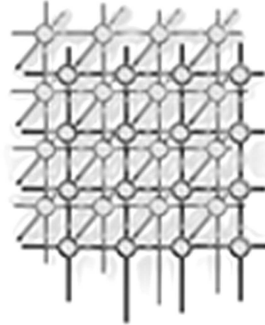


# Supporting Resource Reservation and Allocation for Unaware Applications in Grid Systems



Antonella Di Stefano<sup>1</sup>, Marco Fargetta<sup>1</sup>,  
Giuseppe Pappalardo<sup>2</sup>, and Emiliano Tramontana<sup>2</sup>

<sup>1</sup>*Dipartimento di Ingegneria Informatica e Telecomunicazioni*

<sup>2</sup>*Dipartimento di Matematica e Informatica  
Università di Catania, Italy*

---

## SUMMARY

The dynamics of resource request rates in a Grid system can be wide-ranging, to the point that request peaks for a single resource can be difficult to handle and end up greatly increasing the response time. Once a request has been submitted by a client, this has to cope with the potential overload showing up. However, it is impractical to cure overload once it takes place, by negotiating a different service or finding an equivalent resource, since the client would then bear the delay due to negotiation and re-submission anyway.

Instead, in order to effectively exploit Grid resources, it is crucial that applications perform reservation of resources before using them. Reservation allows a management system to consider application needs in advance and take suitable actions to improve resource availability. In particular, this enables additional resources to be secured beforehand in the background, if appropriate to avoid a potential overload, rather than paying the involved costs when overload arises.

This paper proposes a software architecture that integrates applications with Globus services, to conveniently handle resource reservation and allocation. Within the architecture, the computational reflection technology transparently connects applications with components that take care of advance reservation, as they dynamically sense the applications' resource needs. This dynamic information augments static knowledge gathered offline from static program code analysis.

KEY WORDS: Grid systems, reservation, computational reflection

---

\*Correspondence to: Emiliano Tramontana, Dipartimento di Matematica e Informatica, Università di Catania, Viale A. Doria 6, Catania I-95125, Italy. E-mail: tramontana@dmi.unict.it.



## 1. Introduction

Grid systems provide standard open protocols that enable the sharing of resources belonging to and controlled by different organisations. The Globus toolkit [20] is the most widespread implementation of Grid services and protocols and is currently used to support the major Grid projects [10].

In order to execute an application in a Globus environment, a user needs to find the appropriate resources and deploy the application accordingly. Currently, tools such as the *Resource Broker* for the European DataGrid [12] are employed to help finding the appropriate host for an application and, in some cases, deploying it. Resource allocation is possible through a software component called GRAM [5], which exploits the schedulers that resources themselves have to provide to make their allocation possible. Globus version 2 proposed a software component, called GARA [11], to handle resource reservation, but the latest distribution (version 3.2) does not include GARA anymore.

Resource reservation is crucial in environments as dynamic as Grid ones. The reservation phase provides some confidence that a following allocation request will succeed. Moreover, reservation prevents resources from becoming (over)loaded due to simultaneous requests from several applications. This could make it difficult or impossible for an application to satisfy its temporal constraints or even carry on.

This paper proposes overcoming the lack of reservation support in Globus by introducing appropriate components that automatically reserve the resources an application needs at run-time. For this purpose, a static analysis of the code of the application is performed, to estimate its resource needs. Moreover, the run-time behaviour of the application is monitored to continually adjust, accordingly, resource requests on its behalf. In order to implement the approach outlined, we introduce two main software support components. One of them, introduced in Section 2.1, enhances Globus with the capability of keeping track of all resource requests, so as to know whether a resource is going to be available for a given time. The other component, discussed in Section 5.1, aids a target application by dynamically reserving the needed resources in advance, and then allocating them in time for the application to use them.

In our previous work [7], we proposed a software architecture called STREGA (Support for Transparently handling REsources for Grid Applications), which smoothly handles, on behalf of applications, the phases of resource finding, application deployment, and resource reservation and allocation. In this paper, we show how the profile that characterises an application can be used to guide resource reservation before application execution; moreover, we further elaborate upon STREGA, detailing its components, and describing the additional support, (previously supplied by GARA and) currently unavailable in Globus, needed for Grid resources to be better exploited by jobs. In short, we strive to ensure the effectiveness of the proposed solutions in real-world applicative and architectural scenarios.

