Enhancing Java Grid Computing Security with Resource Control

Jarle Hulaas[1], Walter Binder[1], and Giovanna Di Marzo Serugendo[2]

[1] School of Computer and Communication Sciences
Swiss Federal Institute of Technology Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
{jarle.hulaas,walter.binder}@epfl.ch
[2] Computer Science Department, University of Geneva,
CH-1211 Geneva 4, Switzerland
Giovanna.Dimarzo@cui.unige.ch

Abstract. This paper outlines an original Computational Grid deployment protocol which is entirely based on Java, leveraging the portability of this language for distributing customized computations throughout large-scale heterogeneous networks. It describes practical solutions to the current weaknesses of Java in the fields of security and resource control. In particular, it shows how resource control can be put to work not only as basis for load balancing, but also to increase the security and general attractiveness of the underlying economic model.[3]

Keywords: Grid Computing, Resource Control, Mobile Code.

1 Introduction

Grid computing enables worldwide distributed computations involving multi-site collaboration, in order to benefit from the combined computing and storage power offered by large-scale networks. The way an application shall be distributed on a set of computers connected by a network depends on several factors.

First, it depends on the *application* itself, which may be not naturally distributed or on the contrary may have been engineered for Grid computing. A single run of the application may require a lot of computing power. The application is intended to run several times on different input *data*, or few times, but an a huge amount of data. The application has at its disposal computational, storage and network *resources*. They form a dynamic set of CPUs of different computing power, of memory stores (RAM and disks) of different sizes, and of bandwidths of different capacities. In addition, the basic characteristics of the available CPUs, memory stores and bandwidth are not granted during the whole computation (a disk with an initial capacity of 512MBytes when empty, cannot be considered having this capacity when it is half full). Code and data may be stored at different *locations*, and may be distributed across several databases.

[3] This work was partly financed by the Swiss National Science Foundation.

Computation itself may occur at one or more locations. Results of the computation have to be collected and combined into a coherent output, before being delivered to the client, who may wait for it at still another location. The network *topology* has also an influence on the feasibility of the distribution. Centralized topologies offer data consistency and coherence by centralizing the data at one place, security is more easily achieved since one host needs to be protected. However, these systems are exposed to lack of extensibility and fault-tolerance, due to the concentration of data and code to one location. On the contrary, a fully decentralized system will be easily extensible and fault-tolerant, but security and data coherence will be more difficult to achieve. A hybrid approach combining a set of servers, centralizing each several peers, but organized themselves in a decentralized network, provides the advantages of both topologies. Finally, *policies* have to be taken into account. They include clients and donators (providers) requirements, access control, accounting, and resource reservations.

Mobile agents constitute an appealing concept for deploying computations, since the responsibility for dispatching the program or for managing run-time tasks may be more efficiently performed by a mobile entity that rapidly places itself at strategic locations. However, relying completely on mobile agents for realizing the distribution complicates security tasks, and may incur additional network traffic.

This paper proposes a theoretical model combining the use of a trusted, stationary operator with mobile agents, running inside a secure Java-based kernel. The operator is responsible for centralizing client requests for customized computations, as well as security and billing tasks, and for dispatching the code on the Grid. We exploit the portability and networking capabilities of Java for providing simple mobile software packets, which are composed of a set of bytecode packages along with a configurable and serializable data object. We thus propose to use what is sometimes called $single-hop\ mobile\ agents$, which, compared to fully-fledged mobile agents, do not require heavy run-time support. These agents prevent the operator from becoming a bottleneck, by forwarding input code and data to computation locations and performing some management tasks. They start the different parts of the computations, ensure the management and monitoring of the distributed computations, and eventually collect and combine intermediate and final results.

The objective of this model is to propose a realistic deployment scenario, both from an economic and technical point of view, since we describe a setting where providers of computing resources (individuals or enterprises) may receive rewards in proportion to their service, and where issues like performance and security are addressed extensively, relying on actual tools and environments. While we put emphasis on being able to support *embarassingly parallel* computations, the model is sufficiently general to enable the distribution of many other kinds of applications.

Section 2 reviews distributed computations, Section 3 presents the model, Section 4 advocates the application of Java in Grid computing, whereas Section 5 describes the design and implementation of a secure execution environment,

which constitutes a first necessary step towards the full realization of the model. Finally, Section 6 summarizes some related approaches, before concluding.

2 Distributed Computations

Worldwide distributed computations range from parallelization of applications to more general Grid distributions.

2.1 Parallelization

Distribution of computing across multiple environments shares similarities with the parallelization of code on a multi-processor computer. We distinguish two cases, the first one corresponds to single instruction, multiple data (SIMD), while the second one corresponds to multiple instruction, multiple data (MIMD). Figure 1 shows both cases.

