attribute name.

Our current RSL parser and resource manager disambiguate between these two parameter types on the basis of the parameter name. That is, the resource manager knows which fields it will accept as scheduler parameters and assumes all others are MDS attribute names. Name clashes can be disambiguated by using the complete distinguished name for the MDS field in question.

The ability to include constraints on MDS attribute values in RSL specifications is important. As we discuss in Section 5 below, the state of resource managers is stored in MDS. Hence, resource specifications can refer to resource characteristics such as queue-length, expected wait time, number of processors available, etc. This technique provides a powerful mechanism for controlling how an RSL specification is interpreted.

The following example of a multirequest is derived from the example shown in Figure 2.

```
+(&(count=80)(memory>=64M)(executable=sf_express)
   (resourcemanager=ico16.mcs.anl.gov:8711))
 (&(count=256)(network=atm)(executable=sf_express)
   (resourcemanager=neptune.cacr.caltech.edu:755))
 (&(count=300)(memory>=64M)(executable=sf_express)
   (resourcemanager=modi4.ncsa.edu:4000))
```

Notice that this is a ground request: every component of the multirequest specifies a resource manager. A co-allocator can use the `resourcemanager` parameters specified in this request to determine to which resource manager each component of the multirequest should be submitted.

Notations intended for similar purposes include the Condor "classified ad" [18] and Chapin's "task description vector" [5]. Our work is novel in three respects: the tight integration with a directory service, the use of specification rewriting to express broker operations (as described below), and the fact that the language and associated tools has been implemented and demonstrated effective when layered on top of numerous different low-level schedulers.

We conclude this section by noting that it is the combination of resource brokers, information service, and RSL that makes online control possible in our architecture. Together, these services make it possible to construct requests dynamically, based on current system state and negotiation between the application and the underlying resources.

5 Local Resource Management

We now proceed to describe the lowest level of our resource management architecture, namely the local resource managers. The implementation of this entity in our architecture is called a Globus Resource Allocation Manager (GRAM), which is responsible for

1. processing RSL specifications representing resource requests, by either denying the request or by creating one or more processes (a "job") that satisfy that request;

2. enabling remote monitoring and management of jobs created in response to a resource request; and

3. periodically updating the MDS information service with information about the current availability and capabilities of the resources that it manages.

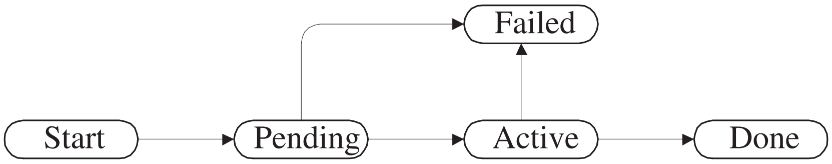


Figure 5: State transition diagram for resource allocation requests submitted to the GRAM resource management API

As indicated above, a GRAM is intended to serve as the interface between a wide area metacomputing environment and an autonomous entity able to create processes, such as a parallel computer scheduler or a Condor pool. Notice that this means that a resource manager need not correspond to a single host or a specific computer, but rather to a service that acts on behalf of one or more computational resources. This use of local scheduler interfaces was first explored in the software environment for the I-WAY networking experiment [9], but is extended and generalized here significantly to provide a richer and more flexible interface.

A resource specification passed to a GRAM is assumed to be ground: that is, to be sufficiently concrete that the GRAM can identify local resources that meet the specification without further interaction with the entity that generated the request. A particular GRAM implementation may achieve this goal by scheduling resources itself or, more commonly, by mapping the resource specification into a request to some local resource allocation mechanisms. (To date, we have interfaced GRAM to six different schedulers or resource allocators: Condor, EASY, Fork, LoadLeveler, LSF, and NQE.) Hence, the GRAM API plays for resource management a similar role to that played by IP for communication: it can co-exist with local mechanisms, just as IP rides on top of ethernet, FDDI, or ATM networking technology.

