
A survey on approaches to gridification



Cristian Mateos^{1,2}, Alejandro Zunino^{1,2,*},[†] and Marcelo Campo^{1,2}

¹*ISISTAN Research Institute, UNICEN, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina*

²*Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina*

SUMMARY

The Grid shows itself as a globally distributed computing environment, in which hardware and software resources are virtualized to transparently provide applications with vast capabilities. Just like the electrical power grid, the Grid aims at offering a powerful yet easy-to-use computing infrastructure to which applications can be easily ‘plugged’ and efficiently executed. Unfortunately, it is still very difficult to Grid-enable applications, since current tools force users to take into account many details when adapting applications to run on the Grid. In this paper, we survey some of the recent efforts in providing tools for easy gridification of applications and propose several taxonomies to identify approaches followed in the materialization of such tools. We conclude this paper by describing common features among the proposed approaches, and by pointing out open issues and future directions in the research and development of gridification methods. Copyright © 2007 John Wiley & Sons, Ltd.

Received 26 March 2007; Revised 29 June 2007; Accepted 4 July 2007

KEY WORDS: grid computing; grid development; gridification tools

1. INTRODUCTION

The term ‘Grid Computing’ came into daily usage about 10 years ago to describe a form of distributed computing in which hardware and software resources from dispersed sites are virtualized to provide applications with a single and powerful computing infrastructure [1]. This infrastructure, known as the *Grid*[‡] [2], is a distributed computing environment aimed at providing secure and coordinated computational resource sharing between organizations. Within the Grid, the use of

*Correspondence to: Alejandro Zunino, ISISTAN Research Institute, UNICEN, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina.

[†]E-mail: azunino@exa.unicen.edu.ar

[‡]Researchers commonly speak about ‘the Grid’ as a single entity, albeit the underlying concept can be applied to any Grid-like setting.

resources such as processing power, disk storage, applications and data, often spread across different physical locations and administrative domains, is shared and optimized through virtualization and collective management.

Grid infrastructures were originally intended to support computation-intensive, large-scale scientific problems and applications by linking supercomputing nodes [3]. During the first half of the 1990s, the inception and increasing popularity of Internet standards gave birth to an early phase of the Grid evolution, later known as Volunteer Computing [4]: users from all over the world are able to donate CPU cycles by running a free program that downloads and analyzes scientific data while their PCs are idle (e.g. when the screensaver is activated). Examples of these projects are Distributed.net [5] (Internet's first general-purpose distributed computing project), Folding@home [6] (protein folding), SETI@home [7] (search for extraterrestrial intelligence) and, more recently, Evolution@home (evolutionary biology) [8]. Few years after the introduction of Volunteer Computing, the first middlewares for implementing Grid systems over the Internet appeared. Examples are Legion [9], Condor [10] and Globus [11].

Nowadays, Grid Computing is far from only attracting the scientific community. Organizations of all types and sizes are becoming aware of the great opportunities this paradigm offers to share and exploit computational resources, such as information and services. In fact, a number of projects have been actively working towards providing an infrastructure for commercial and enterprise Grid settings [12–14]. Furthermore, many well-established standardization forums have produced the first global standards for the Grid. Recent results of these efforts include the Open Grid Services Architecture (OGSA) [15], a service-oriented Grid system architecture, and the Web Services Resource Framework (WSRF) [16], a framework for modeling and accessing Grid resources using Web services [17].

Although many technological changes in both software and hardware have occurred since the term 'Grid' was first introduced, a recent survey [18] indicates that there are hardly any significant disagreements within the Grid research community about the Grid vision. In fact, Ian Foster, considered by researchers to be the father of the Grid, proposed a checklist [19] for determining whether a system is a Grid or not, which has been broadly accepted.

Likewise, the basic Grid idea has not changed considerably within the past 10 years [18]. The term 'Grid' comes from an analogy with the electrical power grid. Essentially, the Grid aims at allowing users access computational resources as transparently and pervasively as electrical power is now consumed by appliances from a wall socket [20]. Indeed, one of the goals of Grid computing is to allow software developers to code an application (i.e. 'the appliance'), deploy it on the Grid (i.e. 'plug it') and then let the Grid to autonomously locate and utilize the necessary resources to execute the application. Ideally, it would be better to take *any* existing application and put it to work on the Grid, thus effortlessly taking advantage of Grid resources to improve performance. Sadly, the analogy does not completely hold yet, since it is hard to 'gridify' an application without manually rewriting or restructuring it to make it Grid-aware. Unlike the electrical power grid, which can be easily used in a plug-and-play fashion, the Grid is rather complex to use [21].

In this sense, the purpose of this paper is to summarize the state of the art on Grid development approaches, focusing specifically on those that target easy *gridification*, that is, the process of adapting an ordinary application to run on the Grid. It is worth mentioning that this paper does not exhaustively analyze the current technologies for implementing or deploying Grid applications. Instead, this paper discusses existing techniques to gridify software that has not been at first thought to be deployed on Grid settings, such as desktop applications or legacy code. In order to limit the

scope of the analysis, we will focus our discussion on the amount of effort each proposed approach demands from the user in terms of source code refactoring and modification. As a complement, for each approach, we will analyze the anatomy of applications after gridification and the kind of Grid resources they are capable of transparently leveraging.

The rest of the paper is organized as follows. The next section presents the most relevant related work. Section 3 briefly explains the anatomy of the Grid from a technical point of view. Section 4 discusses the evolution of gridification technologies. Section 5 surveys some of the most representative approaches for gridifying applications. Section 6 summarizes the main features of the surveyed approaches and proposes several taxonomies to capture a big picture of the area. On the basis of these taxonomies, Section 7 identifies common characteristics and trends. Section 8 presents the concluding remarks.

2. RELATED WORK

In [22], the authors point out the programming and deploying complexity inherent in Grid Computing. They state that there is a need for tools to allow application developers to easily write and run Grid-enabled applications, and propose OGSA as the reference Grid architecture towards the materialization of such tools. The authors also identify a taxonomy of Grid application-level tools that is representative enough for many projects in the Grid community. This taxonomy distinguishes between two classes of application-level tools for the Grid: programming models (i.e. tools that build on the Grid infrastructure and provide high-level programming abstractions) and execution environments (i.e. software tools into which users deploy their applications). The discussion is clearly focused on illustrating how these models and environments can be used to develop Grid applications from scratch, rather than gridify existing applications.

Another recent survey on Grid application programming tools can be found in [23]. Here, several functional and non-functional properties that a Grid programming environment should have are identified, and some tools based on these properties are reviewed. The survey concludes by deriving a generic architecture for building programming tools that are capable of addressing the whole set of properties, which prescribes a component-based approach for materializing both the runtime environment and the application layer of a Grid platform. However, the work does not discuss aspects related to gridification of existing applications either.

A survey on Grid technologies for wide-area distributed computing can be found in [24], where the most predominant trends for accelerating Grid application programming and deployment are identified. This work aims at providing an exhaustive list of Grid Computing projects ranging from programming models and middlewares to application-driven efforts, while our focus is exclusively on methods seeking to attain easy pluggability of conventional applications into the Grid. The authors emphasize on the need for a Grid framework that is adaptable and extensible enough to cope with the ‘waning star’ effect that has historically made predominant distributed computing technologies less popular. In other words, as Grid technologies evolve, this Grid framework should be able to evolve with them. A similar work is [25], in which a thorough examination of technologies for the materialization of Data Grids—those providing services and infrastructures to manage huge amounts of data—is presented. The survey compares Data Grids with other distributed data-intensive paradigms in great detail, and proposes various taxonomies to characterize the approaches that are currently being followed in the construction and materialization of Data Grids, focusing on aspects

such as data transport and replication, resource allocation and job scheduling. On the basis of this analysis, the authors identify scalability, interoperability and data maintainability as the requirements that still need to be properly addressed before Data Grids are massively adopted for developing large-scale, collaborative data-sharing and scientific applications. Finally, in [26], a taxonomy identifying architectural approaches followed in the implementation of resource management systems for the Grid is proposed. Roughly, the survey describes the common requirements for resource management systems and presents an abstract functional model, from where it derives the proposed taxonomy. The survey found that most approaches to Grid resource management are being developed in the context of computational and service-oriented Grids, but little research is being done in the context of Data Grids.

