# STREGA: A Support for Transparently Handling Resources for Grid Applications

Antonella Di Stefano, Marco Fargetta
Dip. di Ing. Informatica e Telecomunicazioni
Università di Catania, Catania, Italy
{adistefa, mfargetta}@diit.unict.it

Giuseppe Pappalardo, Emiliano Tramontana
Dipartimento di Matematica e Informatica
Università di Catania, Catania, Italy
{pappalardo, tramontana}@dmi.unict.it

## Abstract

*The Grid is a dynamic environment in which resources can quickly go from idle to busy state depending on application operations. In such a scenario, resources can be used more effectively by introducing reservation and allocation.*

*As a solution, this paper proposes STREGA: a software architecture that handles resource reservation and greatly simplifies the integration of applications with a Grid environment. In it, resources needed by applications are automatically detected, and operations such as resource reservation and allocation are accordingly transparently performed e.g. using Globus services.*

*Within STREGA, some components are aimed at understanding the needs of application classes, other components dynamically re-adapt resource requests on the basis of the observed application behaviour. Additional components are proposed to support reservation when this is unavailable from the underlying system (i.e. Globus and the OS).*

## 1. Introduction

Grid systems provide standard open protocols that enable sharing of resources controlled by different domains.

Executing a Grid application, once implemented and available, is not really smooth. It poses several difficulties related with operations typical of the Grid environment, such as deployment, resource reservation and allocation [8]. When submitting an application, users must pass an authentication phase [8, 10]. Moreover, users have to find the needed resources among those available on the Grid, such as a host holding the appropriate hardware and software, the repository providing the necessary input files, etc.

Once needed resources have been identified, the application has to be deployed accordingly. Although some software facilities are nowadays available, such as the *Resource Broker* for DataGrid [1], which help choosing the host where to run an application, their user is nevertheless still asked to write appropriate code describing the job and driving the matchmaking phase, which will determine the host where the application is deployed. In an alternative approach, based on services provided by a library such as e.g. CoG [16], an application has to include invocations to both find resources and deploy itself. No existing approach can be considered fully transparent.

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

The Globus toolkit is the most widespread implementation of Grid services and protocols and is currently used to support the major Grid projects [8]. Recent Globus extension proposals contain a software component, called GARA [9], which handles resource reservation. Currently in Globus, resource allocation is made possible by a software component called GRAM [4]. GRAM uses the schedulers provided by the resources to handle their allocation (suitable underlying schedulers are therefore needed).

The aim of this paper is twofold: firstly it proposes some software components that support the phases of resource finding, application deployment, and resource reservation and allocation; secondly, it shows how applications can be transparently provided with the proposed support components, by means of an integrating software architecture, which we call STREGA (Support for Transparently handling REsources for Grid Applications).

Our approach focuses on Java applications, but is easily adapted to other (possibly non object-oriented) contexts.

Integration is easily achieved thanks to two devices: (i) support components employed are capable of estimating the needs of an application, which relieves programmers or users from the burden of providing information necessary for integration; (ii) the connection mechanism used does not

force applications to be aware of the said support.

The connection mechanism exploited is based on *computational reflection* [13]. Thanks to the use of reflection in STREGA, we manage to introduce Grid related concerns (such as resource finding, reservation and allocation) into applications that do not consider such issues.

This paper is structured as follows. Next section introduces the concept of computational reflection. Section 3 describes the STREGA software architecture. Finally, conclusions are drawn in Section 4.

## 2. Computational Reflection

*Computational reflection* provides a software system with means to observe some of its own parts and perform operations on them [13]. 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. A metaobject associated with an object is also able to *inspect* the object to retrieve its state and structure at run-time.

Java supports inspection, while interception can be supplied by additional packages, such as Javassist [3].

Reflection is effective in separating the development of parts of code of different nature, while providing the necessary connection between them at run-time. Reflective systems have been proposed to separate typical application functionalities from supplementary concerns such as synchronisation [15], distribution [6], etc.

## 3. The STREGA Software Architecture

STREGA aims to transparently provide applications with reservation handling. STREGA baselevel can be identified with the application logic (addressing user-oriented issues like simulations, transformations of raw data, and other computations); whereas the metalevel handles Grid resource reservation and allocation by interfacing with Globus services, with the twofold goal of understanding which resources the application needs, and properly connecting application classes to Globus components.

As sketched in Figure 1, STREGA relies on the following metalevel classes to integrate applications with Globus.

**Resource Finder (Reder)** is responsible to find Grid resources satisfying specified criteria. For this it accepts a request for needed resources, produces the equivalent code describing the resources in an appropriate form, and finds them, relying on the indexing services available in Globus or over it.