In this work, particular emphasis is placed upon the reservation of *computing* resources (i.e. CPUs), although we have applied the architecture and the policies proposed to *storage* (disk) and *network* (bandwidth) resources too. Indeed, it is a more challenging feat to reserve computing resources, for at least two reasons: (i) their preemptable nature (as opposed, e.g., to disk space), as well as (ii) the comparatively higher difficulty of predicting what requirements concerning them will actually arise from the run-time behaviour of applications. In any case,



in the sequel, in the description of the proposed architecture components, the term “resource” should often be understood, for the sake of precision, as “*computing* resource”.

STREGA components are designed to transparently integrate with an application to provide it with resource handling support. Integration is easily achieved, in two senses: (i) supporting components are capable of estimating the application’s needs, which relieves programmers and users from the burden of providing information otherwise necessary for integration; (ii) the application is not forced to be aware of STREGA, thanks to the connection mechanism employed. While the integrating architecture is not tailored to a specific programming language, our experiments have focused on estimating needs of Java applications.

The connection mechanism employed is based on the technology of *computational reflection* [18]. Through it, STREGA manages to seamlessly introduce Grid related concerns (such as resource finding, reservation and allocation) into an application that did not originally consider them, without requiring any intervention whatsoever on application source code. Moreover, integration between STREGA components and a conventional Grid system is performed automatically, and does not imply any additional effort in the deployment phase. Finally, it is worth noting that STREGA components are application-independent, hence easily reused.

As a result, applications to be integrated into a Grid infrastructure become no more difficult to develop than Grid-unaware ones. The effort of application developers is greatly reduced, for they are spared the burdens of: interfacing with libraries supporting interactions with Grid components, selecting proper types of hosts typically used in Grid systems, connecting with the communication infrastructure available, etc.

This paper is structured as follows. Next section introduces the Grid environment that we take as a reference in our experiments. Section 3 discusses the concept of computational reflection. Sections 4 and 5 describe the proposed reservation approach and its implementation with the STREGA software architecture. Related work is discussed in Section 6.

## 2. Reference Environment

Within the European DataGrid [12] (EDG) project, an architecture inspired by the “anatomy of the Grid” vision [17] has been developed to offer job and resource management services. These are based on, and use the services of, the Globus toolkit version 2.

The job management service provides EDG users with tools (such as shell scripts, web portals, etc.) to submit jobs to be executed. Typically, a user employs an access point, termed a *User Interface*, to send a *Resource Broker* (RB) the commands that will guide job submission. Every site which jobs can be submitted to is called a *Computing Element* (CE). Each CE holds a number of *worker nodes*, hosts where the job will ultimately run. It should be emphasised that, from outside, jobs can only be submitted to the CE as a whole, rather than to individual worker nodes, which are indeed not visible outside the CE.

The RB determines, among the CEs it knows about, those holding resources that match the job requirements. One of these CEs is then chosen, by either the RB or the user, and the job sent to it. Within the CE, the job is usually handled by a batch system, such as PBS (Portable Batch System) (see e.g. [6]), which assigns the job to a worker node as soon as one



is available. Within such batch systems, the job will then be the only one executing on that worker node, until termination: no concurrent activity by multiple jobs ever takes place on any worker node<sup>†</sup>. Moreover, regardless of its timing and computing requirements, a job cannot exploit the computing power of more than one worker. Indeed, in the EDG environment a job can parallelise its activity, splitting into multiple processes or threads, but only within the assigned worker node. On the other hand, it is possible to build a “composite job” as a collection of jobs, assigned each to a different host; in this solution, however, parallelism and distribution concerns, rather than being transparent for the application, must be tackled explicitly by it.

For our investigations and proposals we take as a basis the environment outlined above.

Although, as noted in [6, 1], batch schedulers like PBS or LSF are widely used to assign worker nodes to submitted jobs, we do not consider them flexible enough to support reservation or timing requirements for jobs. I.e., it is impossible to reserve a host for future use, whether before or during its executing an application. As a solution, we try to overcome batch scheduling limitations by giving worker nodes the capability of handling more than one job at a time, and enhancing the site (CE) level scheduler to exploit it.

For this purpose, the CE has to be endowed with suitable components aimed at implementing a dynamic, many-to-many mapping between jobs and worker nodes. Thus, mapping decisions may result, as deemed appropriate, in a job executing on multiple worker nodes<sup>‡</sup>, as well as multiple jobs executing simultaneously on a worker node. Consequently, we assume that the latter scenario is permitted by job management at the worker node level.

## 2.1. JAM — The Job Allocation Manager

The key component, among those to be introduced within a CE for job management, is *JAM*, the Job Allocation Manager responsible for (i) reservation of the CE’s worker nodes, and (ii) support for the concurrent execution of jobs within a node.