In case (a), the client's host ships the same code, but with a different accompanying data to multiple locations. After computation, the different results are sent back to the client's host. The final result is simply the collection of the different results. This kind of distribution is appropriate for intensive computing on a huge amount of the same type of data. It corresponds to the distribution realized by the SETI@home[4] experiment that uses Internet connected computers in the Search for Extraterrestrial Intelligence (SETI). Donators first download a free program. The execution of the program then downloads and analyzes radio telescope data. Note that in this case, the downloaded data may come from a different source.

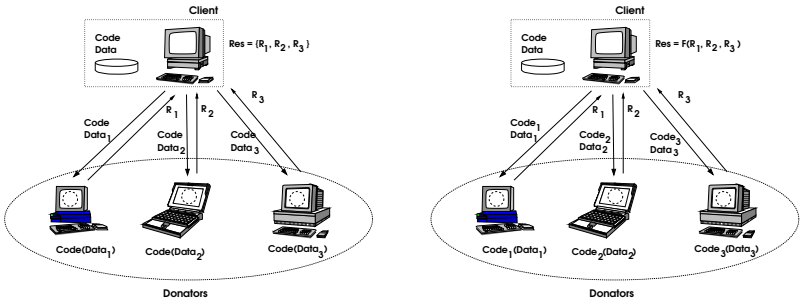
In case (b), code and data are split into several parts, then pairs of code and data are sent to several locations. The result is obtained by a combination (some function) of the different results.

Such a distribution is suitable for applications that can be divided into several pieces. This scheme fits the case of Parabon[5]. The client defines jobs to be performed. Transparently, the API divides the job into several tasks, on the client side; a task is made of a code, data, and some control messages. Tasks are sent to the Parabon server, which then forwards each task to a donator, using the donator's CPU idle time for computing the task. Once the task is achieved, the server sends back the result to the client, where the API then combines all results together, before presenting them to the client.

As a particular case of the MIMD example, code and data may be divided into several sequential parts. Computation would occur then in a pipeline-like style, where the next piece of code runs on the result of the previous computation.

These examples all exploit idle CPU time of the computer participating in the computations. The execution of the code on the data will ideally occur inside a secure "envelope", which ensures, on one hand, that the donator cannot exploit

[4] http://setiathome.ssl.berkeley.edu/
[5] http://www.parabon.com



(a) SIMD (b) MIMD

Fig. 1. Parallelisation

the code, the data and the results of the client; on the other hand, that the client does not execute malicious code in the donator's host. This is however not the case in practice, since current environments cannot provide such guarantees. The model proposed here at least partly addresses this issue (see Section 4).

2.2 Grid

The more general case of distributed computing is provided by the Grid computing concept which enables collaborative multi-site computation [11]. Grid computing goes beyond traditional examples of peer-to-peer computing, since there is a concern of proposing a shared infrastructure for direct access to storage and computing resources.

Figure 2 shows a generic Grid computation, encompassing the different classes of Grid applications [10]. The client, requesting the computation, the software to run, the data, and the results may be located at different sites. The data is even distributed across two databases. In this example, the code and the two pieces of data are moved to the donator's location, where the computation takes place. The result is then shipped to the client.

The CERN DataGrid [7] provides an example where physicists are geographically dispersed, and the huge amount of data they want to analyze are located worldwide.

3 Proposed Model

In this section we give an overview of our overall architecture, we outline our business model, and describe the different roles of participants in our Grid computing infrastructure, as well as their interactions.

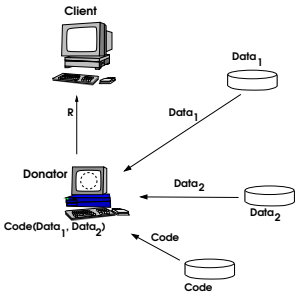


Fig. 2. Grid

3.1 Participating Parties in the Grid Computing Model

Our model involves 3 distinct parties: the *operator* of the Grid, *resource donators*, and *clients*. The operator is in charge of maintaining the Grid. With the aid of a mobile *deployment agent*, he coordinates the distribution of applications and of input data, as well as the collection and integration of computed results. The operator downloads the applications, and distributes them to resource donators that perform the actual computation.

Clients wishing to exploit the Grid for their applications have to register at a server of the operator before they are allowed to start computations. During the registration step, the necessary information for billing is transmitted to the operator. Afterwards the client is able to send a *deployment descriptor* to the operator.

The deployment descriptor comprises the necessary information allowing the operator to download the application, to prepare it for billing and accounting, and to distribute the application and its streams of input data to different active resource donators, taking into consideration their current load. The mobile deployment agent, which is created by the operator based on the contents of the client's deployment descriptor and coordinates the distributed client application, is not bound to a server of the operator; the client may specify the server to host the deployment agent, or decide to let the agent roam the Grid according to its own parameters. This approach improves scalability and ensures that the operator does not become a bottleneck, because the operator is able to offload deployment agents from his own computers.