The GRAM API provides functions for submitting and for canceling a job request, and for asking when a job (submitted or not) is expected to run. An implementation of the latter function may use queue time estimation techniques [24]. When a job is submitted, a globally unique *job handle* is returned that can then be used to monitor and control the progress of the job. In addition, a job submission call can request that the progress of the requested job be signaled asynchronously to a supplied *callback URL*. Job handles can be passed to other processes, and callbacks do not have to be directed to the process that submitted the job request. These features of the GRAM design facilitate the implementation of diverse higher-level scheduling strategies. For example, a high-level broker or co-allocator can make a request on behalf of an application, while the application monitor the progress of the request.

5.1 GRAM Scheduling Model

We discuss briefly the scheduling model defined by GRAM, as this is relevant to subsequent discussion of co-allocation. This model is illustrated in Figure 5, which shows the state transitions that may be experienced by a GRAM job.

When submitted, the job is initially **pending**, indicating that resources have not yet been allocated to the job. At some point in time, the job is allocated the requested resources and the application starts running. The job then transitions to the **active** state. At any point prior

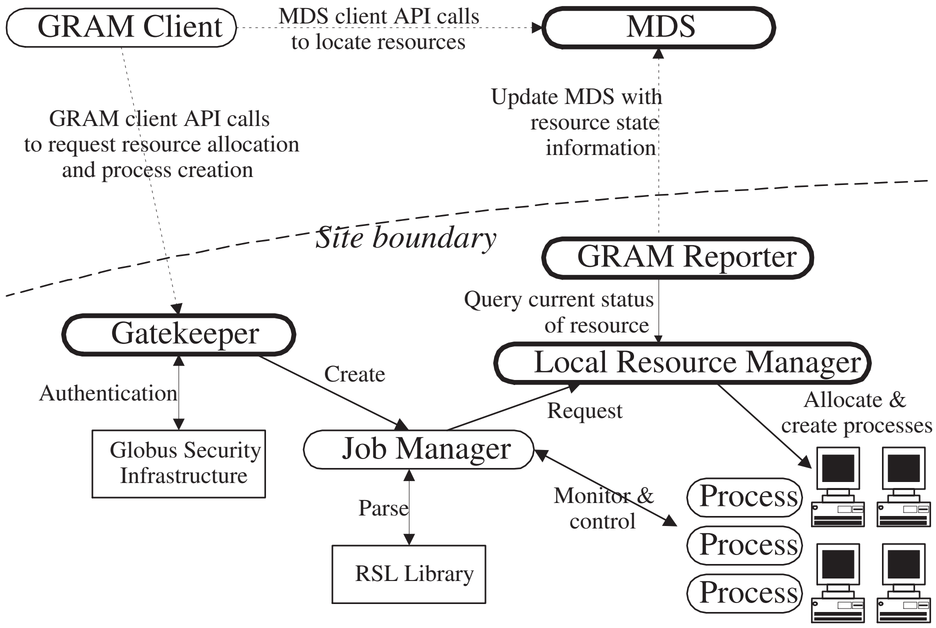


Figure 6: Major components of the GRAM implementation. Those represented by thick-lined ovals are long-lived processes, while the thin-lined ovals are short-lived processes created in response to a request.

to entering the **done** state, the job can be terminated, causing it to enter the **failed** state. A job can fail due to explicit termination, an error in the format of the request, a failure in the underlying resource management system or a denial of access to the resource. The source of the failure is provided as part of the notification of state transition. When all of the processes in the job have terminated and resources have been deallocated, the job then enters the **done** state.

5.2 GRAM Implementation

The GRAM implementations that we have constructed have the structure shown in Figure 6. The principal components are the GRAM client library, the gatekeeper, the RSL parsing library, the job manager, and the GRAM reporter. The Globus security infrastructure (GSI) is used for authentication and for authorization.

The *GRAM client library* is used by an application or a co-allocator acting on behalf of an application. It interacts with the GRAM gatekeeper at a remote site to perform mutual authentication and transfer a request, which comprises a resource specification, a callback (described below), and a few other components that are not relevant to the current discussion.

The *gatekeeper* is an extremely simple component that responds to a request by doing three things: performing mutual authentication of user and resource, determining a local user name for the remote user, and starting a job manager which executes as that local user and actually handles the request. The first two security-related tasks are performed by calls to the Globus security infrastructure (GSI), which handles issues of site autonomy and substrate heterogeneity in the security domain. In order to start the job manager, the gatekeeper must run as a privileged program: on Unix systems, this is achieved via `suid` or `inetd`. However, because the interface to the GSI is small and well defined, it is easy for organizations to easily and approve the gatekeeper code. In fact, the gatekeeper code has successfully undergone security reviews at a number of large supercomputer centers. The mapping of remote user to locally recognized user name minimizes the amount of code that must run as a privileged program; it also allows us to delegate most authorization issues to the local system.