JAM deals with worker node reservation on the basis of the timing characteristics of jobs. I.e., JAM considers a worker node to be available when it either has not been reserved for some time, or has been reserved by jobs whose computational needs and timing requirements are compatible with the concurrent execution on it of an additional job.

For a new job submission on a CE, a reservation request must be issued to its JAM by a suitable tool, such as a command line on a shell or a web portal facility. If the reservation is accepted by JAM, a corresponding (worker node) allocation will follow at the proper time, meaning that the job will start executing on a worker node. Indeed, a reservation request to JAM carries the *starting time* from which and (optionally) the *timeframe* over which the allocated worker node should be used.

---

<sup>†</sup>Even if (all threads of) the current job were blocked (waiting for I/O to complete) on a worker node, no other job would be scheduled to run on it.

<sup>‡</sup>In [9] we have investigated automatic distribution support based on allocation policies that minimise the communication overhead among classes by avoiding spreading highly interactive ones over several hosts.



Comparing the timeframe in the request to the estimated worst case execution time (cf. Section 4.1), JAM may decide to grant the job extra worker nodes, should it reckon that one might not suffice to accommodate the timing requirement in the reservation request<sup>§</sup>. JAM's reservation policy is detailed in Section 4.2.

To guide allocation in such a way to implement its reservation decisions, JAM will set every worker node's local scheduler parameters so as to start or stop job execution at the appropriate times.

### 3. Computational Reflection

*Computational reflection* provides a software system with means to observe some of its own parts and perform operations on them [18]. A reflective object-oriented system usually consists of two, or more, levels. According to the *metaobject model*, which is the most widespread one, objects at the lower level, termed *baselevel*, are transparently observed and influenced by higher level objects, residing at the *metalevel*. These *metaobjects* (instances of a special class *Metaobject*) can modify the behaviour of their associated baselevel objects by *intercepting* operations on them, e.g. instantiations and invocations. The metaobject associated with an object is also capable of *inspecting* it, to retrieve its state and structure at run-time.

Reflection is effective in separating the development of parts of code of different nature, while deferring to run-time the necessary connection between them. Reflective systems have been proposed to separate core application functionalities from supplementary concerns such as synchronisation [21], distribution [9], etc.

Java supports inspection, while interception can be supplied by additional packages, such as Javassist [3]. We have selected the latter, for our goal of enhancing existing applications with resource reservation capabilities.

### 4. The Reservation Approach

We perform automatic reservation of Grid resources in two phases. The first phase takes place before an application is started, and is based on an a priori, static, "rough" characterisation of the application. This characterisation is provided through a distributed service, detailed in Section 4.1, which produces an application static profile (*ASP*) and a unique ASP identifier (*ASP-ID*) which allows the ASP to be retrieved afterwards. The ASP is computed just once for an application, and is referred to by JAM for reserving resources each time an application has to be executed.

The second phase consists of adapting reservation to the needs arising during application execution. Introducing run-time adjustments of the reserved resources aims at coping with the unavoidable uncertainty inherent to a static analysis, and the relatively unpredictable flow of control within the application.

---

<sup>§</sup>It is worth noting that this may bring about an intra-application distribution unanticipated at design time.



In the framework described, a user who wants to run an application has just to specify the *ASP-ID* and the desired job's *starting* and (optionally) *timeframe*, and send a request to JAM. JAM will take care of finding the available needed resources and performing the first phase of reservation accordingly.

#### 4.1. Computing the Application Static Profile

A user intending to submit a new application should first let the target CE obtain an estimate of the application's resource needs. For this purpose, we have made available a distributed *ASP evaluation service* across the Grid.

Currently, the ASP is stored within the host providing the evaluation service, but future versions of this service could exploit Java annotations to embed ASP data into classes as meta-data [14]. In the current version the outcome for the user is an ASP-ID to be used in the following execution requests. The JAM on the target CE will query the evaluation service to obtain details about the job, i.e. its ASP.

To find out what a Java application needs, the ASP evaluation service performs a static analysis of its bytecode, determining the following information.

**ASP-1** The list of run-time libraries used by the application. We inspect the libraries specified in `import` Java statements and check the parameters of `Class.forName()` methods<sup>¶</sup>.

**ASP-2** The list of threads, their mutual dependencies and the application classes implementing them. To obtain such a list, we check, for each class, whether it invokes methods `start()`, extends JDK class `Thread`, or implements JDK interface `Runnable`.