Resource donators are users connected to the same network as the operator (e.g., the Internet) who offer their idle computing resources for the execution of parts of large-scale scientific applications. They may receive small payments for the utilization of their systems, or they may donate resources to certain kinds of applications (e.g., applications that are beneficial for the general public). Resource donators register at the Grid operator, too. They receive a dedicated execution environment to host uploaded applications. Portability, high performance, and security are key requirements for this execution platform. Section 4 gives detailed information on our platform, which is completely based on the

Java language. The operator dispatches downloaded applications to active resource donators. The deployment agent is in charge of supervising the flows of initial and intermediate data to and from the resource donators, as well as the final results, which are passed back to the destination designated by the client. Allowing the deployment agent to be moved to any machine on the Grid improves efficiency, as the deployment agent may locally access the required data there. As explained later, the deployment agent, or its clones, is also responsible for minimizing the flows of Grid management data between the donators and the operator.

3.2 Business Model

In our model the operator of the Grid acts as a trusted party, since he is responsible of all billing tasks[6]. On the one hand, clients pay the operator for the distributed execution of their application. On the other hand, the operator pays the resource donators for offering their idle computing resources.

The client buys *execution tickets* (special tokens) from the operator, which the deployment agent passes to the resource donators for their services. The resource donators redeem the received execution tickets at the operator. The execution tickets resemble a sort of currency valid only within the Grid, where the operator is the exclusive currency issuer. They enable micro-payments for the consumption of computing resources. There are 3 types of execution tickets: tickets for CPU utilization, for memory allocation, and for data transfer over the network. The coordinating deployment agent has to pass execution tickets of all types to a resource donator for exploiting his computing resources.

Execution tickets have to be protected from faking, e.g., by cryptographic means, and from duplication, as the operator keeps track of all tickets actually issued. Execution tickets can be distributed at a fine granularity. Hence, the loss of a single execution ticket (e.g., due to the crash of a resource donator) is not a significant problem. In case the deployment agent does not receive the desired service from a resource donator for a given execution ticket, it will report to the operator. If it turns out that a resource donator collects tickets without delivering the appropriate service, the operator may decide to remove him from the Grid. The detection of such malicious donators is possible by correlating the amount of requested tickets with the work actually performed, which is measured by CPU monitoring inside the dedicated execution environment.

3.3 Deployment of Applications

In order to start an application, the client transmits a deployment descriptor to the operator, who will retrieve and dispatch the application to different resource donators and also create a deployment agent for the coordination of the distributed execution of the application.

[6] The operator may also be responsible for guaranteeing that only applications corresponding to the legal or moral standards fixed by the donators are deployed.

The deployment descriptor, sent by the client, consists of the following elements:

- A description of the application's code location and structure. The client informs the operator of the application he wants to run. The operator will then download the application, and prepare it for resource control, before dispatching it to the donators. The application's structure establishes cut points and defines the different parts of the application that can run concurrently, as well as possible computational sequences. The client may specify himself the composition of computations, which reflects the calculus he desires to achieve (SIMD, MIMD, other). However, he does not customize the part of the description related to cut points, since it is tightly dependent on the application;
- A description of the source for input data. Usually, scientific applications have to process large streams of input data, which can be accessed e.g. from a web service provided by the client. The interface of this service is predefined by the operator and may support various communication protocols (e.g., RMI, CORBA, SOAP, etc.);
- A descriptor of the destination for output results. Again, this element designates the location of an appropriate service that can receive the results;
- Quality-of-service (QoS) parameters. The client may indicate the priority of the application, the desired execution rate, the number of redundant computations for each element of input data (to ensure the correctness of results), whether results have to be collected in-order or may be forwarded out-of-order to the client, etc. The QoS parameters allow the client to select an appropriate tradeoff between execution performance, costs, and reliability. The QoS parameters are essential to select the algorithms to be used by the deployment agent. For instance, if the client wishes in-order results, the deployment agent may have to buffer result data, in order to ensure the correct order.

In the following we summarize the various steps required to deploy a client application in the Grid. Figure 3 illustrates some of them.

1. Prospective resource donators and clients download and install the mobile code environment employed by the chosen operator, in order to be able to run the computations and/or to allow the execution of deployment agents.
2. Donators register with the operator and periodically renew their registration by telling how much they are willing to give in the immediate future; a calibration phase is initially run at each donator site to determine the local configuration (processor speed, available memory and disk space, quality and quantity of network access, etc.).
3. A client registers with the operator and sends the deployment descriptor (steps 1 and 2 of Figure 3).
4. The operator reads the deployment descriptor and:

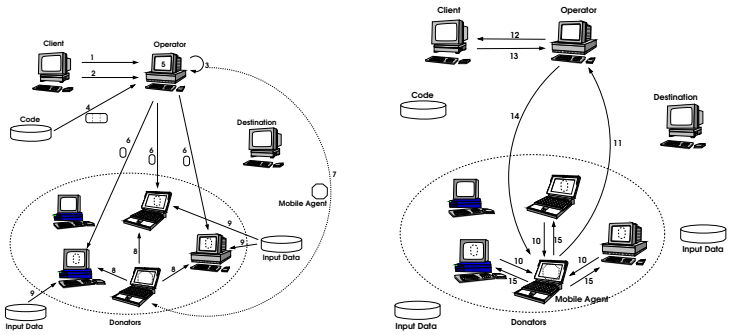
(a) Chooses an appropriate set of donators according to the required service level and to actually available resources; a micro-payment scheme is initiated, where fictive money is generated by the operator and will serve as authorization tokens for the client to ask donators for resources; a first wave of credit is transferred to the donator set, thus signifying that the corresponding amount of resources are reserved.
(b) Creates a mobile agent, the deployment agent, for coordinating the distribution, execution and termination of the client application (step 3); this deployment agent will shift the corresponding load from the operator to the place designated by the client, or to a donator chosen according to load balancing principles; the deployment agent may clone itself or move to the appropriate places for ensuring that input and output data is transferred optimally, thus avoiding useless bottlenecks at central places like the operator server.
(c) Downloads the client application (step 4) and rewrites it (reification of resources, step 5); the resulting code is signed to prevent tampering with it, and deployed directly from the operator's server (step 6).
(d) Dispatches the deployment agent to the appropriate initial place for execution (step 7).

5. The deployment agent launches the distributed computation by indicating (step 8) to each donator-side task where to locate its respective share of input data (step 9), and starts monitoring the computation.
6. The deployment agent acts as a relay between the operator and the donators. The agent receives regular status reports from the various locations of the resource-reified application (step 10); this enables him to monitor the progress of the computations, and to detect problems like crashes and to assist in the recovery (e.g., by preparing a fresh copy of the appropriate input data, or by finding a new donator to take over the corresponding task); the status reports are filtered and forwarded to the operator (step 11) in order to help maintaining a reasonably good view of the global situation (the operator might decide to schedule a second application on under-utilized donators); when necessary, the operator will ask the client for more credit (step 12), who will then buy more authorization tokens from the operator (step 13). The deployment agent then passes the execution tickets to the donators (steps 14 and 15)
7. When results have to be collected, the deployment agent may clone or migrate to the destination (step 16) and coordinate the incoming flows of data (step 17). He may perform a last filtering and integrity control of data before it is definitely stored.

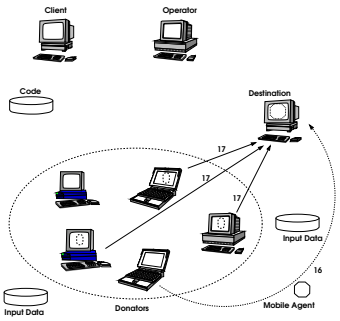
We favored this model over a completely decentralized peer-to-peer setting, since it simplifies the implementation of a global strategy for load balancing and ensures that some trusted party – the operator – can control the computations as they progress. In this approach, the operator also is in a natural position for managing all operations related to the validation of client-side payments and corresponding authorizations. Using mobile code for the deployment agent

ensures that the server of the operator does not become a bottleneck and a single point of failure. In the current model, the application code has to be transferred to the operator's computer, since it needs to be checked for security purposes (e.g., by code inspection), to be prepared for billing and accounting (using resource reification), and to be partitioned according to the deployment descriptor.

Currently, in order to simplify the deployment of the dedicated execution environment, resource reification is performed at the operator's site. However, this might also occur at the donator sites, at the expense of possibly transforming the same application in the same way on n sites, hence a waste of resources. Another disadvantage is that the donators would be in direct contact with the end-client: this hampers the transparency of our current proposal, and might have practical inconveniences by short-circuiting the trusted third-party that the operator implements (e.g., the application could no longer be verified and digitally signed by the operator, which has a recognized certificate).



(a) Deployment (b) Monitoring



(c) Results

Fig. 3. Application Distribution

4 Using Java for the Distribution of Computations

Here we motivate the use of Java for the implementation of distributed computations and their distribution within a network. In our model we use Java-based mobile agents for the distribution of deployment agents (to a server specified by the client) and of computational tasks (to resource donators). A secure Java-based kernel, the JavaGridKernel, serves as execution platform for deployed components in both cases. In that way, we leverage the benefits of Java and of mobile code, while at the same time offering enhanced security to protect hosts from faulty applications.

4.1 Why Java?

Recently, platforms for Grid computing have emerged that are implemented in Java. For instance, Parabon offers an infrastructure for Grid computing which is based completely on Java. In fact, the Java language offers several features that ease the development and deployment of a software environment for Grid computing. Its network-centric approach and its built-in support for mobile code enable the distribution of computational tasks to different computer platforms.

Java runtime systems are available for most hardware platforms and operating systems. Because of the heterogeneity of the hardware and of operating systems employed by Internet users, it is crucial that a platform for large-scale Grid computing be available for a large variety of different computer systems. Consequently, a Java-based platform potentially allows every computer in the Internet to be exploited for distributed, large-scale computations, while at the same time the maintenance costs for the platform are minimal ("write once, run everywhere").