A *job manager* is responsible for creating the actual processes requested by the user. This task typically involves submitting a resource allocation request to the underlying resource management system, although if no such system exists on a particular resource, a simple `fork` may be performed. Once processes are created, the job manager is also responsible for monitoring the state of the created processes, notifying the callback contact of any state transitions, and implementing control operations such as process termination. A job manager terminates once the job for which it is responsible has terminated.

The *GRAM reporter* is responsible for storing into MDS various information about scheduler structure (e.g., whether the scheduler supports reservation and the number of queues) and state (e.g., total number of nodes, number of nodes currently available, currently active jobs, and expected wait time in a queue). An advantage of implementing the GRAM reporter as a distinct component is that MDS reports can continue even when no gatekeeper or job manager is running: for example, when the gatekeeper is run from `inetd`.

As noted above, GRAM implementations have been constructed for six local schedulers to date: Condor, LSF, NQE, Fork, EASY, and LoadLeveler. Much of the GRAM code is independent of the local scheduler and so only a relatively small amount of scheduler-specific code needed to be written in each case. In most cases, this code comprises shell scripts that make use of the local scheduler's user-level API. State transitions are mostly handled by polling, because this proved to be more reliable than monitoring job process by using mechanisms provided by the local schedulers.

6 Resource Brokers

As noted above, we use the term *resource broker* to denote an entity in our architecture that translates abstract resource specifications into more concrete specifications. As illustrated in Figure 2, this definition is broad enough to encompass a variety of behaviors, including application-level schedulers [3] that encapsulate information about the types of resource required to meet a particular performance requirement, resource locators that maintain information about the availability of various types of resource, and (ultimately) traders that create markets for resources. In each case, the broker uses information maintained locally, obtained from MDS, or contained in the specification to *specialize* the specification, mapping it into a new specification that contain more detail. Requests can be passed to several brokers, effectively composing the behaviors of those brokers, until eventually the specification is specialized to the point that it identifies a specific resource manager. This specification can then be passed to the appropriate GRAM or, in the case of a multirequest, to a resource co-allocator.

We claim that our architecture makes it straightforward to develop a variety of higher-level schedulers. In support of this claim, we note that following the definition and implementation of GRAM services, a variety of people, including people not directly involved in GRAM definition, were able to construct half a dozen resource brokers quite quickly. We describe three of these here.

6.1 Nimrod-G

David Abramson and Jonathan Giddy are using GRAM mechanisms to develop Nimrod-G, a wide-area version of the Nimrod [2] tool. Nimrod automates the creation and management of large parametric experiments. It allows a user to run a single application under a wide range of input conditions and then to aggregate the results of these different runs for interpretation. In effect, Nimrod transforms file-based programs into interactive "meta-applications" that invoke user programs much as we might call subroutines.

When a user first requests a computational experiment, Nimrod/G queries MDS to locate suitable resources. It uses information in MDS entries to identify sufficient nodes to perform the experiment. The initial Nimrod-G prototype operates by generating a number of independent jobs, which are then allocated to computational nodes using GRAM. This module hides the nature of the execution mechanism on the underlying platform from Nimrod, hence making it possible to schedule work using a variety of different queue managers without modification to the Nimrod scripts. As a result, a reasonably complex cluster computing system could be retargeted for wide area execution with relatively little effort.

In the future, the Nimrod-G developers plan to provide a higher level broker that allows the user to specify time and cost constraints. These constraints will be used to select computational nodes that can meet user requirements for time and cost or, if constraints cannot be met, to explain the nature of the cost/time tradeoffs. As part of this work, a dynamic resource allocation module is planned that will monitor the state of each system, and relocate work when necessary in order to meet the deadlines.