**ASP-3** For each thread and for the whole application, the estimated number of threads created inside its code. We count the statements starting new threads and weigh them according to the context where they appear (conditional statements, cycles, etc.), so as to consider whether actual parallelism between threads occurs at run-time. This provides an indication of the concurrent computations started, hence the amount of load generated.

**ASP-4** For each method, thread and for the whole application, an estimation of the respective degree of use of processor, disk and network.

**ASP-5** For each thread and for the whole application, an estimation of their worst case execution time (WCET) on a target host. Our approach to estimating the execution time of methods is based on that pioneered by Puschner and Koza [19]. We assume that loops have been appropriately annotated with an upper bound, and recursion is not used.

List ASP-1 above, which is related to static resources, will be needed to select worker nodes providing a run-time environment adequate for a given application. Items ASP2-5, instead,

---

<sup>¶</sup>However, some run-time linked classes could be overlooked by our analysis, e.g. those whose name is determined at run-time by a parameter being passed. Some adjustments are therefore introduced according to the actual execution (cf. Section 5).



are used for the sake of reservation and allocation of dynamic resources. ASP-5 is used if the user's reservation request omits a timeframe for the job.

We now detail how the estimated degrees of processor, disk and network use (ASP-4) are determined, for an application (and its parts), by a static analysis of its bytecode.

We begin by attributing a weight to all occurrences of: (i) JVM opcodes, (ii) method invocations to packages related to disk access, and (iii) method invocations to network-related packages.<sup>||</sup> The weight of an occurrence reflects, in any case, its context; e.g., weight is low within a `catch` block, but grows with the loop nesting level and annotated (cf. ASP-5) upper bound. Moreover, the weight of an opcode also reflects the associated JVM instruction complexity, and that of a disk or network method call depends on the nature of the containing application (e.g., a file method call gets a higher weight within a file management, rather than image processing, application).

For each method call, an aggregate  $op_{tot}$  is determined by adding up, for each instruction occurring in its body, a summand so (recursively) defined:

1. if the occurrence is of category (i), (ii) or (iii), the summand is its weight;
2. otherwise, the instruction must be a call to a method unrelated to disk or network, and the relevant summand is  $op_{tot}$  for this call.

Recursion in the second defining clause of  $op_{tot}$  will eventually terminate, for the whole application has been assumed to be recursion-free. A clarification is in order for this clause, when the call is to the method `start()` of a `Thread` object. Then  $op_{tot}$  for this call is computed as though it were a `run()` for that object.

Aggregate parameters  $op_{disk}$  and  $op_{net}$  are similarly defined, except that only category (ii) and (iii) are mentioned in the first defining clause above, respectively.

We also introduce parameters  $op_{tot}$ ,  $op_{disk}$  and  $op_{net}$  for the whole application (i.e., those computed for the call to `main()`), and for each of its threads (those computed for the `start()` call initiating the thread). For any method, thread and the application, we then set:

$$du = op_{disk}/op_{tot}, \quad nu = op_{net}/op_{tot}, \quad pu = 1 - du - nu,$$

which are intended to describe the use rate of disk, network and processor respectively. E.g.  $du$  represents how much disk operations weigh related to the total amount of computations carried out.

The above parameters have been successfully used in [8], to characterise the nature of a class, and select the most appropriate allocation policy. Of course, in general, estimations based on static code inspection can never thoroughly take into account the variability existing at run-time due to e.g. actual input data, method parameters or loop iterations. To overcome this uncertainty we shall introduce components that observe the behaviour of applications at run-time and take appropriate actions when estimated and observed values diverge (cf. Section 5).

---

<sup>||</sup>We are investigating the actual operations performed by some Java packages (such as `java.io` and `java.net`), so as to filter out the invocations that do not correspond to any activity on the disk or the network, respectively.



## 4.2. JAM's Reservation Policy for Worker Nodes

JAM's main task is to decide how to cope with reservation requests submitted by applications wishing to be executed, or already executing, on its site (CE). For a new job submission, JAM must choose whether to accept the request (in due time, this will lead to the job being allocated a worker node). In such a case the request had been issued with the estimated  $pu$  for the whole application (available in ASP-4, Section 4.1). Moreover, JAM can also receive, from a job already running on its site, a request to reserve resources needed for the execution of a new thread, which is also characterised by an estimated  $pu$  (cf. ASP-4). Details of the mechanisms whereby such requests are submitted to JAM are depicted in Figure 2 and discussed in Section 5.1. In any case, JAM's decision should be taken on the basis of both the estimated requested  $pu$  and the estimated workload on the worker node considered as a candidate for later allocation. JAM will evaluate, as described below, whether the requested use would be compatible with the estimated resulting workload for the relevant worker node.