The next section explains the internal structure of the Grid as it is conceived today.

3. THE GRID: CONCEPTS AND ARCHITECTURE

A good starting point to understand the analogy with the goal of explaining between the Grid and the electrical power grid is GridCafé [27], a project from CERN[§] with the goal of explaining the basics of the Grid to a wider audience. Basically, GridCafé compares both infrastructures according to the following features:

- **Transparency:** The electrical power grid is transparent because users do not know how and from where the power they use is obtained. The Grid is also transparent, since Grid users execute applications without worrying about what computational resources are used to perform the computations, or where these resources are located.
- **Pervasiveness:** Electricity is available almost everywhere. The Grid is also pervasive, since according to the Grid vision computing resources and services will be accessible not only from PCs but also from laptops and mobile devices. Consequently, reusing existing pervasive infrastructures (e.g. the Internet) and ubiquitous Web technologies such as Web browsers, Java [28] and Web Services could be a big step towards complete pervasiveness and therefore easy adoption of the Grid.
- **Payment:** Grid resources are essential utilities, since they will be provided—just like electricity—on an on-demand and pay-per-use basis. The idea of billing users for the actual use of resources on the Grid finds its roots in an old computational business model called Utility Computing, also known as On-Demand Computing. A good example of a project actively working on utility-driven technologies for the Grid is Gridbus [29].

While the power grid infrastructure links together transmission lines and underground cables to provide users with electrical power, the Grid aims at using the Internet as the main carrier for connecting mainframes, servers and even PCs to provide scientists and application developers with a myriad of computational resources. From a software point of view, this support represents the bottommost layer of a software stack that is commonly used to describe the Grid in architectural terms. This architecture is depicted in Figure 1.

[§]The CERN (European Organization for Nuclear Research) is the world's largest particle physics laboratory, which has recently become a host for Grid Computing projects.

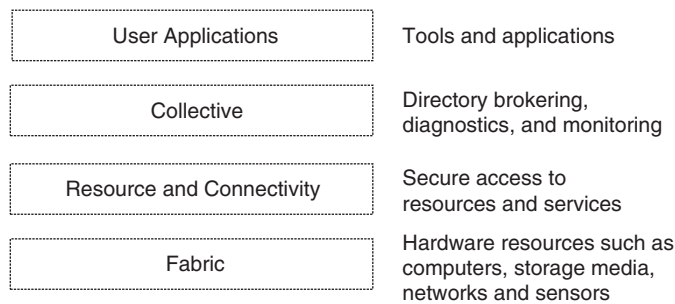


Figure 1. The Grid software stack [30].

The stack is composed of four layers: *Fabric*, *Resource and Connectivity*, *Collective* and *User Applications*. Roughly, the Resource and Connectivity layer consists of a set of protocols capable of being implemented on top of many resource types (e.g. TCP, HTTP). Resource types are defined at the Fabric layer, which in turn are used to construct metasevices at the Collective layer, and Grid applications at the User Applications layer. The main characteristics of each layer are described as follows:

- **Fabric:** As mentioned above, this layer represents the physical infrastructure of the Grid, including resources such as computing nodes and clusters, storage systems, communication networks, database systems and sensors, which are accessed by means of Grid protocols.
- **Resource and Connectivity:** Defines protocols to handle all Grid-specific transactions between different resources on the Grid. Protocols at this layer are further categorized as connectivity-related protocols, which enable the secure exchange of data between Fabric layer resources and perform user authentication, and resource-related protocols, which permit authenticated users to securely negotiate access to, interact with, control and monitor Fabric layer resources.
- **Collective:** The collective layer contains protocols and services associated with capturing interactions across collections of resources. Services offered at this layer include *directory* (discover resources by their attributes), *coallocation* (coordinated resource allocation), *job scheduling*, *resource brokering*, *monitoring* and *diagnosis* (detect and handle failures, overloads, etc.), and *data replication*.
- **User Applications:** Each of the previous layers exposes well-defined protocols and APIs that provide access to services for resource management, data access, resource discovery and interaction, and so on. On the other hand, the User Application layer comprises applications that operate within the Grid, which are built upon Grid services by means of those APIs and protocols.

It is worth noting that some ‘applications’ within the topmost layer may in turn be Grid programming facilities, such as frameworks and middlewares, exposing themselves to protocols and APIs on which more complex applications (e.g. workflow systems) are created. In fact, these facilities can be seen as the ‘wall socket’ by which applications are connected to the Grid. Application developers are likely to use high-level software tools that provide a convenient programming environment and isolate the complexities of the Grid, rather than use Grid services directly.

However, applications that have not been written to run on the Grid still have to be adapted in order to use the functionality provided by Grid programming facilities. In other words, these kinds of applications need to be gridified so that they can take advantage of Grid services and resources through a specific middleware or framework. As a consequence, an extra development effort is required from application programmers which might not have the necessary skills or expertise to port their applications to the Grid. To sum up, the foreseen goal of gridification is to let conventional applications benefit from Grid services without requiring these applications to be modified.

4. GRIDIFICATION TECHNOLOGIES: ORIGINS AND EVOLUTION

It is difficult to determine exactly when the term ‘gridification’ was first introduced, but the idea of achieving easy pluggability of ordinary applications into the Grid surely took a great impulse at the time the analogy between electrical power grids and computational Grids was established. Nowadays, the concept of gridification is widely recognized among the Grid research community, and many researchers explicitly use the term ‘gridification’ to refer to this idea. The evolution of Grid technologies from the point of view of gridification is presented in the following paragraphs.

The first attempt to achieve gridification began with the use of popular technologies traditionally employed in the area of Parallel and Distributed Computing, such as PVM [31], MPI [32] and RMI [33]. Basically, the underlying programming models of these technologies were reconsidered for use in Grid settings, yielding as a result standardized Grid programming APIs such as MPICH-G2 [34] (message passing) and GridRPC [35] (remote procedure call). Grid applications developed under these models are usually fragmented into ‘masters’ and ‘workers’ components communicating through ad hoc protocols and interaction mechanisms. Developers are also responsible for managing parallelization and location of application components. As a consequence, at this stage there was no clear idea of Grid resource *virtualization*. Consequently, gridification was mainly concerned with taking advantage of the Grid infrastructure, that is, the Fabric layer of the software stack shown in Figure 1.

The second phase of the evolution of gridification technologies involved the introduction of Grid middlewares. Some of them were initially focused on providing services for automating the scavenging of processing power, memory and storage resources (e.g. Condor, Legion), while others aimed at raising the level of abstraction of Grid functionality by providing *metaservices* (brokering, security, monitoring, etc.). A representative example of a middleware in this category is Globus, which has become the *de facto* standard for building Grid applications. Overall, users are now supplied with a concrete virtualization layer that isolates the complexities of the Grid by means of services. In fact, technologies such as MPICH-G2 and GridRPC are now seen as middleware-level services for communication rather than Grid programming facilities *per se*. Gridification is therefore conceived as the process of writing/modifying an application to utilize the various services provided by a specific Grid middleware. As the reader may observe, the main goal of gridification technologies at this stage is to materialize the middle layers of the Grid software stack.