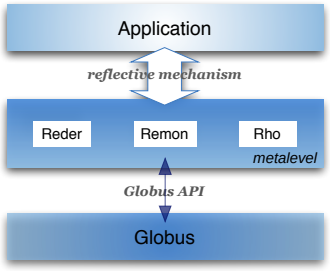


**Figure 1. STREGA overview**

**Resource Request Monitor (Remon)** understands the requirements of the application classes, and dynamically reserves and allocates resources on behalf of the application according to its run time evolution.

**Resource Holder (Rho)** takes into account resource reservation and allocation requests issued on behalf of various classes (not necessarily all within the same application). Reservation is performed by using Globus services when available, or ad hoc services (to be specifically implemented) otherwise.

As we argue in [6] (in a context different but in this respect comparable to the present one), besides the latter metalevel classes, additional ones are in fact necessary to support distributed concerns such as e.g. objects and operations delivering. Especially noteworthy among them are `Locator` and `ServerProxy`, which are used respectively to store the location of distributed objects, and to receive data and commands on the server side.

Let us note that the authorisation of a user who wants to work with Grid hosts is performed automatically within the Globus toolkit. A user needs only to initialise the Globus service responsible to certify his identity to resource managers throughout the execution of the application. Thus no metalevel support is needed for transparent authorisation.

STREGA metalevel classes will now be described in greater detail.

### 3.1. Resource Finder (Reder)

Usually, in a Grid environment, users, once logged in, need to find the appropriate *static resources* (i.e. hosts, operating system, run time libraries, input files, and/or applications) before they can execute their activity. Generally, as far as *dynamic resources*, such as host workload, available bandwidth, etc., are concerned, users cannot make assumptions on their dynamics or state, and are not enabled to check or trust their conditions.

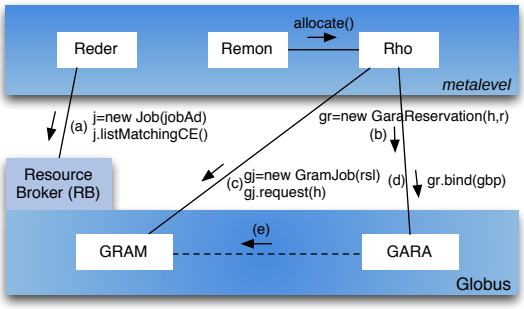


**Figure 2. STREGA and Globus interactions**

In STREGA, static resources are found, while the application executes and transparently for it, by class `Reder`. This class implements queries to a DataGrid facility, called *Resource Broker (RB)*, that, on the basis of the requirements presented to it and the resources it knows about, finds the most appropriate host. To this aim an RB exploits *Information Services*, which can be considered as repositories holding a list of available resources. Several RBs are available, each referring to groups of repositories. The querying process assumes that the reference to at least an RB is known. The RB concept is modelled after the *Matchmaker* [14].

Accordingly, class `Reder` is endowed with method `find(String spec)`, which takes a list of static resources as the input parameter `spec`; transforms it into the equivalent *Classified Advertisement (ClassAd)*[1] [14]; invokes the appropriate RB service; and finally returns a list of available hosts satisfying the request. Figure 2(a) shows the interactions between `Reder` and the RB via the CoG library. Results from RB are appropriately cached into `Reder` to avoid frequent and time consuming queries.

### 3.2. Resource Request Monitor (Remon)

The Grid is a dynamic environment in which resources can quickly go from idle to busy state depending on how applications use them. On the one hand, resources can be replaced or disconnected without advance warnings. On the other hand, due to the lack of precise knowledge about the applications that will be executed, it is difficult to forecast resource use beforehand. For these reasons, and in the absence of reservation mechanisms, a running application could experience difficulties when using a resource whose degree of utilisation varies. As a result it could become impossible for the application to, e.g., satisfy its temporal constraints or even carry on.

In STREGA, resource reservation and allocation are performed both when the application is started and while the application is running, so as to adapt to the needs it exhibits at run time. Thanks to the reflective approach, applications are provided with resource reservation and allocation, without forcing their programmers, nor their users, to explicitly handle these issues. The task of monitoring application classes and estimating their needed resources is entrusted to metaobject `Remon`. This metaobject is associated with each baselevel application class so as to intercept all operations and transform implicit needs into explicit resource reservation and allocation requests.