To begin with, in an off-line measurement phase, we determine a load state in which the further execution of even a small job would take an excessive processing time. For this purpose, we run a tiny *sample job*  $SJ$  and measure its execution time when no other jobs are executing (i.e. when only the basic services are active), and as we progressively increase the load by executing a known *load job*  $LJ$  in a growing number of concurrent instances. Assume that, when this number reaches a threshold  $n2much$ , the execution time taken by  $SJ$  is considered to have become *inadmissible* (cf. the concept of *relaxation* [8]).

$LJ$  consists of an active object whose class has  $pu=1$ , i.e. is CPU-intensive enough that, whenever each new instance is started, response times increase significantly.

Suppose now that a node is running a set of jobs, and let  $PU$  denote the sum of their  $pus$ . The resulting workload will essentially be the same as that determined by  $PU$  load jobs  $LJ$  (each with  $pu=1$ ). Thus, another way to interpret the previous definitions is that if  $PU=n2much$ , an extra job would (essentially like  $SJ$ ) become inadmissibly slow. Also, under an (approximate) linearity assumption, if  $PU = \alpha \cdot n2much$  ( $0 \leq \alpha \leq 1$ ), the response time of an extra job would stay about  $\alpha$  times shorter than its inadmissibility threshold.

We can now formulate the reservation policy we adopt: a request to run a job in a given timeframe will be accepted unconditionally, if there are no reservations pending for that timeframe, or if the sum  $PU$  of their  $pus$  and the new request's does not exceed  $\frac{1}{2} \cdot n2much$ . I.e., a job request is accepted if the new job would run at least twice faster than (hence well clear of) its inadmissibility threshold. This provides enough leeway to accommodate extra resource needs arising at run-time (see below).

The described policy is summarised in Figure 1, where reservations are represented as gray dashed blocks, and the possibility to have extra jobs running in an interval is indicated by a white area.

At run-time, JAM notices the reservation requests that have been accepted without an ensuing allocation yet. These *pending* requests will be considered as expired (and eliminated) after a fixed validity interval  $\Delta t$ . In Figure 1, a gray dashed block becomes a white area when no allocation has been performed within  $\Delta t$  from its expected start time,

Reservation requests originating from jobs already running on the same worker node, and specifying a starting time very close to the current time, represent needs that become apparent

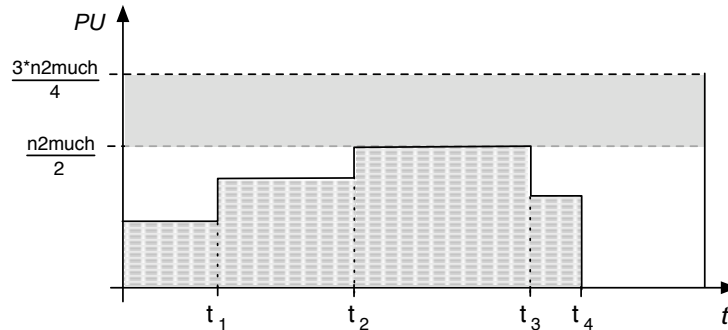


Figure 1. Reservation policy

only at run-time. These are considered as high priority requests, as far as possible, since they could give the job the opportunity to terminate successfully. Since such requests should be shortly followed by an allocation, they are evaluated against current (rather than estimated) workload conditions. A request is accepted, however high the requested job's  $pu$ , when the host workload is lower than  $\frac{1}{2} \cdot n2much$ , for the whole reservation interval. Instead, when the measured workload is higher than  $\frac{1}{2} \cdot n2much$ , a request is only accepted if its  $pu$  would not make the workload exceed  $\frac{3}{4} \cdot n2much$ , also counting all the pending requests during the reservation interval. Finally, no requests whatsoever is accepted when the measured workload exceeds the  $\frac{3}{4} \cdot n2much$  threshold.

## 5. The STREGA Architecture

The STREGA reflective architecture aims at transparently providing an application with reservation facilities. The STREGA *baselevel* consists of the application, which addresses *functional*, end-user issues like, e.g., simulation, raw data transformation, number crunching, etc. On the other hand, the STREGA *metalevel* monitors the application's run-time behaviour, checks its resource needs, and properly handles requests for resource reservation and allocation, by suitably interfacing with the Globus GRAM service.

The application is integrated with the metalevel on the worker node where it runs, just before execution, during class loading\*\*. After this integration, thanks to the reflective mechanisms exploited, the application, unaware of this at source-level, becomes transparently endowed with resource reservation capabilities.