The appearance of the first Grid middlewares was followed by the introduction of Grid programming toolkits and frameworks. In this step, the problem of writing applications for the Grid received more attention and the community recognized a common behavior shared by different Grid applications. The idea behind these technologies is to provide generic APIs and programming templates to unburden developers of the necessity of knowing the many particularities for contacting individual

Grid services (e.g. protocols and endpoints), of capturing common patterns of service composition (e.g. secure data transfer) and of offering convenient programming abstractions (e.g. master-worker templates). The most important contribution of these solutions is to capturing common Grid-dependent code and design in an application-independent manner. These tools can be seen as an incomplete application implementing non-application-specific functionality, with *hot-spots* or *slots* where programmers should put application-specific functionality in order to build complete applications [36,37].

For example, the Java CoG Kit [38] provides an object-oriented, framework-based interface to Globus-specific services. The Grid Application Toolkit (GAT) [39,40] and SAGA [41] are similar to the Java CoG Kit, but they offer APIs for using Grid services that are independent of the underlying Grid middleware. With respect to template-based Grid frameworks, some examples are MW [42], AMWAT [43] and JaSkel [44]. All in all, the goal of these tools is to make Grid programming easier. The conception of gridification at this phase does not change too much from that of the previous one, but Grid programming is certainly done at a higher level of abstraction. As a consequence, less design, code, effort and time are required when using these tools.

Up to this point, the most remarkable characteristic shared among the above technologies is that gridification is done in a *one-step* fashion, that is, there is no clear separation between the tasks of writing the pure functional code of an application and adding its Grid concerns. The Grid technology being used plays a central role during the entire Grid application development process, since developers Grid-enable applications as they code them by keeping in mind a specific Grid middleware, toolkit or framework. Therefore, technologies promoting *one-step* gridification assume that developers have a solid knowledge on Grid programming and runtime facilities.

Alternatively, there are currently a number of Grid projects promoting what we might call a *two-step* gridification methodology, which is intended to support users having little or even no background on Grid technologies. Basically, the ultimate goal of this line of research is to come out with methods that allow developers to focus first on implementing and testing the functional code of their applications, and *then* on automatically Grid-enabling them. As a consequence, this approach is suited for gridifying applications that were not initially designed to run on the Grid. It is worth noting that technologies under this gridification paradigm can be seen as a complement to the ones described previously. In fact, active research is being done to develop more usable and intuitive Grid programming models, toolkits and middlewares.

Figure 2 shows how the evolution of Grid technologies has reduced the knowledge that is necessary to gridify an application. As depicted in the figure, we identify four separate phases in this evolution. Transition between two consecutive phases is given by a radical change in the conception of the notion of gridification. In the first phase, 'gridify' means to manually use the Grid infrastructure. In the second phase, where virtualization of Grid resources through services was introduced, 'gridify' refers to adapt applications to using Grid services. The third phase witnessed the introduction of the first Grid development technologies materializing the common behavior of Grid applications; therefore, gridification takes place at a higher level of abstraction. Finally, the fourth phase incorporated the notion of two-step gridification: Grid technologies recognized the need to provide methods to *transform* ordinary applications to Grid-aware ones with little effort.

Certainly, the relationship between the two axes is not linear, but it is descriptive enough to get an idea about the consequences of gridification in the long term. As the reader can see, the *ideal* method for gridification would yield a hypothetical value for Grid awareness equal to *zero*, that is, the situation in which developers can effectively exploit the Grid without explicitly using any

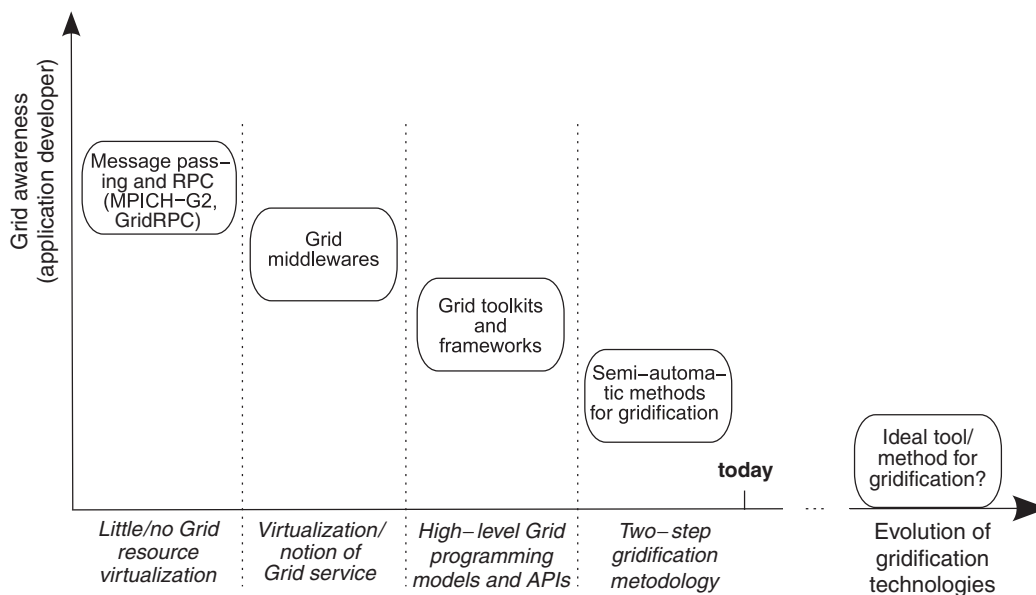


Figure 2. Origins and evolution of gridification technologies.

Grid technology in their code. In this paper, we are interested in reviewing the existing approaches that are focused on supporting two-step gridification. The next section discusses the most relevant projects for the purpose of this article.

5. GRIDIFICATION PROJECTS

In light of the gridification problem, a number of studies have proposed solutions to port existing software to the Grid. For example, Ho *et al.* [45] presented an approach to assist users in gridifying complex engineering design problems, such as aerodynamic wing design. Similarly, Wang *et al.* [46] introduced a scheme of gridification specially tailored to gridify scientific legacy code. In addition, Kolano [47] proposed an OGSA-compliant *naturalization*[¶] service for the Globus platform that automatically detects and resolves software dependencies (e.g. executables, system libraries, Java classes, among others) when running CPU-intensive jobs on the Grid.

Although the above technologies explicitly address the problem of achieving easy gridification, they belong to what we might identify as early efforts in the development of true gridification methods, which are characterized by solutions lacking generality and targeting a particular application type or domain. Nonetheless, there are a number of projects attempting to provide more generic, semi-automatic methods to gridify a broader range of Grid applications, mostly in the form

[¶]The American Heritage Dictionary defines naturalization as 'adapting or acclimating (a plant or an animal) to a new environment; introducing and establishing as if native'.

of sophisticated programming and runtime environments. In this sense, Sections 5.1–5.10 present some of these projects.

5.1. GEMMLCA

GEMMLCA (Grid Execution Management for Legacy Code Architecture) [48] is a general architecture for transforming legacy applications to Grid services without the need for code modification. GEMMLCA allows users to deploy a legacy program written in any programming language as an OGSA-compliant service. The access point for a client to GEMMLCA is a front end offering services for gridifying legacy applications, and also for invoking and checking the status of running Grid services. An interesting feature of this front end is that it is fully integrated with the P-GRADE [49] workflow-oriented Grid portal, thus allowing the creation of complex workflows where tasks are actually gridified legacy applications.

GEMMLCA aims at providing an infrastructure to deploy legacy applications as Grid services without re-engineering their source code. As depicted in Figure 3, GEMMLCA is composed of four layers:

- *Compute Servers*: Represents hardware resources, such as servers, PCs and clusters, on which legacy applications in the form of binary executables are potentially available. Basically, the goal of GEMMLCA is to make these applications accessible through Web Services-enabled Grid services.
- *Grid Host Environment*: Implements a service-oriented Grid layer on top of a specific OGSA-compliant Grid middleware. Current distributions of GEMMLCA support Globus version 3.X and 4.X.