Apart from its portability and compatibility, language safety and a sophisticated security model with flexible access control are further frequently cited advantages of Java. As security is of paramount importance for the acceptance of a platform for Grid computing, the security and safety features of Java are highly appreciated in this context.

4.2 Performance Issues

Java has its origins in the development of portable Internet applications. The first implementations of Java runtime systems were interpreters that inefficiently executed Java Virtual Machine (JVM) bytecode [16] on client machines. Also, several features of the Java programming language impact performance: the fact that it is a type safe, object-oriented, general-purpose programming language, with automatic memory management, and that its implementation does not directly support arrays of rank greater than one, means that its execution may be less efficient compared to more primitive or specialized languages like C and Fortran.

However, optimizations performed by current state-of-the-art Java runtime systems include the removal of array bounds checking, efficient runtime type

checking, method inlining, improved register allocation, and the removal of unnecessary synchronization code. See [15] for a survey of current compilation and optimization techniques that may boost the performance of Java runtime systems for scientific computing. In [17] the authors report that some Java applications already achieve 90% of the performance of equivalent compiled Fortran programs.

Considering the advantages of Java for the development and deployment of platforms for Grid computing, we think that a minor loss of performance can be accepted. Furthermore, the availability of more nodes where distributed computations can be carried out may often outweigh minor performance losses on each node. Ultimately, we are confident that maturing Java runtime systems will offer continuous performance improvements in the future.

4.3 Security Considerations

A high level of security is crucial for the acceptance of a platform for Grid computing. At first glance, Java runtime systems seem to offer comprehensive security features that meet the requirements of an execution environment for Grid computing: language safety, classloader namespaces, and access control based on dynamic stack introspection. Despite these advantages, current Java runtime systems are not able to protect the host from faulty (i.e., malicious or simply bugged) applications.

In the following we point out serious deficiencies of Java that may be exploited by malicious code to compromise the security and integrity of the platform (for further details, see [6]). Above all, Java is lacking a *task model* that could be used to completely isolate software components (applications and system services) from each other. A related problem is that, unfortunately, thread termination in Java is an inherently unsafe operation, which may e.g. leave shared objects, such as certain internals of the JVM, in an inconsistent state. Also related to the lack of task model is the absence of *accounting and control of resource consumption* (including but not limited to memory, CPU, threads, and network bandwidth). Concerning the implementation of current standard Java implementations, an issue is that several bytecode verifiers sometimes accept bytecode that does not represent a valid Java program: the result of the execution of such bytecode is undefined, and it may even compromise the integrity of the Java runtime system. Finally, while the security model of Java offers great flexibility in terms of implementing access control, it lacks central control: security checks are scattered throughout the classes, and it is next to impossible to determine with certainty whether a given application actually enforces a particular security policy.

All these shortcomings have to be considered in the design and implementation of Java-based platforms for Grid computing. Therefore, massive re-engineering efforts are needed to create sufficiently secure and reliable platforms.

5 The JavaGridKernel for the Secure Execution of Mobile Code

We have designed JavaGridKernel, a Java-based middleware that provides solutions to the security problems mentioned before and, hence, represents a state-of-the-art platform for the creation of secure environments for Grid computing. Several researchers have stressed the importance of multi-tasking features for Java-based middleware [2]. An abstraction similar to the *process* concept in operating systems is necessary in order to create secure execution environments for mobile code. However, many proposed solutions were either incomplete or required modifications of the Java runtime system.

In contrast, the JavaGridKernel has been designed to ensure important security guarantees without requiring any native code or modifications of the underlying Java implementation. The JavaGridKernel builds on the recent Java Isolation API [13], which offers the abstraction of *Isolates*, which fulfill a similar purpose as processes in operating systems and can be used to strongly protect Java components from each other, even within the same JVM. The Isolation API ensures that there is no sharing between different Isolates. Even static variables and class locks of system classes are not shared between Isolates in order to prevent unwanted side effects. Isolates cannot directly communicate object references by calling methods in each other, but have to resort to special communication links which allow to pass objects by deep copy. An Isolate can be terminated in a safe way, releasing all its resources without hampering any other isolate in the system. The Java Isolation API is supposed to be supported by future versions of the JDK. For the moment, it is necessary to resort to research JVMs that already provide the Isolation API, such as the MVM [8].