---

\*\*This integration can be performed for good, on Grid sites where the application should be permanently available.

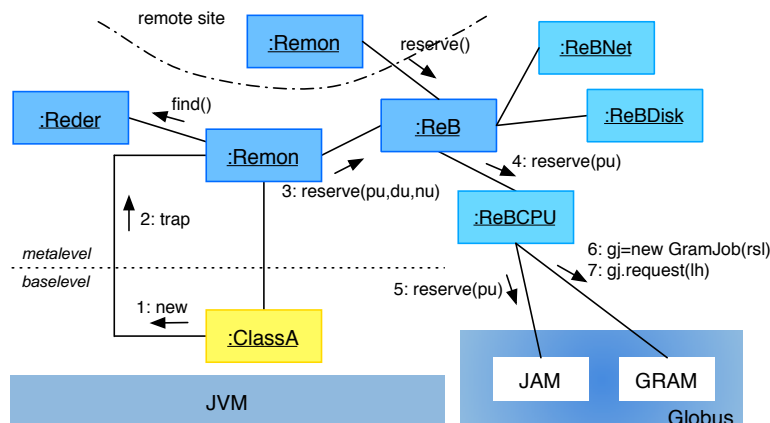


Figure 2. STREGA and Globus interactions

As outlined in Figure 2, STREGA relies mainly on the following metalevel classes.

**Resource Request Monitor (Remon)** monitors the threads spawned by the application, estimates the related resource needs, and dynamically requests resources to be reserved and allocated on behalf of the application, according to its observed run-time evolution.

**Reservation/Allocation Broker (ReB)** deals with resource reservation and allocation requests issued on behalf of various objects within the same application; **ReB** acts as a broker between **Remon**, and services JAM (the underlying resource reservation service described in Section 4.2) and GRAM (the Globus resource allocation service).

As we argue in [9] (in a context comparable to the present one), additional classes, besides the above metalevel ones, are in fact necessary to support distributed concerns such as, e.g., object and operation delivery. Especially noteworthy among them are **Locator** and **ServerProxy**, which are used respectively to maintain the location of distributed objects, and to receive data and commands from remote hosts where a part of the application classes execute.

A further useful component, called **ReDer**, was introduced in [7] to answer queries aimed at finding worker nodes that satisfy the application's static requirements. For this purpose **ReDer** relies on the indexing services available in Globus or over it.

Finally, it is worth noting that no metalevel support is needed for the sake of transparent user authorisation, which is performed automatically within the Globus toolkit. All a user has to do is initialise the Globus service responsible to certify his/her identity to resource managers throughout the execution of the application.

The overhead introduced by the proposed metalevel classes is estimated offline, employing the scheme described in Section 4.1. The initial amount of resources requested for an application is incremented in order to offset this (estimated) overhead.



---

STREGA metalevel classes **Remon** and **ReB** will now be described in greater detail.

### 5.1. Resource Request Monitor (**Remon**)

Metaobject **Remon** transforms estimations of resource needs into explicit requests to the other metalevel objects **Reder** and **ReB**.

**Remon** is reflectively associated with all<sup>††</sup> application classes that spawn a new thread at run-time (i.e. classes extending **Thread** or implementing *Runnable*). These classes are known from information ASP-2, Section 4.1, extracted from the bytecode.

When an object invokes **new** on such a class, so that a thread is spawned to run the newly created object, a need arises for more resources, for the CPU will be asked to operate in parallel for both the new and the invoking objects. Thus, it seems appropriate that, upon intercepting such a **new** (Figure 2 (1,2)), **Remon** should send **ReB** a reservation request for a computing resource (3).

As a strategy to avoid too frequent resource requests, those actually issued specify a larger amount of resources than strictly necessary. This resource surplus is proportional to the number of remaining instances that will be created subsequently by the class whose **new** has been intercepted (this information is available from ASP-3, extracted from the bytecode).

Although **Remon** requests resource reservation to **ReB** upon intercepting a **new**, it postpones resource allocation until **start()** is actually invoked on the new instance, provided of course that **ReB** has granted the requested reservation in the meantime.

Should **ReB** decide instead that no more requests can be accepted for the local worker node, **Remon** will take responsibility and try to secure the needed computing resources on other worker nodes. For this purpose, **Remon** will obtain, from **Reder**, the list of hosts suitable for the remote allocation of the object that the intercepted **new** intended to create.

### 5.2. Reservation/Allocation Broker (**ReB**)

Metalevel class **ReB** acts as a broker between the application's resource needs and the available resource reservation (JAM) and allocation (GRAM) services.