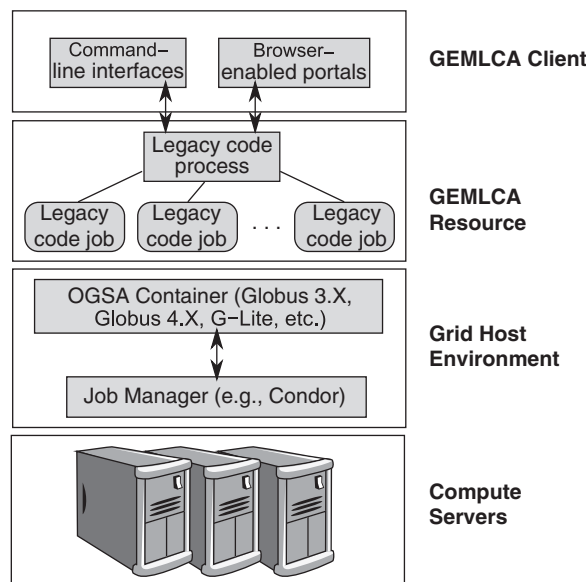


Figure 3. Overview of GEMMLCA.

- *GEMLCA Resource*: Provides portal services for gridifying existing legacy applications.
- *GEMLCA Client*: This layer comprises the client-side software (i.e. command-line interfaces and browser-enabled portals) by which users may access GEMLCA services.