To find out what a class needs, `Remon` statically analyses its bytecode, determining the following data, as appropriate.

i. the list of run time libraries used by the class;

ii. the list of input files accessed by the class;

iii. an indication whether it extends the standard JDK class `Thread` or implements JDK interface `Runnable`;

iv. the estimated number of objects created inside its code;

v. an estimation of the degree of use of processor, disk and network that this class performs once instantiated.

Metaobject `Remon` transforms the estimation of needed resources into explicit requests to the other metalevel objects `Reder` and `Rho`. Lists (i) and (ii) above, being related to static resources, appear in requests to `Reder` to find a host. Items (iii), (iv) and (v) above, instead, are used in interactions with `Rho` for the sake of reservation and allocation of dynamic resources.

The estimated degree of processor, disk and network use are determined in advance for a class, by an analysis of its bytecode. Three totals called $op_{tot}$, $op_{disk}$ and $op_{net}$ are calculated by adding up all weighted occurrences of, respectively: all opcodes, method invocations to packages related to disk access, and method invocations to network-related packages. The weight given to each invocation opcode occurrence depends on the associated JVM instruction complexity, the nature of the application (e.g. a file copier application is given different weights than an image recognition application) and its context (including, e.g., the loop nesting level). Then we set:

$$du = \frac{op_{disk}}{op_{tot}} \qquad nu = \frac{op_{net}}{op_{tot}} \qquad pu = 1 - du - nu$$

which indicate disk, network and processor use respectively. These three parameters are intended to describe the use rate of the respective resource. E.g. $du$ represents how much disk operations are performed related to the total amount of computations carried out. The higher the value of $du$, the faster the disk needed for such a class.

[1] The ClassAd language is a data model that can be used to represent services and constraints.

In previous investigations [5], the above parameters have been successfully used to characterise the nature of a class, and understand the most appropriate allocation. Moreover, since they do not require substantial computations, they can be calculated at run time, just before an object is instantiated. In any case they need to be determined just once for each application class.

When `Remon` intercepts the creation of a new instance of an application class, it determines whether it will be invoked concurrently with other application threads (cf. indication (iii) extracted from the bytecode). If so, more resources are needed, since the CPU will be asked to operate for the invoking object and, in parallel, the newly created object. Thus, it seems appropriate that `Remon` should pass a new reservation request towards `Rho`. However, if an instantiation does not spawn a separate thread, resource reservation is only requested to `Rho` for class parameters that differ substantially from the average ones of previous instantiations.

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 (cf. indication (iv) extracted from the bytecode). When `Remon` intercepts an instantiation, it compares the actual use of resources with the already reserved ones, and decides whether a new reservation request to `Rho` is necessary. If so, resource allocation is postponed until `Rho` grants the requested reservation, This allows the execution of the object's constructor and subsequent method calls on it.

`Remon` handles the scenario in which resource reservation does not succeed, i.e. when `Rho` decides that no more requests can be accepted. In this case the needed resources have to be searched on other hosts. Then `Remon` obtains from `Reder` the list of capable hosts in order to start a phase that allocates objects remotely (see Section 3.4 for the description of interactions between metalevel objects).

### 3.3. Resource Holder (Rho)

Metalevel class `Rho` handles resource reservations and allocations 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 parameters characterising the class (those determined by `Remon`, cf. Section 3.2), and returns an integer resource identifier. The second method takes this identifier as input parameter and uses it to actually allocate the reserved resources.

We consider two reservation scenarios. The first applies when the underlying middleware provides the necessary support for reserving resources, which in Globus is represented by the GARA service. When GARA is available,

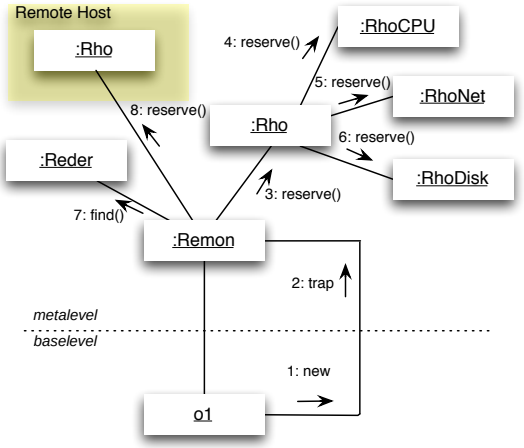


**Figure 3. STREGA class diagram**

we rely on its services and simply implement invocations to it inside `Rho`. It should be noted that, in such a case, GARA's reservation policy imposes some rules (e.g. what to reserve and how to ask for it) to our architectural component `Rho`. In our experiments, `Rho` accesses GARA for the sake of reservation through the Java CoG library. As Figure 2 shows, `Rho` uses constructor `GaraReservation(h,r)` to instantiate a new object of class `GaraReservation`, thus performing the reservation on host `h` of the resources described in `r` (see (b)).