**ReB** handles reservation and allocation through its methods: **reserve(float pu, float du, float nu)** and **allocate(int id)**, respectively. The first method takes as input parameters the resource use rates characterising the captured **new** invocation (and determined via bytecode analysis, as ASP-4, Section 4.1), and returns an integer resource identifier. Later on, method **allocate(int id)** will take this identifier as its input parameter and use it to actually allocate the reserved resources.

**ReB** interacts with classes **ReBCPU**, **ReBDisk** and **ReBNet**, each providing a reservation and allocation policy customised for a resource category. Each of the latter three classes compares, on the one hand, the use rate and timing requirements already bearing on the respective

---

<sup>††</sup>The only exception is the class starting thread **main**, for which the initial resources are assumed to suffice.



resource with, on the other hand, the accumulated amount of requests performed by **Remon** while the application executes.

The task of **ReB** is to listen to every request from **Remon** and re-direct it to the appropriate resource handler class. As Figure 2 shows, classes **ReB**, **ReBCPU**, **ReBDisk** and **ReBNet** cooperate as in design pattern *Facade* [13]. This provides an interfacing class (**ReB**), which shields clients from directly accessing and knowing about many small classes, each implementing a specialised resource reservation policy.

**ReB**'s method `reserve()` invokes its namesake in each resource handling class, with the appropriate parameter characterising the resource category (e.g. `reserve(pu)` for class **ReBCPU**, see Figure 2, (3,4)). In turn, the latter class will pass the appropriate corresponding `reserve()` request to the underlying JAM reservation service ((5), Figure 2). When the reservation of all the resources requested by a `reserve()` invocation completes successfully, an identifier chain completes successfully, to allow the caller to later allocate the reserved resources.

**ReB**'s method `allocate()` will use the identifier returned by `reserve()` to ensure that the allocation of a reserved resource is performed just once. Moreover, it helps checking whether allocation is inappropriately attempted without a previous reservation.

We ultimately perform resource allocation relying on the services provided by GRAM. Method `allocate()` uses the CoG Java library [22] to access GRAM services through the mechanisms sketched in Figure 2 as Java statements (6,7). Our class **ReB** creates first a class **GramJob** instance, say `gj`, exploiting its constructor's RSL [5] string parameter to describe the resources requested; then **ReB** requests their allocation on its own (local) host `lh`, by invoking method `gj.request(lh)`.

As shown in Figure 2, **ReB** can receive reservation requests from **Remon** instances residing on both the local or a remote site. In the latter case, it is important to recognise whether it is the first time the remote **Remon** is trying to reserve resources for the distributed application at hand. If so, a supporting class **ServerProxy** (cf. [9]) has to be provided locally, to receive application data and commands from remote sites. As a result, the resources requested have to be incremented to account for those that will be exploited by **ServerProxy**. **ReB** will instantiate the new **ServerProxy** only if there are enough resources available locally for this purpose (otherwise the reservation request will be ignored on this site).

In the process of remote site selection, we strive to minimise the overhead introduced by the supporting metalevel classes, and for this we favour sites that have already received some application objects. As a side effect, this will increase the likelihood for interacting application objects to reside on the same site, thus reducing network communication.

## 6. Related Work

Jini is a technology providing services that can be used on a local network to publish, aggregate and discover application-related services [15]. To simplify the work of developers needing services made available through Jini, the RIO architecture has been proposed [16]. On a local network, RIO is meant to deploy components that encapsulate services, as well as provide support for activities such as life-cycle management and brokering for components, service monitoring and logging, etc. In order to use a RIO service, a client needs to submit to RIO



an XML document describing its requirements. RIO will find the best service, among the registered ones, with respect to the device providing it and the required quality of service (QoS). Then RIO will allocate the necessary software component into the selected device, and make it available to the client.

RIO requires applications to deal explicitly with QoS, by specifying the desired QoS parameters within service requests. STREGA, instead, ensures applications a satisfactory QoS indirectly and transparently, as a result of the transparent resource reservation support it provides. Moreover, RIO does not cater for reservation, so that resource overload may well occur at run-time. Finally, to exploit RIO services, applications have to be specifically designed to discover services, negotiate QoS parameters, etc. STREGA allows hosts, rather than services, to be transparently discovered at run-time and used by application classes.

Gridkit is a middleware that aims at providing applications with services for communication, and resource discovery and management [2]. There is support for the discovery and management of coarse-grained resources like CPU and storage, as in GRAM, as well as fine-grained resources like threads, buffers, etc. Gridkit resource management deals with the allocation and dynamic reconfiguration of resources for applications built out of OpenCOM [4] components, on the basis of a specification of the application and the QoS required by its components. The goal pursued is to find, for each application component, a node providing the necessary resources, and ensure that these will suffice at run-time in the face of changing application requests.