The gridification scheme of GEMLCA assumes that all legacy applications are binary executable codes compiled for a particular target platform and running on a Compute Server. The Resource layer is responsible for hiding the native nature of a legacy application by wrapping it with a Grid service, and processing service requests coming from users. It is up to the user, however, to describe the execution environment and the parameter information of the legacy application. This is done by configuring an XML-based file called LCID (Legacy Code Interface Description), which is used by the GEMLCA Resource layer to map Grid service requests to job submissions. LCID files provide metadata about the application, such as its executable binary path, the job manager and the minimum/maximum number of processors to be used, and parameter information, given by the name, type (input or output), order, regular expressions for input validation, etc. The following code presents the LCID file corresponding to the gridification of the Unix *mkdir* command:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE GLCEnvironment "gemlcaconfig.dtd">
<GLCEnvironment id="mkdir"
  executable="/bin/mkdir" jobManager="Condor"
  maximumJob="5" minimumProcessors="1">
  <Description>Unix mkdir command</Description>
  <GLCParameters>
    <Parameter name="-p" friendlyName="New_folder"
      inputOutput="Input" order="0" mandatory="No">
      <initialValue />
    </Parameter>
  </GLCParameters>
</GLCEnvironment>
```

As explained, the GEMLCA gridification process demands zero coding effort and little configuration from the user. In spite of this fact, users not having an in-depth knowledge about GEMLCA concepts may experience difficulties when manually specifying LCID files. In this sense, the GEMLCA front end also provides user-friendly Web interfaces to easily describe and deploy legacy applications.

A more serious problem of GEMLCA is concerned with the anatomy of a gridified application. GEMLCA applications are essentially an ordinary executable file wrapped with an OGSA service interface. GEMLCA services serve request according to a very non-granular execution scheme (i.e. running the same binary code on one or more processors), but no internal changes are made in the wrapped applications. As a consequence, the parallelism cannot be controlled in a more grained manner. For many applications, this capability is crucial to achieve good performance.

5.2. GrADS

GrADS (Grid Application Development Software) [50] is a performance-oriented middleware with the goal of optimizing the execution of numerical applications written in C on distributed heterogeneous environments. GrADS puts a strong emphasis on application mobility and scheduling issues in order to optimize application performance and resource usage. Platform-level mobility in GrADS is performed through the so-called *Rescheduler*, which periodically evaluates the performance gains that can potentially be obtained by migrating applications to underloaded resources. This mechanism is known as *opportunistic migration*.

Users wanting to execute an application contact the GrADS *Application Manager*. This, in turn, contacts the *Resource Selector*, which accesses the Globus MDS service to obtain the available list of computing nodes and then uses the NWS (Network Weather Service) [51] to obtain the runtime information (CPU load, free memory and disk space, etc.) from each of these nodes. This information, along with execution parameters and a user-generated execution model for the application, is passed on to the *Performance Modeler*, which evaluates whether the discovered resources are enough to achieve good performance or not. If the evaluation yields a positive result, the *Application Launcher* starts the execution of the application using Globus job management services. Running jobs can be suspended or canceled at any time due to external events such as user intervention.

GrADS provides a user-level C library called SRS (Stop Restart Software) that offers applications functionality for stopping at a certain point of their execution, restarting from a previous point of execution and performing variable checkpointing. To SRS-enable an ordinary application, users have to manually insert instructions into the application source code in order to make calls to the SRS library functions. Unfortunately, SRS is implemented on top of MPI; hence, it can only be used in MPI-based applications. Nonetheless, as these applications are composed of a number of independent, mobile communicating components, they are more granular, thus potentially achieving better use of distributed resources than conventional GrADS applications—that is, without using SRS.

5.3. GRASG

GRASG (Gridify and Running Applications on Service-oriented Grids) [52] is a framework for gridifying applications as Web Services with relatively little effort. Also, in order to make better use of Grid resources, GRASG provides a scheduling mechanism that is able to schedule jobs accessible through Web Services protocols. Basically, GRASG provides services for job execution, monitoring and resource discovery that enhance those offered by Globus.

The architecture of GRASG is depicted in Figure 4. Its main components are four Web Services named Information Service (IS), Resource Allocation and Scheduling Service (RASS), Job Execution Service (JES) and Data Service (DS). Each Grid resource (i.e. a server) is equipped with the so-called *sensors* and wrapped with a JES. Sensors are responsible for capturing and publishing meta-information about their hosting resource (platform type, number of processors, installed applications, workload, etc.), while JES services are responsible for job execution and guaranteeing Quality of Service. More important, a JES wraps all the (gridified) applications installed on a server. External clients can execute gridified applications and ‘talk’ to GRASG components by means of SOAP [53], a well-known protocol for invoking Web Services.

The IS, RASS and DS services are placed on the *Site* layer, which sits on top of the Grid resources. The IS periodically collects information about the underlying resources from their associated sensors and uses this information to satisfy resource requests originating either at the RASS or at an external client. The RASS bridges application clients to JESs. Specifically, the RASS is in charge of processing job execution requests coming from clients, allocating and reserving the needed Grid resources, monitoring the status of running jobs and returning the results back to the clients. Lastly, the DS is used mainly for moving data among computation servers. It is implemented as a Web Service interface to GridFTP [54], an FTP-based, high-performance, secure, reliable data transfer protocol for Grid environments.

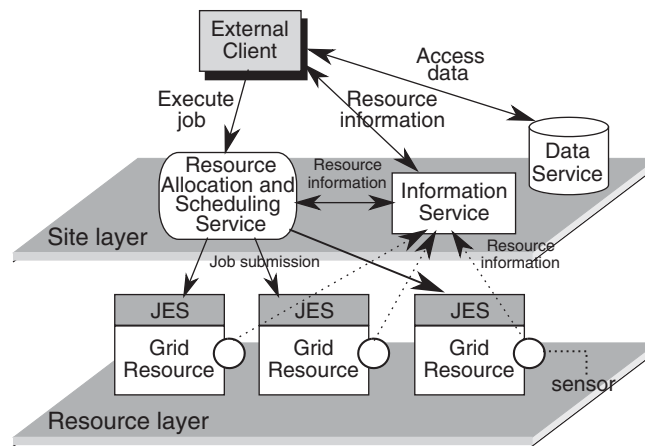


Figure 4. GRASG architecture.

GRASG conceives ‘gridification’ as the process of deploying an existing application (binary executable) on a Grid resource. Once deployed, applications can be easily accessed through their corresponding JES, which stores all the necessary information (e.g. executable paths, system variables, etc.) to execute a gridified or a previously installed application. Similar to GEMLCA, application granularity after gridification is very coarse. To partially deal with potential performance issues caused by this problem, users can define custom scheduling and resource discovery mechanisms for a gridified application by writing new sensors that are based on shell or Perl scripts.

5.4. GridAspecting

GridAspecting [55] is a development process, based on aspect-oriented programming (AOP) [56], to explicitly separate crosscutting Grid concerns in parallel Java applications. Its main goal is to offer guidelines for Grid application implementation focusing on separating the pure functional code as much as possible from the Grid-related code. Besides, GridAspecting relies on a subset of the Java thread model for application decomposition that enables Grid application testing even outside a Grid setting.

GridAspecting uses a finer level of granularity than GEMLCA and GRASG for gridified components. GridAspecting assumes that ordinary applications can be decomposed into a number of independent *tasks*, which can be computed separately. As a first step, the programmer is responsible for identifying these tasks across the, yet non-gridified, application code, and then encapsulating them as Java threads. Any form of data communication from the main application to its task threads should be implemented via parameter passing to the task constructor. As a second step, aspects have to be provided by the programmer in order to map the creation of a task to a job execution request onto a specific Grid middleware (e.g. Globus). At runtime, GridAspecting uses the AspectJ [57] AOP language to dynamically intercept all thread creation and initialization calls emitted by the gridified application, replacing them with calls to the underlying middleware-level execution services by means of those aspects.

Despite being relatively simple, the process requires the developer to follow a number of code conventions. However, applying GridAspecting results in a very modular and testable code. After passing through the gridification process, the functional code of an application is entirely separated from its Grid-related code. As a consequence, a different Grid API can be used without affecting the code corresponding to the application logic.

5.5. GriddLeS

GriddLeS (Grid Enabling Legacy Software) [58] is a development environment that facilitates the construction of complex Grid applications from legacy software. Specifically, it provides a high-level tool for building Grid-aware workflows based on existing, unmodified applications, called *components*. Overall, GriddLeS goals are directed towards leveraging existing scientific and engineering legacy applications and easily wiring them together to construct new Grid applications.

The heart of GriddLeS is GridFiles, a flexible and extensible mechanism that allows workflow components to communicate between each other without the need for source code modification. Basically, GridFiles overloads the common file I/O primitives of conventional languages with functionality for supporting file-based interprocess communication over a Grid infrastructure. In this way, individual components behave as though they were executing in the same machine and using a conventional file system, while they actually interchange data across the Grid. It is important to note that GriddLeS is mainly suited for gridifying and composing legacy applications in which the computation time/communication time ratio is very high. Additionally, components should expose a clear interface in terms of required input and output files, so as to simplify the composition process and not incur component source code modification.