6.2 AppLeS

Rich Wolski has used GRAM mechanisms to construct an application-level scheduler (AppLeS) [3] for a large loosely coupled problem from computational mathematics. As in Nimrod-G, the goal was to map a large number of independent tasks to a dynamically varying pool of available computers. GRAM mechanisms were used to locate resources (including parallel computers) and to initiate and manage computation on those resources. AppLeS itself provided fault tolerance, so that errors reported by GRAM would result in a task being resubmitted elsewhere.

6.3 A Graphical Resource Selector

The graphical resource selector (GRS) illustrated in Figure 7 is an example of an interactive resource selector constructed with our services. This Java application allows the user to build up a network representing the resources required for an application; another network can be constructed to monitor the status of candidate physical resources. A combination of automatic and manual techniques are then used to guide resource selection, eventually generating an RSL specification for the resources in question. MDS services are used to obtain the information used for resource monitoring and selection, and resource co-allocator services to generate the GRAM requests required to execute a program once a resource selection is made.

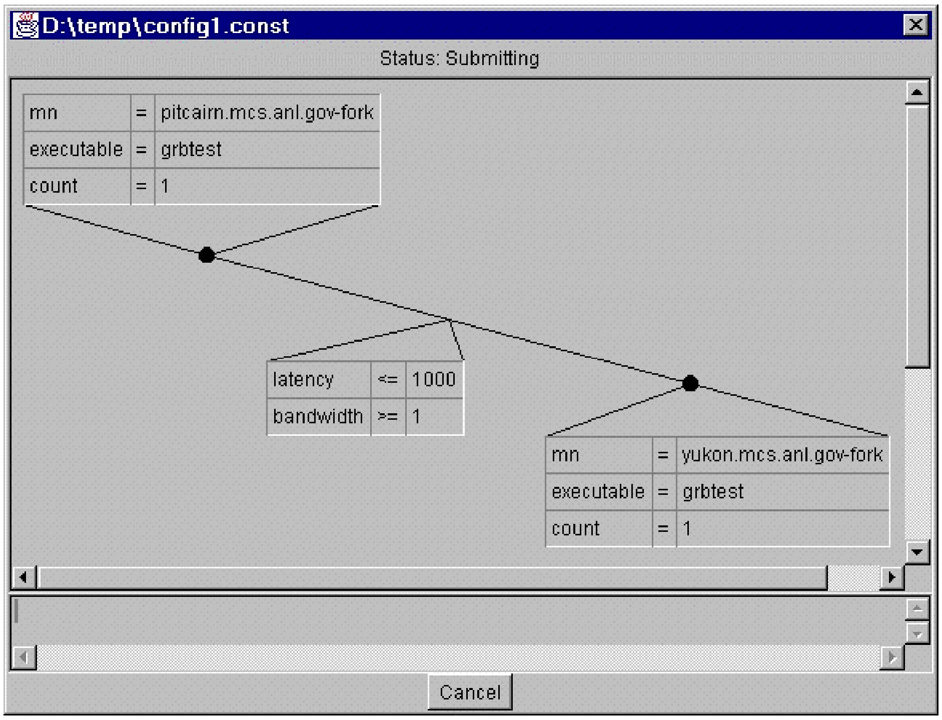


Figure 7: A screen shot of the Graphical Resource Selector. *A more interesting shot will be provided in the final paper.*

7 Resource Co-allocation

Through the actions of one or more resource brokers, the requirements of an application are refined into a ground RSL expression. If the expression consists of a single resource request, it can be submitted directly to the manager that controls that resource. However, as discussed above, it is often the case that a metacomputing application requires that several resources—such as two or more computers and intervening networks—be allocated simultaneously. In these cases, a resource broker produces a multirequest and co-allocation is required. The challenge in responding to a co-allocation request is to allocate the requested resources in a distributed environment, across two or more resource managers, where global state, such as availability of a set of resources, is difficult to determine.

Within our resource management architecture, multirequests are handled by an entity called a resource co-allocator. In brief, the role of a co-allocator is to split a request into its constituent components, submit each component to the appropriate resource manager, and then provide a means for manipulating the resulting set of resources as a whole: for example, for monitoring job status or terminating the job. Within these general guidelines, a range of different co-allocation service can be constructed. For example, we can imagine allocators that:

- mirror current GRAM semantics: that is, require all resources to be available before the job is allowed to proceed, and fail globally if failure occurs at any resource;
- allocate at least N out of M requested resources then return; or
- return immediately, but gradually return more resources as they become available.