In the second scenario, when GARA is unavailable within the underlying middleware, or its facilities are unsatisfactory, `Rho` handles reservation and allocation.

In such a scenario, `Rho` interacts with classes `RhoCPU`, `RhoDisk` and `RhoNet`, each providing a reservation and allocation policy customised for a resource category. The task of `Rho` is to intercept every request from `Remon` and re-direct it to the appropriate resource handler class. As Figure 3 shows, classes `Rho`, `RhoCPU`, `RhoDisk` and `RhoNet` cooperate as in design pattern *Facade* [11]. This provides an interfacing class (`Rho`) that shields clients from directly accessing many small classes, each implementing a specialised resource reservation policy. The interfacing class delegates the work to known classes when asked for a service. The main advantage of this design pattern is promoting weak coupling between clients and the small classes, thus enhancing reuse and evolution.

`Rho`'s method `reserve()` invokes each class with the appropriate parameter characterising the resource category (e.g. `pu` for class `RhoCPU`). When the reservation of all the resources is successful, an identifier is built and returned to the caller to allow subsequent allocation. Zero is returned if resource reservation could not be satisfied.

`Rho`'s method `allocate()` uses the provided identi-

fier to check that allocation for a reserved resource is performed just once. Moreover, it allows checking whether inappropriate allocation is attempted, e.g. by an object that is trying allocation without a previous reservation.

For both the reservation scenarios taken into account, i.e. with or without GARA support, we ultimately perform resource allocation relying on the services provided by GRAM. Method `allocate()` uses the CoG Java library to access GRAM services through the mechanisms sketched in Figure 2 as Java statements. When GARA is not used, because unavailable or unsuitable, then our class `Rho` creates an instance, say `gj`, of class `GramJob` (describing the resources to be requested based on the constructor's RSL [4] string parameter `rsl`), and then requests their allocation on host `h` by invoking method `gj.request(h)` (see Figure 2, (c)). In contrast, when GARA is used, class `Rho` simply invokes method `bind()` on the instance `gr`, used for the previous reservation (Figure 2, (b)), specifying as parameter `gbp` an instance of the Globus class `GaraBindParameters` (see (d)).

As shown in Figure 3, resource reservation requests to `Rho` can be received from `Remon` instances residing on the local host or on remote hosts. In the latter case, we have to distinguish whether `Remon` is trying to reserve resources for the first time for the distributed application at hand.

If so, some support (i.e. `ServerProxy`) has to be provided locally, to receive application data and commands from remote hosts. As a result, the resources requested have to be incremented to account for the resources that will be exploited by `ServerProxy`. When enough resources are available locally, then `Rho` instantiates the new `ServerProxy` and communicates the port that the remote `Remon` will use.

#### 3.3.1 Reserving CPU Power

Class `RhoCPU` estimates at run time whether more requests can still be accepted without overloading the CPU. It is difficult to calculate the load offered by a code fragment by simply analysing it statically, since many factors are uncertain. The actual load offered by a fragment of code, among other things, depends on: the flow of control inside it, the varying CPU use of different parts inside the fragment of code, the effect of simultaneously executing several fragments, the CPU type and the frequency of its cycles, the amount of available memory, etc. [12].

We consider whether to accept the reservation for executing a new instance of a class on the basis of the current workload of a host and $pu$ (see Section 3.2). We consider that the higher $pu$, the higher the workload of a CPU will be after execution of the corresponding instance.

In an off-line measurement phase, we determine the workloads of a host when (1) it is idle, and (2) has reached a stage after which further execution of even a small job takes an excessive processing time. These workloads will be referred to as $low$ and $max$, respectively. For determining the $low$ workload we run a sample algorithm that measures how fast the CPU is when no jobs are executing, i.e. when only the basic services are active (cf. the concept of *relaxation* [5]). For determining the $max$ workload we progressively increase the load by executing a known job an increasing number of times, until the execution time of the sample algorithm is considered too costly. The job we use to load the host consists of an active object whose class has a high $pu$. The number of times the job has to be executed to reach the $max$ workload is stored as $n2max$. Our sample algorithm executes in 10 milliseconds on a idle 2GHz Pentium processor with 512MB of RAM. We consider that such a host has reached the $max$ workload when the execution time of the sample algorithm is 100 times longer than on the idle host (1 second in our case).

We then calculate an intermediate threshold, called $mid$, as the mean value between the two previous thresholds. Finally a $high$ threshold is calculated as the mean value between the $mid$ and $max$ thresholds.