One crucial feature missing in Java is resource management, i.e., accounting and limiting the resource consumption (e.g., CPU and memory) of Java components. In the context of the JavaGridKernel, resource management is needed to prevent malicious or erroneous code from overusing the resources of the host where it has been deployed (e.g., denial-of-service attacks). Moreover, it enables the charging of clients for the consumption of their deployed applications. To address these issues, we have developed J-RAF2[7], The Java Resource Accounting Framework, Second Edition, which enables fully portable resource management in standard Java environments [5]. J-RAF2 transforms application classes and libraries, including the Java Development Kit, in order to expose details concerning their resource consumption during execution. J-RAF2 rewrites the bytecode of Java classes before they are loaded by the JVM. Currently, J-RAF2 addresses CPU, memory and network bandwidth control. For memory control, object allocations are intercepted in order to verify that no memory limit is exceeded. For CPU control, the number of executed bytecode instructions are counted and periodically the system checks whether a running thread exceeds its granted CPU quota. This implements a rate-based control policy; an additional upper hard limit on the total CPU consumed by any given computation can also be

[7] http://www.jraf2.org/

set and an associated overuse handler would then send an appropriate message to the deployment agent, to displace the task and hopefully prevent the loss of intermediate results. Control of network bandwidth is achieved by wrapping the standard input-output libraries of the JDK inside our own classes. J-RAF2 has been successfully tested in standard J2SE, J2ME, and J2EE environments. Due to special implementation techniques, execution time overhead for resource management is reasonably small, about 20–30%.

The Java Isolation API and J-RAF2 together provide the basis for the JavaGridKernel, which offers operating system-like features: Protection of components, safe communication, safe termination of components, and resource management. The JavaGridKernel extends these features with mechanisms to dynamically deploy, install, and monitor Java components. It should be noted that these mechanisms apply to the Java Virtual Machine and the associated executable bytecode representation: this approach is therefore in reality not restricted to programs written in the Java source language, but extends to all languages implemented on the JVM.

In the past we used the J-SEAL2 mobile object kernel [4] to implement an infrastructure for Grid computing. J-SEAL2 severely restricted the programming model of Java in order to enforce protection, safe communication, and termination of components. As the Java Isolation API provides these features in a standardized way without restricting the programming model, it is better suited for a Grid environment where also components designed without these J-SEAL2 specific restrictions in mind should be deployed.

The JavaGridKernel is perfectly suited for the development of platforms for Grid computing: It is small in size (only a few additional classes are required at run-time) and compatible with the Java 2 platform (but requires the Java Isolation API). Therefore, the distribution and installation of the kernel itself incurs only minimal overhead. The JavaGridKernel supports mobile code, which enables the distribution and remote maintenance of scientific applications. Finally, whereas scientific applications make heavy use of CPU and memory resources, the resource control features provided by J-RAF2 ensure a fair distribution of computational resources among multiple applications and prohibit an overloading of host machines. As explained below, resource control also provides additional security by thwarting dishonest behaviours on the donator side, thus making the model more attractive from an economic perspective.

The JavaGridKernel provides five special components: A mediator component to control the execution of uploaded applications, a network service to receive application code (Net-App service), a second network service allowing applications to receive input data and to transmit their results (Net-Data service), a system monitor to prevent an overloading of the machine, as well as a monitor window that displays information regarding the running applications, the elapsed time, etc. to the resource donator. In the following we give an overview of these components:

– The *mediator* is responsible for the installation and termination of applications, as well as for access and resource control. It utilizes the Net-App

service to receive control messages from the deployment agents that coordinate the distributed applications. It receives application archives, which contain the application code as well as a deployment descriptor. The deployment descriptor comprises a unique identifier of the application, as well as information concerning the resource limits and the priority of the application. The unique application identifier is needed for dispatching messages to the appropriate application. Requests to terminate an application are also received from the Net-App service. The mediator component ensures that applications employ only the Net-Data service and guarantees that an application only receives its own input data and that its output data is tagged by the application identifier. The mediator uses the system monitor in order to detect when the machine is busy; in this case, applications are suspended until the system monitor reports idle resources.

- The *Net-App service* is responsible for exchanging system messages with the coordinating deployment agent. When the platform is started, the Net-App service contacts the operator's server, which may transfer application archives to the platform. Optionally, a *persistency service* can be used to cache the code of applications that shall be executed for a longer period of time. The Net-App service also receives requests to terminate applications that are not needed anymore.
- The *Net-Data service* enables applications to receive input data and to deliver the results of their computation to the coordinating server. Messages are always tagged by an application identifier in order to associate them with an application. Access to the Net-Data service is verified by the mediator component. Frequently, continuous streams of data have to be processed by applications. The Net-Data service supports (limited) buffering of data to ensure that enough input data is available to running applications.
- The *system monitor* has to detect whether the machine is busy or idle. If the computer is busy, applications shall be suspended in order to avoid an overloading of the machine. If the computer is idle, applications shall be started or resumed. An implementation of the system monitor may employ information provided by the underlying operating system. However, such an approach compromises the full portability of all other components, since it relies on system-dependent information. Therefore, we follow a different approach: J-RAF2 enables the reification of the CPU consumption of applications [5], which allows to monitor the progress of applications. If the number of executed instructions is low (compared to the capacity of the hosting computer), even though applications are ready to run, the system monitor assumes that the computer is busy. Therefore, it contacts the mediator component in order to suspend computations. If, on the other hand, at the same time, requests for tickets originate from the same donator, it may be interpreted as malicious behaviour. Periodically, the system monitor resumes its activity in order to notice idle computing resources. When the computer becomes idle, all applications are resumed.
- The *monitoring window* presents information about the past and current work load of the system to the resource donator. It shows detailed status in-

formation of the running applications, the time elapsed for the computations, the estimated time until completion, if available, as well as some general information regarding the purpose of the computation. As the resource donator is in control of his system, it is important to show him detailed information of the utilization of his machine.