Each of these services is useful to a class of applications. To date, we have had the most experience with a co-allocator that takes the first of these approaches: that is, extends GRAM semantics to provide for simultaneous allocation of a collection of resources, enabling the distributed collection of processes to be treated as a unit. We discuss this co-allocator in more detail.

Fundamental to a GRAM-style concurrent allocation algorithm is the ability to determine if the desired set of resources is available at some time in the future. If the underlying local schedulers support reservation, this question can be easily answered by obtaining a list of available time slots from each participating resource manager, and choosing a suitable timeslot [23]. Ideally, this scheme would use transaction-based reservations across a set of resource managers, as provided by Gallop [26]. In the absence of transactions, the ability to either make a tentative reservation or to retract an existing reservation in needed. However, in general, a reservation-based strategy is limited due to the fact that currently deployed local resource management solutions do not support reservation.

In the absence of reservation, we are forced to use indirect methods to achieve concurrent allocation. These methods optimistically allocate resources in the hope that the desired set will be available at some "reasonable" time in the future. Guided by sources of information, such as the current availability of resources (provided by MDS) or queue-time estimation [24, 7], a resource broker can construct a RSL request that is *likely*, but not guaranteed, to succeed. If for some reason the allocation eventually fails, all of the started jobs must be terminated. This approach has several drawbacks:

- It is inefficient in that computational resource are wasted while waiting for all of the requested to become available.

- We need to ensure that application components do not start to execute before the co-allocator can determine if the request will succeed. Therefore, the application must perform a barrier operation to synchronize startup across components, meaning that the application must be altered beyond what is required for GRAM.

- Detecting failure of a request can be difficult if some of the request components are directed to resource managers that interface to queue-based local resource management systems. In these situations, a timeout must be used to detect failure.

However, in spite of all of these drawbacks, co-allocation can frequently be achieved in practice as long as the resource requirements are not large compared to the capacity of the metacomputing system.

We have implemented a GRAM-compatible co-allocator that implements a job abstraction in which multiple GRAM subjobs are collected into a single distributed job entity. State information for the distributed job is synthesized from the individual states of each subjob and job control (e.g., cancellation) is automatically propagated to the resource managers at each subjob site. Subjobs are started independently and as discussed above must perform a runtime checkin operation. With the exception of this checkin operation, the co-allocator interface is a drop-in replacement for GRAM.

We have used this co-allocator to manage resources for SF-Express [19, 4], a large scale distributed interactive simulation application. Using our co-allocator and the GUSTO testbed, we were able to simultaneously obtain 852 compute nodes on three different architectures located at six different computer centers, controlled by three different local resource managers. The use of a co-allocation service significantly simplified the process of resource allocation and application startup.

Running SF-Express "at scale" on a realistic testbed allowed us to study the scalability of our co-allocation strategy. One clear lesson learned is that the strict "all or nothing" semantics of the distributed job abstraction severely limited scalability. Even if each individual parallel computer is reasonably reliable and well understood, the probability of subjob failure due to improper configuration, network error, authorization difficulties, etc., increases rapidly as the number of subjobs increases. Yet many such failure modes resulted simply from a failure to allocate a specific instance of a commodity resource, for which an equivalent resource could easily have been substituted. Because such failures frequently occur after a large number of subjobs have been successfully allocated, it would be desirable to make the substitution dynamically, rather than to cancel all the allocations and start over.

We plan to extend the current co-allocation structure to support such dynamic job structure modification. By passing information about the nature of the subjob failure out of the co-allocator, a resource broker can edit the specification, effectively implementing a backtracking algorithm for distributed resource allocation. Note that we can encode the necessary information about failure in a modified version of the original RSL request, which can be returned to the component that originally requested the co-allocation services. In this way, we can iterate through the resource-broker/co-allocation components of the resource management architecture until an acceptable collection of resources has been acquired on behalf of the application.

8 Conclusions

We have described a resource management architecture for metacomputing systems that addresses requirements for of site autonomy, heterogeneous substrates, policy extensibility, co-allocation, and online control. This architecture has been deployed and applied successfully in a large testbed comprising 15 sites, 330 computers, and 3600 processors, within which LSF, NQE, LoadLeveler, EASY, Fork, and Condor were used as local schedulers.