At run time, we periodically sense the actual host workload, and count reservation requests that have been accepted without an ensuing allocation yet. These *pending* requests are considered no more valid, and thus eliminated, after a fixed time-frame. A new reservation request is accepted for any value of $pu$, provided the measured workload is below the $mid$ threshold and the number of pending requests less than $n2max/2$. When the measured workload is between $mid$ and $high$ and the pending requests are less than $n2max/4$, we accept requests whose $pu$ is below 0.5. We do not accept any request when the workload exceeds the $high$ threshold.

Using the described algorithm, once reservation is performed, `RhoCPU` will avoid to accept requests that potentially overload the CPU.

#### 3.3.2 Reserving Network and Disk

Recently, some communication protocols have appeared that allow data transmission modes to be differentiated, in order to support non-trivial quality of service management [2]. In this context, resource reservation can support the achievement of a better communication service. For this purpose, rather than employing a reservation service constrained by a time-based approach [7], which assumes knowledge that we cannot extract from the class code (such as starting time and duration of the communication), we propose a simpler reservation policy, similar to that devised for CPU power, in Section 3.3.1.

I.e., in order to decide when the available bandwidth or disk can be further reserved on a host, appropriate thresh-

olds are determined off-line, and, while the application runs, class `RhoNet` and `RhoDisk` periodically monitor bandwidth and disk use, respectively. Requests to reserve network, or disk, are accepted when current use is compatible with the thresholds, the pending requests and the current request, along the lines adopted for CPU use.

### 3.4. Interactions between STREGA Components

Figure 3 illustrates how STREGA components interact. As soon as an application object performs an operation, e.g. instantiates a new object (see (1) in Figure 3), the associated metaobject `:Remon` intercepts the operation (see (2)) and checks whether enough resources (e.g. processing power) have been allocated so far. If not, reservation is required for such resources by means of `:Rho` (see (3)) (only one instance of this class is used for the whole host).

Object `:Remon` tries to reserve new resources, by invoking `:Rho`'s method `reserve()` and specifying the parameters $pu$, $du$ and $nu$ extracted from the class corresponding to the associated object. When GARA services are unavailable `:Rho` performs reservation by means of classes `:RhoCPU`, etc. (see (4, 5, 6) in Figure 3). When the reservation is successful, `:Rho` returns an identifier to allow subsequent allocation; otherwise `:Remon` tries to find appropriate resources on other hosts by using `:Reder`'s `find()` method (see (7)).

Object `:Reder` returns the list of the hosts providing the needed resources. The object to be allocated will be instantiated on the host where its reservation has been successful (see (8) in Figure 3). The first time a class has to be instantiated on a remote host, this is transferred there by using the class `ServerProxy`, which runs as an independent thread. Before executing the `ServerProxy`, an allocation request is sent to the GRAM of the remote host for both this server and the object itself.

Finally, it is worth recalling that whenever application objects on a host invoke a method of a remote object, such an invocation is delivered through the network to the remote class `ServerProxy`.

## 4. Conclusions

This work has proposed a reflective software architecture called STREGA, which transparently integrates applications into a Grid environment. Integration support is given by specialised components that transform the implicit resource requests by an application into explicit requests to its environment. The aim of the provided components is to find the appropriate resources, either local or distributed, and reserve and allocate them on behalf of the application as needed at run time. We considered both the scenarios where reservation is supported by the Globus GARA component and where GARA is unavailable.

## References


[1] World wide web home page for the European Datagrid project. http://www.eu-datagrid.org, 2004.

[2] L. Breslau, E. W. Knightly, S. Shenker, I. Stoica, and H. Zhang. Endpoint Admission Control:Architectural Issues and Performance. In *Proceedings of the ACM Sigcomm*, Stockholm, Sweden, 2000.

[3] S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of ECOOP*, volume 1850 of *LNCS*, 2000.

[4] 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.

[5] 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.

[6] 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.

[7] D. Ferrari, A. Gupta, and G. Ventre. Distributed advance reservation of real-time connections. *Springer-Verlag on Multimedia Systems*, 5(3), 1997.

[8] I. Foster and C. Kesselman, editors. *The Grid (2nd Edition): Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[9] 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.

[10] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proceedings of CCS*, 1998.

[11] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA, 1994.

[12] M. A. Iverson, F. Özgüner, and G. Follen. Run-time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing. In *Proceedings of HPDC*, 1996.

[13] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA*, volume 22 (12) of *Sigplan Notices*, Orlando, FA, 1987.

[14] R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2(2), 1999.

[15] 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*. 2000.

[16] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8–9), 2001.