The mobile code execution environment for the deployment agents is based on the JavaGridKernel as well. But as the deployment agents stems from the operator, a trusted party, the security settings are relaxed. There are a few mandatory services needed by the deployment agent: access to the client web services that provide the input data and consume the output results, as well as network access for the communication with resource donators and the operator. Communication with the resource donators is necessary for the transmission of the application data, while communication with the operator is essential for the implementation of a global strategy for load balancing and for payment issues.

Regarding mobile code security, let us recall that the research community has not yet found a complete and general solution to the problem of malicious (donator) hosts, thus low-level tampering with Java bytecode (even cryptographically signed) or with the JVM is always possible if the attacker is sufficiently motivated, e.g., if a donator wants to steal results belonging to the client, or to hack the resource consumption mechanism in order to artificially increase his income. Our portable resource control mechanisms can nevertheless be exploited in several ways to detect such behaviours. First, if it is possible to determine statically the amount of resources required to achieve the given task, this detection will be trivial even across heterogenous machines, since J-RAF2 expresses CPU resource quantities in a portable unit of measurement (the bytecode). Second, it will always be possible to compare total consumptions at various donator machines and to correlate them with the size of the requested computations. The malicious host problem is however present in grid computing in general, not only with the mobile code approach proposed here.

6 Related Work

The primary purpose of mobile code is to distribute applications and services on heterogeneous networks. Many authors relate mobile code, and more often mobile agents as a practical technology for implementing load-balancing in wide-area networks like the Internet. Load-balancing can be either static (with single-hop agents, in the sense that once a task is assigned to a host, it does not move anymore) or dynamic (with multi-hop mobile agents enabling process migration). A survey of load-balancing systems with mobile agents is presented in [12]. Security and efficiency have immediately been recognized as crucial by the research community, but it was necessary to wait for technology to mature. Resource monitoring and control is needed for implementing load-balancing, and more generally for realizing secure and efficient systems, but is unavailable in standard Java, and particularly difficult to implement in a portable way. For

instance, Sumatra [1] is a distributed resource monitoring system based on a modified JVM called Komodo. See [5] for a further study on the portability of resource monitoring and control systems in Java.

According to [20], almost all Grid resource allocation and scheduling research follows one of two paradigms: centralized omnipotent resource control - which is not a scalable solution - or localized application control, which can lead to unstable resource assignments as "Grid-aware" applications adapt to compete for resources. Our primary goal is however not to pursue research on *G-Commerce* [20], even though we sketch an economical model based on virtual currency. For these reasons, our approach is hybrid. We relax the conservative, centralized resource control model by proposing an intermediary level with our deployment agents, designed to make the architecture more scalable. We have identified a similar notion of mobile coordination agent in [9], with the difference that our agents do not only implement application-level coordination (synchronization, collection of intermediate results), but also management-level activities (local collection and filtering of load-balancing data), following the general approach we exposed in [19]. As described in [18], control data generated by distributed resource control systems may be huge - and even higher in G-commerce systems, because of bidding and auctioning messages - and mobile agents may thus profitably be dispatched at the worker nodes for filtering the data flows at their source. We propose a further level of filtering to be accomplished by the deployment agents; this is even more necessary as we intend to control all three resources (CPU, memory and network). CPU is widely regarded as the most important factor. In [14] the authors propose to place worker agents within a Grid according not only to CPU load, but also to network bandwidth requirements; they relate a speed improvement of up to 40%, but the measurements were made in local-area clusters instead of dynamic sets of Internet hosts. Finally, memory control is usually ignored, but we contend that it has to be implemented in order to support typical scientific Grid computations, since they often imply storing and processing huge amounts of data.

Among the approaches that are not agent-based, the Globus initiative provides a complete toolkit addressing, among others, issues such as security, information discovery, resource management and portability. The Globus toolkit is being adopted as a standard by most multi-organisational Grids [10, 11]. The latest major version, Globus Toolkit 3, allows for the execution of Java code; it has a resource management facility, which is partly based on native code, and is thus not entirely portable. Resource management in Globus is not designed to be as accurate as provided by J-RAF2, and more specifically, resource accounting is not provided, which prohibits our fine-grained monitoring and incentive of usage-based payment for offered computing resources. As several aspects in Globus 3, the protection between jobs is biased towards the availability of the Unix kind of processes; this provides for good security, but is more expensive in memory space than Java isolates, which are designed for security without compromising the possibility of sharing Java bytecode between protection domains. Finally, the basic job deployment and coordination mechanisms of Globus are

not as flexible as the one permitted by the presently proposed mobile-agent based approach. These are a few aspects where we can propose some enhancements, but one should not be mistaken about the fact that Globus is an accomplished framework, whereas this paper essentially represents a theoretical work, based on a set of concrete building blocks.