The primary focus of our future work in this area will be on the development of more sophisticated resource broker and resource co-allocator services within our architecture, and on the extension of our resource management architecture to encompass other resources such as disk and network. We are also interested in the question of how policy information can be encoded so as to facilitate automatic negotiation of policy requirements by resources, users, and processes such as brokers acting as intermediaries.

Acknowledgments

We gratefully acknowledge the contributions made by many colleagues to the development of the GUSTO testbed and the Globus resource management architecture: in particular, Doru Marcusiu at NCSA and Bill Saphir at NERSC. This work was supported by DARPA under contract N66001-96-C-8523, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] Cray Research, 1997. Document Number IN-2153 2/97.

[2] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proc. 4th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1995.

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing '96*. ACM Press, 1996.

[4] S. Brunett and T. Gottschalk. Scalable ModSAF simulations with more than 50,000 vehicles using multiple scalable parallel processors. In *Proceedings of the Simulation Interoperability Workshop*, 1997.

[5] S. Chapin. Distributed scheduling support in the presence of autonomy. In *Proc. Heterogeneous Computing Workshop*, pages 22–29, 1995.

[6] Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Moré. The Network-Enabled Optimization System (NEOS) Server. Preprint MCS-P615-0996, Argonne National Laboratory, Argonne, Illinois, 1996.

[7] A. Downey. Predicting queue times on space-sharing parallel computers. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[8] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

[9] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY metacomputing experiment. *Concurrency: Practice & Experience*. to appear.

[10] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[11] GENIAS Software GmbH. CODINE: Computing in distributed networked environments, 1995. http://www.genias.de/genias/english/codine.html.

[12] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, 1994.

[13] The PSCHED API Working Group. PSCHED: An API for parallel job/resource management version 0.1, 1996. http://parallel.nas.nasa.gov/PSCHED/.

[14] R. Henderson and D. Tweten. Portable Batch System: External reference specification. Technical report, NASA Ames Research Center, 1996.

[15] International Business Machines Corporation, Kingston, NY. *IBM Load Leveler: User's Guide*, September 1993.

[16] J. Jones and C. Brickell. Second evaluation of job queuing/scheduling software: Phase 1 report. NAS Technical Report NAS-97-013, NASA Ames Research Center, Moffett Field, CA 94035-1000, 1997. http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-97-013/jms.eval.rep2.html.

[17] David A. Lifka. The ANL/IBM SP scheduling system. In *The IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 187–191, April 1995.

[18] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[19] P. Messina, S. Brunett, D. Davis, T. Gottschalk, D. Curkendall, L. Ekroot, and H. Siegel. Distributed interactive simulation for synthetic forces. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[20] K. Moore, G. Fagg, A. Geist, and J. Dongarra. Scalable networked information processing environment (SNIPE). In *Proceedings of Supercomputing '97*, 1997. http://scxy.tc.cornell.edu/sc97/proceedings/TECH/MOORE/INDEX.HTM.

[21] B. C. Neuman. Prospero: A tool for organizing internet resources. *Electronic Networking: Research, Applications, and Policy*, 2(1):30–37, Spring 1992.

[22] B. C. Neuman and S. Rao. The Prospero resource manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice & Experience*, 6(4):339–355, 1994.

[23] R. Ramamoorthi, A. Rifkin, B. Dimitrov, and K.M. Chandy. A general resource reservation framework for scientific computing. In *Scientific Computing in Object-Oriented Parallel Environments*, pages 283–290. Springer-Verlag, 1997.

[24] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. Technical report, 1997. Submitted.

[25] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating system services for wide area applications. Technical Report UCB CSD-97-938, U.C. Berkeley, 1997.

[26] J. Weissman. Gallop: The benefits of wide-area computing for parallel processing. Technical report, University of Texas at San Antonio, 1997.

[27] J. Weissman and A. Grimshaw. A federated model for scheduling in wide-area systems. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, 1996.

[28] S. Zhou. LSF: Load sharing in large-scale heterogeneous distributed systems. In *Proc. Workshop on Cluster Computing*, 1992.