Although both STREGA and Gridkit aim at ensuring that at run-time the needed resources be available to application components, only STREGA performs its activities autonomously and transparently from applications. Within STREGA, developers need not specify information on applications and QoS attributes as in Gridkit, which greatly reduces their effort. Moreover, STREGA supports any Java application, whereas Gridkit only addresses those developed using the OpenCOM technology.

Finally, Gridkit resource management cannot guarantee that important requests will all be honoured, since multiple requests arising at run-time could overload resources, slowing down even crucial requests. In contrast, since STREGA uses reservation, requests that could result in overload later on are refused immediately, avoiding potential conflicts with existing executions.

## 7. Conclusions

In a Grid system, resource reservation is crucial to help coping with the uncertainty about the jobs submitted and the load rate of resources. In this work, a component supporting reservation has been designed that bases its decisions on the intrinsic characteristics of a job, statically estimated on program code (later corrected by dynamic monitoring). This frees users from the burden of estimating what their jobs need before attempting reservation.

A reflective software architecture called STREGA has been described, which transparently integrates jobs into a Grid environment, with the goal of sensing their implicit resource requests and perform run-time adjustments on resource reservation. This aims at curing the unavoidable uncertainty on the actual behaviour of jobs, and better following their time-varying needs.



Future work will tackle the scalability issues involved in the design of the proposed reservation supporting component.

## REFERENCES

1. A. Ali, A. Anjum, A. Mehmood, R. McClatchey, I. Willers, J. Bunn, H. Newman, M. Thomas, and C. Steenberg. A Taxonomy and Survey of Resource Planning and Reservation Systems for Grid Enabled Analysis Environment. In *Proceedings of the International Symposium on Distributed Computing and Applications to Business Engineering and Science*, 2004.
2. W. Cai, G. Coulson, P. Grace, G. S. Blair, L. Mathy, and W. K. Yeung. The Gridkit Distributed Resource Management Framework. In *Proceedings of the European Grid Conference*, Science Park Amsterdam, The Netherlands, February 2005.
3. S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of ECOOP*, volume 1850 of *LNCS*, 2000.
4. M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In R. Guerraoui, editor, *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 160–178. Springer, 2001.
5. K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of the IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.
6. D. De Roure, M. A. Baker, N. R. Jennings, and N. R. Shadbolt. The Evolution of the Grid. In F. Berman, G. Fox, and A. J. G. Hey, editors, *Grid Computing - Making the Global Infrastructure a Reality*. John Wiley and Sons Ltd, August 2002.
7. A. Di Stefano, M. Fargetta, G. Pappalardo, and E. Tramontana. STREGA: A Support for Transparently Handling Resources for Grid Applications. In *Proceedings of WETICE ETNGRID*. IEEE, June, 14-17 2004.
8. A. Di Stefano, L. Lo Bello, and E. Tramontana. Factors Affecting the Design of Load Balancing Algorithms in Distributed Systems. *Journal of Systems and Software*. Elsevier, 48, 1999.
9. A. Di Stefano, G. Pappalardo, and E. Tramontana. Introducing Distribution into Applications: a Reflective Approach for Transparency and Dynamic Fine-Grained Object Allocation. In *Proceedings of ISCC'02*, Taormina, Italy, 2002.
10. I. Foster and C. Kesselman, editors. *The Grid (2nd Edition): Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.
11. I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proceedings of IWQoS*, 1999.
12. F. Gagliardi, B. Jones, M. Reale, and S. Burke. European DataGrid Project: Experiences of Deploying a Large Scale Testbed for E-science Applications. *Lecture Notes in Computer Science*, 2459, 2002.
13. E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA, 1994.
14. Java Technology. Annotations. <http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>.
15. Jini Technology. Jini. <http://www.jini.org>.
16. Jini Technology. Rio. <http://rio.jini.org>.
17. C. Kesselman, I. Foster, and S. Tuecke. The Anatomy of the Grid - Enabling Scalable Virtual Organizations. *The International Journal of Supercomputer Applications and High Performance Computing*, May 2001.
18. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA*, volume 22 (12) of *Sigplan Notices*, Orlando, FA, 1987.
19. P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2):159–176, September 1989.
20. The Globus Alliance. Home page and publications. <http://www.globus.org>.
21. E. Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *LNCS*. Springer-Verlag, 2000.
22. G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.