Compared to a previous workshop position paper of ours [3] the model presented here relies on concepts and tools that are more mature and provide better guarantees of security and portability, while enabling a much more natural programming model than the one imposed by the J-SEAL2 mobile agent platform [4].

7 Conclusion

Our goal is to customize computations on open Internet Grids. To this end, we believe that a Grid environment should provide high-level primitives enabling the reuse and combination of existing programs and distributed collections of data, without forcing the client to dive into low-level programming details; the Unix scripting approach is our model, and this translates into our abstract *deployment descriptor* proposal. From the implementation point of view, this translates into a mobile *deployment agent*, which synthesizes and enhances the benefits of several previous approaches: the deployment agent optimizes its own placement on the Grid, and consequently it reduces the overall load by minimizing the communications needed for application-level as well as management-level coordination. There are of course still some open questions. The first pertains to the actual efficiency of the proposed model, which cannot be entirely determined before the complete implementation of the distributed control mechanisms. We have however tried to address the performance issue both at the host level, by proposing a solution which tries to minimize the management overhead, and at the global level, with a mobile agent-based approach which makes the whole system more scalable. The second concerns human factors such as validating the economical model (will it be attractive enough to generate real revenues?), or enabling the donator to decide on the lawfulness or ethics of computations submitted to him. This paper however concentrates on technological aspects, and claims that the comprehensive combination of a *pure Java* implementation enhanced with a secure, resource controlled execution platform is a unique asset for the portability, security and efficiency required for the success of Internet-based Grid computing.

References

1. A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for Resource-Aware mobile programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet, Second International Workshop*, volume 1222 of *LNCS*. Springer, July 1996.
2. G. Back and W. Hsieh. Drawing the red line in Java. In *Seventh IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, March 1999.

3. W. Binder, G. Di Marzo Serugendo, and J. Hulaas. Towards a Secure and Efficient Model for Grid Computing using Mobile Code. In *8th ECOOP Workshop on Mobile Object Systems, Malaga, Spain, June 10*, 2002.
4. Walter Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, January 2001.
5. Walter Binder, Jarle Hulaas, Alex Villazón, and Rory Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2001)*, Tampa Bay, Florida, USA, October 2001.
6. Walter Binder and Volker Roth. Secure mobile agent systems using Java: Where are we heading? In *Seventeenth ACM Symposium on Applied Computing (SAC-2002)*, Madrid, Spain, March 2002.
7. P. Cerello and al. Grid Activities in Alice. In *International Conference on Computing in High Energy Physics 2001 (CHEP'01)*, 2001.
8. Grzegorz Czajkowski and Laurent Daynes. Multitasking without compromise: A virtual machine evolution. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, USA, October 2001.
9. P. Evripidou, C. Panayiotou, G. Samaras, and E. Pitoura. The pacman metacomputer: Parallel computing with java mobile agents. *Future Generation Computer Systems Journal, Special Issue on Java in High Performance Computing*, 18(2):265–280, October 2001.
10. I. Foster and C. Kesselman. Computational Grids. In *The Grid: Blueprint for a Future Computing Infrastructure*, chapter 2. Morgan Kaufmann, 1999.
11. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid - Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.
12. J. Gomoluch and M. Schroeder. Information agents on the move: A survey on load-balancing with mobile agents. *Software Focus*, 2(2), 2001.
13. Java Community Process. JSR 121 – Application Isolation API Specification. Web pages at http://jcp.org/jsr/detail/121.jsp.
14. A. Keren and A. Barak. Adaptive placement of parallel java agents in a scalable computer cluster. In *Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, CA, USA, February 1998. ACM Press.
15. Andreas Krall and Philipp Tomsich. Java for large-scale scientific computations? In *Third International Conference on Large-Scale Scientific Computations (SCICOM-2001)*, Sozopol, Bulgaria, June 2001.
16. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
17. J. E. Moreira, S. P. Midkoff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.
18. O. Tomarchio, L. Vita, and A. Puliafito. Active monitoring in grid environments using mobile agent technology. In *2nd Workshop on Active Middleware Services (AMS'00) in HPDC-9*, August 2000.
19. A. Villazón and J. Hulaas. Active network service management based on meta-level architectures. In *Reflection and Software Engineering*, volume 1826 of *LNCS*, June 2000.
20. R. Wolski, S. Plank, T. Bryan, and J. Brevik. Analyzing Market-based Resource Allocation Strategies for the Computational Grid. *International Journal of High Performance Computing Applications*, 15(3), 2001.