GridFiles makes use of a special language-dependent routine, called *FileMultiplexer*, which intercepts file operations and processes them according to a redirection scheme. Current materializations include local file system redirection, remote file system redirection based on GridFTP and remote process redirection based on sockets. When using process redirection, a multiplexer placed on the sending component is linked with a multiplexer on the receiving component through a buffered channel, which automatically handles data synchronization. In any case, the type of redirection is dynamically selected depending on whether the file identifier represents a local file, a remote file or a socket, and the target's location for the redirection (file or component) is obtained from the GNS (GriddLeS Name Server). The GridFiles mechanism is shown in Figure 5.

The GriddLeS approach is simple, yet very powerful. Application programmers can write and test components without taking into account any Grid-related issues such as data exchanging, synchronization or fault-tolerance, which in turn are handled by the underlying multiplexer being used. Another interesting implication of this fact is that implemented components can transparently operate either as a desktop program or as a block of a bigger application. The weak point of GriddLeS is that its runtime support suffers from portability problems, since it is necessary to have a new implementation for each programming language and OS platform. Also, its implicit socket-based communication mechanism lacks the level of interoperability required by current Grids.

5.6. Ninf-G

Ninf-G [59] is a C/FORTRAN programming environment that aims at providing a simple Grid programming model mostly for non-computer scientists. It builds on top of the Globus toolkit and

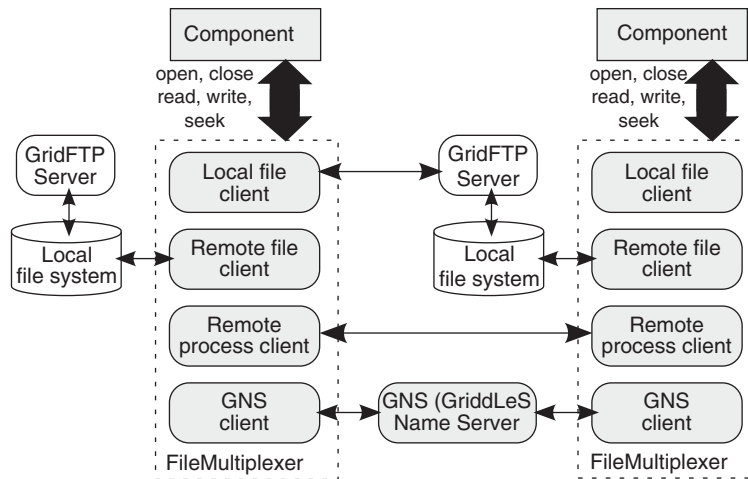


Figure 5. GridFiles: file request redirection.

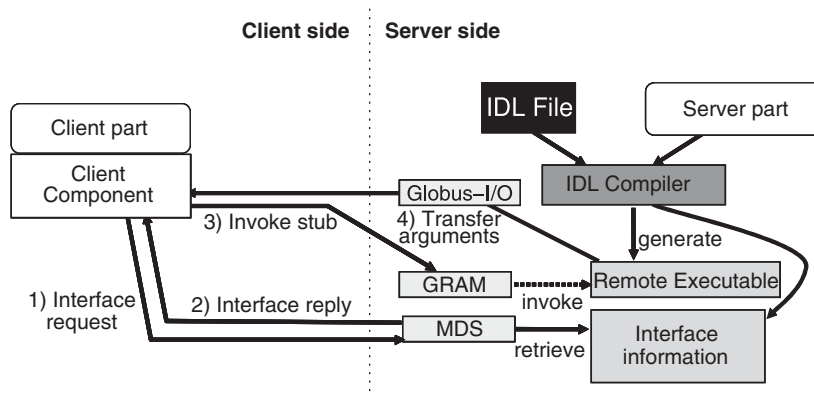


Figure 6. Ninf-G architecture.

offers a reference implementation of the GridRPC specification. Ninf-G provides familiar RPC semantics so that the complicated structure of the Grid is hidden behind an RPC-like interface.

Figure 6 describes the architecture of Ninf-G, which is based on two major components: *Client Component* and *Remote Executable*. The Client Component consists of a client API and libraries for GridRPC invocations. The Remote Executable comprises a stub and system-supported wrapper functions, both similar to those provided by Java RMI or CORBA [60]. The stub is automatically generated by Ninf-G from a special IDL file describing the interface of a remote executable. Both client and server programs are obtained after gridifying an application.

When executing a gridified application, the Client Component and the Remote Executable communicate with each other by using Globus services. First, the Client Component gets the

IDL information for the server-side stub, comprising the remote executable path and parameter encoding/decoding information. This is done by means of the MDS (Monitoring and Discovery System), the Globus network directory service. Then, the client passes the executable path to the Globus GRAM (Grid Resource Allocation Manager), which invokes the server-side part of the application. On execution, the stub requests the invocation arguments from the client, which are transferred using the Globus-IO service.

Roughly, the first step to gridify an application is to identify a client part and one or more server parts. The user should properly restructure its application whenever a server part cannot be straightforwardly obtained from the code, such as merging the most resource-consuming functions into a new one and picking the latter as the server program. In any case, the user must carefully remove any data dependence between the client and the server program, or among server parts (e.g. global variables). Up to this point, the gridification process does not require to be performed within a Grid setting.

The next step is concerned with inserting Ninf-G functions into the client program so as to enable it to interact, via RPC, with its server part(s). Ninf-G has a number of built-in functions for initiating and terminating RPC sessions and, of course, performing asynchronous or synchronous RPC calls. Typical scenarios when gridifying a code with Ninf-G are illustrated in Figure 7.

Deploying a gridified application involves creating the executables on each server. First, the user must specify the interface for the server program(s) using Ninf-G IDL, which is used to automatically generate server-side stubs. Finally, the user must manually register this information in MDS. Although simple, these tasks can be tedious if several applications are to be gridified.

5.7. PAGIS

PAGIS [61] is a Grid programming framework and execution environment suitable for unskilled Grid developers. PAGIS provides a component-based programming model that emphasizes on separating *what* an application does from *how* it does it. Roughly, putting an application to work on the Grid with PAGIS first requires dividing the application into communicating components, and then implementing how these components are executed and controlled within a Grid environment.

A PAGIS application comprises a number of components connected through a network of unidirectional links called a *process network*. In PAGIS terminology, components and links are known as *processes* and *channels*, respectively. A process is a sequential Java program that incrementally



<pre>main(){ pre_processing(); call_library(args); }</pre>	<p>Gridification</p> 	<pre>main(){ for (i=0;i<task_no;i++) task_processing(args); }</pre>
<pre>main(){ pre_processing(); grpc_call(handle, "call_library", args); }</pre>	<p>Gridification</p> 	<pre>main(){ for (i=0;i<task_no;i++) grpc_call_async(dest[i], "task_processing", args); grpc_wait_all(dest); }</pre>

Figure 7. Gridifying applications with Ninf-G: typical scenarios.

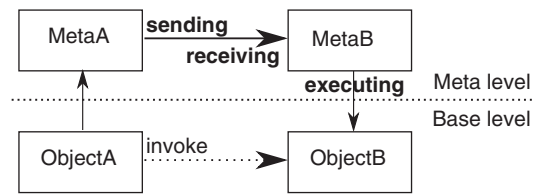


Figure 8. Overview of metalevel programming.

reads data from its incoming channels in a first-in first-out fashion, transforms the data and produces the output to some or all of its outgoing channels. At runtime, PAGIS creates a thread for each process of a network and maintains a producer–consumer buffer for each channel. Production of data is non-blocking, whereas consumption from an empty stream is blocking. As the reader may observe, this mechanism shares many similarities with the Unix process pipelining model.

Similar to most component-based frameworks, PAGIS processes are described in terms of *ports*. Ports define a communication contract with a process in the same way as classes define interfaces for objects in object-oriented languages. In this way, applications are described by connecting ports through channels. PAGIS includes an API, called PNAPI (Process Network API), that provides several useful abstractions for describing applications in terms of process networks. Additionally, it offers a graphical tool for visually creating, composing and executing process networks.

PAGIS allows a process network to be supplied with Grid behavior by means of *metalevel programming*. Conceptually, metalevel programming divides an application into a *base level*, composed of classes and objects implementing its functional behavior, and a *meta level*, consisting of *metaobjects* that reify elements of the application at runtime—mostly method invocations—and perform computations on them. Figure 8 illustrates the basics of metalevel programming. Both base level objects, *ObjectA* and *ObjectB*, have been assigned two different metaobjects. As a consequence, *MetaA* receives all method invocations sent from *ObjectA* and redirects them to the target's metaobject (in this case *MetaB*), which actually carries out the invocations. The labels in bold represent the phases of a method invocation with which customized user actions can be associated.

PAGIS introduces the *MetaComputation* metaobject, specially designed to represent a running process network as a single structure. Users can then materialize complex Grid functionality by attaching metaobjects^{||} to *MetaComputation* metaobjects. For example, one might implement a custom metaobject for transferring certain method invocations to a remote metaobject, thus achieving load balancing. Similarly, a metaobject that monitors and records the various runtime aspects of an application can be easily implemented by logging information such as timing, source and destination objects, among others, prior to method redirection.

The gridification scheme proposed by PAGIS is indeed interesting, since it allows to furnish ordinary applications (i.e. the base level) with Grid-dependent behavior (i.e. the meta level) without affecting its source code. The only requirement is that those applications should be appropriately transformed so that they are structured as a process network. Similar to GridAspecting, a PAGIS application (i.e. a process) is specified at a task level of granularity.

^{||}Strictly speaking, these are meta-metaobjects, since they intercept method calls performed by other metaobjects.

5.8. Proactive

Proactive [62] is a Java-based middleware for object-oriented parallel, mobile and distributed computing. It includes an API that isolates many complex details of the underlying communication and reflection Java APIs, on top of which a component-oriented view is provided. This API also includes functionality to transform conventional Java classes to a Proactive application. The programming model featured by Proactive has also been implemented in C++ and Eiffel.

A typical Proactive application is composed of a number of mobile entities called *active objects*. Each active object has its own thread of control and an entry point, called the *root*, by which the object functionality can be accessed from ordinary objects. Active objects serve method calls issued from other active/ordinary objects and also request for services implemented by other local or remote active objects. Method calls sent to active objects are synchronized using the *wait-by-necessity* mechanism, which transparently blocks the requester until the results of a call are received. At the ground level, this mechanism relies on meta-programming techniques similar to that of PAGIS; thus, it is very transparent to the programmer.

JVMs participating in a computation can host one or more *nodes*. A node is a logical entity that groups and abstracts the physical location of a set of active objects. Nodes are identified through a symbolic name, typically a URL. Therefore, active objects can be programmatically attached/detached from nodes without the need for manipulating low-level information such as network addresses or ports. Similarly, active objects can be sent for execution to remote JVMs by simply assigning them to a different ‘container’ node.

Standard Java classes can be easily transformed into active objects. For example, let us assume that we have a class named *C*, which exposes two methods *foo* and *bar*, with return type *void* and *double*, respectively. The API call:

```
C c = (C) ProActive.newActive("C", args, "rmi://isistan.exa.unicen.edu.ar/myNode");
```

creates—by means of RMI—a new active object of type *C* on the node *myNode*. Further calls to either *foo* or *bar* are asynchronously handled by Proactive, and any attempt to read the result of an invocation to *bar* blocks the caller until the result is computed. In a similar way, the API can be used to straightforwardly publish an active object as a SOAP-enabled Web Service.

Another interesting feature provided by Proactive is the notion of *virtual nodes*. The idea behind this concept is to abstract away the mapping of active objects to physical nodes by eliminating from the application code elements such as host names and communication protocols. Each virtual node declared by the application is identified through a plain string and mapped to one or a set of physical nodes by means of an external XML deployment descriptor file. As a consequence, the resulting application code is independent of the underlying execution platform and can be deployed on different Grid settings by just modifying its associated deployment descriptor file.

There are, however, some code conventions that programmers must follow before gridifying an ordinary Java class as an active object. First, classes must be serializable and include a default constructor (i.e. with no arguments). Second, the result of a call to a non-void method should be placed on a local variable for the wait-by-necessity mechanism to work. Return types for non-void methods should be replaced by system-provided wrappers accordingly. In our example, we have to replace the return type in the *bar* method with a Proactive API class that wraps the *double* Java primitive type.

5.9. Satin

Satin [63] is a Java framework that allows programmers to easily parallelize applications based on the divide-and-conquer paradigm. The ultimate goal of Satin is to free programmers from the burden of modifying and hand-tuning applications to exploit a Grid setting. Satin is implemented on top of Ibis [64], a programming environment with the goal, of providing an efficient Java-based platform for Grid programming. Ibis consists of a highly efficient communication library, and a variety of programming models, mostly for developing applications as a number of components exchanging messages through messaging protocols such as Java RMI and MPI.

Satin extends Java with two primitives to parallelize single-threaded conventional Java programs: *spawn*, to create subcomputations (i.e. divide), and *sync*, to block execution until the results from subcomputations are available. Methods considered for parallel execution are identified by means of *marker interfaces* that extend the *satın.Spawnable* interface. Furthermore, a class containing spawnable methods must extend the *satın.SatinObject* class and implement the corresponding marker interface. In addition, the result of the invocation of a spawnable method must be stored on a local variable. The next code shows the Satin version of a simple recursive solution to compute the *k*th Fibonacci number:

```
interface FibMarkerInterface extends satın.Spawnable{
    public long fibonacci(long k);
}

class Fibonacci extends satın.SatinObject implements FibMarkerInterface{

    public long fibonacci(long k){
        if (k < 2)
            return k;
        // The next two calls are automatically spawned,
        // because "fibonacci" is marked in FibMarkerInterface
        long f1 = fibonacci(k - 1);
        long f2 = fibonacci(k - 2);
        // Execution blocks until f1 and f2 are instantiated
        super.sync();
        return f1 + f2;
    }

    static void main(String[] args){
        ...
        Fibonacci fib = new Fibonacci();
        // also spawned
        long result = fib.fibonacci(k);
        // Blocks the main application thread
        // until a result is obtained
        fib.sync();
        ...
    }
}
```

After indicating spawnable methods and inserting appropriate synchronization calls into the application source code, the programmer must feed a compiled version of the application to a tool that translates, through Java bytecode instrumentation, each invocation to a spawnable method into a Satin runtime task. For example, in the code shown above, a task is generated for every single call to the *fibonacci* method.

Since each task represents the invocation (recursive or not) to a spawnable method, their granularity is clearly smaller than the granularity of tasks similar to the ones supported by GridAspecting

or PAGIS. A running application may therefore be associated with a large number of fine-grained tasks, which can be executed on any machine. For overhead reasons, most tasks are processed on the machine in which they were created. In order to efficiently run gridified programs, Satin uses a task execution scheme based on a novel load-balancing algorithm called CRS (Cluster-aware Random Stealing). With this algorithm, when a machine becomes idle, it attempts to steal a task waiting to be processed from a remote machine. Intra-cluster steals have a greater priority than wide-area steals. This policy fundamentally aims at saving bandwidth and minimizing the latencies inherent in slow wide-area networks.

5.10. XCAT

XCAT [65] is a component-based framework for Grid application programming built on top of Web Service technologies. XCAT applications are created by connecting distributed components (OGSA Web Services) that communicate either by SOAP messaging or by an implicit notification mechanism. XCAT is compliant with Common Component Architecture [66], a specification with the goal of yielding a reduced set of standard interfaces that a high-performance component framework should provide to or expect from components in order to achieve easy distributed component composition and interoperability.

XCAT components are connected by *ports*. A port is an abstraction representing the interface of a component. Ports are described in SIDL (Scientific Interface Definition Language), a language for describing component operations in terms of the data types often found in scientific programs. There are two kinds of ports: *provides* ports, representing the services offered by a component, and *uses* ports, describing the functionality a component might need but is implemented by another component. Furthermore, XCAT provides ports can be implemented as OGSA Web Services. XCAT uses ports can connect not only to any typed XCAT provides port (i.e. those described in SIDL) but also to any OGSA-compliant Web Service.

XCAT allows scientific legacy applications to be deployed as components without code modification using the concept of *application manager* (see Figure 9). Basically, each legacy application is

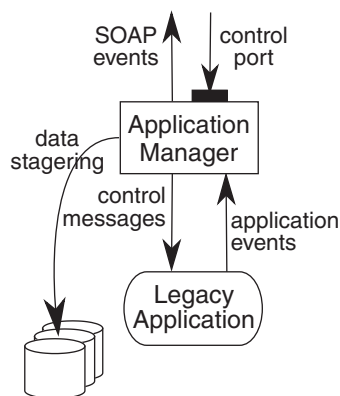


Figure 9. XCAT application managers.

wrapped with a generic component (application manager) responsible for managing and monitoring the execution of the application, and staging the necessary input and output data. The manager also serves as a forwarder for events taking place inside the wrapped application, such as file creation, errors and execution finalization/crash. Application managers can be connected to each other and have one special port by which standard components can control them. It is worth noting that legacy applications have, in general, a large granularity. As a consequence, XCAT shares some of the limitations of GEMICA and GRASG with respect to the granularity of gridified applications.

6. A TAXONOMY OF GRIDIFICATION APPROACHES

Table I summarizes the main characteristics of the approaches described in the previous sections. To better understand the structure of the table, the reader should recall the analogy between the Grid and the electrical power grid discussed in Section 3.

Basically, each row of the table represents a ‘wall socket’ by which applications are gridified and connected to the Grid. The ‘Appliance type’ column symbolizes the kind of applications supported by the gridification process, whereas the ‘Grid-aware appliance’ column briefly describes the new anatomy of applications after passing through the gridification process. In addition, lower-level Grid technologies upon which each approach is built are also listed. Finally, we center our discussion on the different approaches to gridification (i.e. the ‘plugging techniques’) according to the taxonomies presented in the following subsections.

In particular, the taxonomies of Sections 6.1 and 6.2 describe, from the point of view of source code modification, the different ways in which an ordinary application can be affected by the gridification process. The taxonomy presented in Section 6.3 represents the observed granularity levels to which applications are Grid enabled. Finally, the taxonomy included in Section 6.4 briefly categorizes the approaches according to the kind of Grid resources they aim to virtualize. These taxonomies are simple, but comprehensive enough to cover the various aspects of gridification.

6.1. Application re-engineering

The application re-engineering taxonomy defines the extent to which an application must be manually modified in order to obtain its gridified counterpart. In general, the static anatomy of every application can be described as a number of *compilation units* combined with a certain *structure*. Compilation units are programming language-dependent pieces of software (e.g. Java classes, C and Perl modules, etc.) assembled together to form an application. Usually, a compilation unit corresponds to a single source code file. Furthermore, the way compilation units are combined determines the structure of the application (e.g. the class hierarchy of a Java application, the dependence graph between functions within a C program). According to Figure 10, the anatomy of a conventional application might be altered in the following ways:

- *Structure only*: Some approaches alter the internal structure of the application, restructuring it in such a way that some of its constituent parts are reorganized. For example, GridAspecting requires the user to identify tasks within the application that can potentially be executed concurrently. Similarly, the PAGIS framework requires to restructure applications as a set of

Table I. Summary of gridification tools.

Wall socket	Appliance type	Plugging technique highlights	Grid-aware appliance	Underlying technologies
GEMLCA	Binary executable	The user must specify the interface of his/her application (XML file)	Globus-wrapped binary executable	Web Services; Globus
GrADS	C application (MPI-based if explicit migration is to be used)	Instruction insertion if using SRS	Globus-wrapped binary executable	Web Services; Globus; NWS; MPI
GRASG	Binary executable	Hand-tuning of applications through Perl/shell scripting	Binary executable interfaced through a <i>JES</i> Web Service	SOAP-based Web Services; Globus
Grid-aspecting	Task-parallel Java application	Manual task decomposition and Grid concerns (aspects) implementation	Multi-threaded, aspect-enhanced Java application	AspectJ
GriddLeS	Stream-based binary executable	Transparent over loading of system libraries implementing file/sockets operations across the Grid	Globus-wrapped binary executable (<i>component</i>)	Globus; GridFTP
Ninf-G	C/Fortran application	Users decompose applications into client/server parts, and connect them using GridRPC calls	Client- and server-side binary executables	Globus; GridRPC
PAGIS	Java application	Users identify components and assemble them through <i>channels</i> to build <i>process networks</i>	Binary executable (<i>process</i>)	—
Proactive	Java application	Source code conventions; proxy-based wrapping	Active object	SOAP-based Web Services; RMI
Satin	Divide and conquer Java application	Source code conventions; bytecode instrumentation enabling recursive method calls to be <i>spawnable</i>	Single-threaded, parallel Java application	Ibis
XCAT	Binary executable	The user must specify the interface of his/her executable in SIDL	Binary executable interfaced through an <i>application manager</i>	Web Services

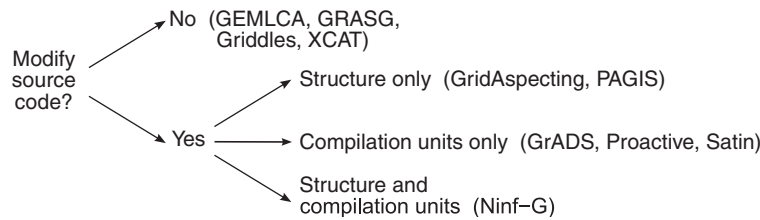


Figure 10. Application re-engineering taxonomy.

components exposing and invoking services through well-defined interfaces. However, in both cases, the user code originating these tasks and components practically remains unchanged. In general, this procedure makes the appearance of the original application significantly different from that of the Grid-aware application, but the pure implementation code is practically the same. In other words, even though the code within compilation units may slightly change, the focus of structure modification is on redesigning the application rather than on rewriting it (i.e. internally modify methods/procedures).

Structure modification is a very common approach to gridification among template-based Grid programming frameworks. With these frameworks, the user adapts the structure of his/her application to a specific template implementing a recurring execution pattern defined by the framework. For example, JaSkel [44] is a Java framework for developing parallel applications that provides a set of abstract classes and a skeleton catalog, which implements interaction paradigms such as farm, pipeline, divide-and-conquer and heartbeat templates. Another example is MW [42], a framework based on the popular master–worker paradigm for parallel programming.

- *Compilation units only*: Conversely, other approaches alter only some compilation units of the application. For instance, Proactive and Satin require the developer to modify certain methods within the application to make them compliant to a specific coding convention. But, in both cases, the class hierarchy of the application is barely modified. A taxonomy of gridification techniques for single compilation units is presented in the next subsection.

Examples of compilation unit modification are commonly found in the context of distributed programming. For instance, this technique is frequently employed when a single-machine Java application is adapted to using a distributed object technology such as RMI or CORBA. Some of the (formerly local) objects are explicitly distributed on different machines and looked up by adding specific API calls inside the application code. However, the behavioral relationships between those distributed objects do not change. Similar examples can also be found in distributed procedural programming using technologies such as MPI or RPC.

- *Structure and compilation units*: Of course, gridification methods may also modify both the structure and compilation units of the application. For example, Ninf-G demands the developer to split an application into client- and server-side parts and then to modify the client so as to remotely interact with the server(s). Note that the internal structure of the application changes dramatically, since a single server part may contain a code combining the functions originally placed at different compilation units. Overall, not only is the ordinary application refactored by creating many separate programs, but also several modifications to some of the original functions are introduced.

Intuitively, the first technique enables the user to perform modifications at a higher level of abstraction than the second one. Users are not required to provide code for using Grid functionality and deal with Grid details but have to change the application shape. As a consequence, the application logic is not significantly affected after gridification. In principle, the most undesirable technique is by far to modify the structure and compilation units of an application, since not only the application shape but also the nature of its code are changed. However, it is very difficult to determine whether a technique is better than the others, as the amount of effort necessary to gridify an application with either of the three approaches depends on its complexity/size, the amount/type of modifications imposed by the gridification method for restructuring and/or rewriting the application, the programming language, and the particular Grid setting and underlying technologies being used for application execution.

6.2. Compilation unit modification

The compilation unit modification taxonomy determines how applications are altered following gridification with respect to modification of their compilation units. As shown in Figure 11, we can broadly identify the following categories:

- *Instruction insertion*: The most intuitive way to gridify is, as its name indicates, by manually inserting instructions implementing specific Grid functionality at proper places within the code. A case of instruction insertion arises in GrADS when the user wishes to explicitly control application migration and data staging. A clear advantage of this technique is that the programmer can optimize his/her application at different levels of granularity to produce a very efficient Grid application. However, in most cases, the application logic is literally mixed up with Grid-related code, thus making maintainability, legibility, testing and portability to different Grid platforms very hard.

There are many Grid middlewares that require users to employ instruction insertion when gridifying compilation units. For example, JavaSymphony [67] is a programming model, the purpose of which is to simplify the development of performance-oriented, object-based Grid applications. It provides a semi-automatic execution model that deals with migration, parallelism and load balancing of applications, and at the same time allows the programmer to control—via instruction insertion—such features as needed. Other examples are Javelin 3.0 [68] and GridWay [69], two platforms for deployment and execution of CPU-intensive applications which require users to modify applications in order to exploit job checkpointing and parallelization.

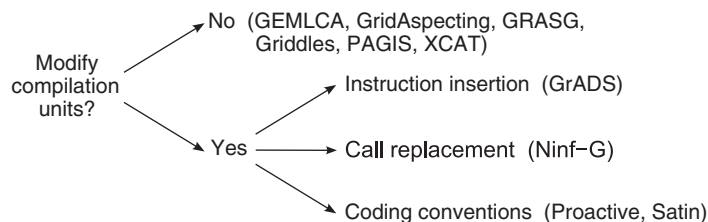


Figure 11. Compilation unit modification taxonomy.

- *Call replacement*: A very common technique to gridify compilation units is by replacing certain groups of sequential instructions by appropriate calls to the underlying middleware API. Such instructions may range from operations for carrying out interprocess communication to coding for manipulating data. Unlike the previous case, call replacement places more emphasis on replacing certain pieces of conventional code by the Grid-aware code instead of inserting new instructions throughout the user application.

Call replacement assumes that users know what portions of their code should be replaced in order to adapt it for running on a particular Grid middleware. Nevertheless, in order to help users in doing this task, Grid middlewares usually offer guidelines tutoring users on how to port their applications. For instance, gridifying with the Globus platform involves replacing socket-based communication code by calls to the Globus I/O library, transforming conventional data copy/transfers operations by GridFTP operations, and finally replacing all resource discovery instructions (e.g. for obtaining available execution nodes) by calls to the MDS service.

Clearly, call replacement is a form of gridification suitable for users who are familiar with the target middleware API. Users not having a good understanding of the particular API to be used may encounter difficulty in porting their applications to the Grid. Another drawback of the approach is that the resulting code is highly coupled with a specific Grid API, thus having many of the problems suffered by instruction insertion. These issues are partially solved by toolkits that attempt to offer a comprehensive, higher-level programming API on top of middleware-level APIs (e.g. Java CoG Kit, GAT). However, developers are forced to learn yet another programming API. Indeed, Grid toolkits help alleviate developer pain caused by call replacement, but certainly they are not the cure.

- *Coding conventions*: This technique is based on the idea that all the compilation units of an ordinary application must obey certain conventions about their structure and coding style prior to gridification. These conventions allow tools to properly transform a gridified application into one or more middleware-level execution units. For example, Proactive requires application classes to extend the Java *Serializable* interface. Moreover, Satin requires that the result of any invocation to a recursive method is placed on a variable, rather than accessing it directly (e.g. pass it on as an argument to another method). Unlike instruction insertion and call replacement, the gridified code is in general not tied to any specific Grid API or library.

To provide an illustrative example in the context of conventional software, we could cite JavaBeans [70], a widely known specification from Sun that defines conventions for writing reusable software components in Java. In order to operate as a JavaBean, a class must follow conventions about method naming and behavior. This, in turn, enables easy graphical reuse and composition of JavaBeans to create complex applications with little implementation effort.

It is worth pointing out that using any of the above techniques does not automatically exclude from using the others. In fact, they usually complement each other. For example, Satin, despite being focused on gridifying by imposing coding conventions, requires programmers to coordinate several calls to a spawnable computation within a method by explicitly inserting special synchronizing instructions. Furthermore, it is unlikely that an application that has been adapted to use a specific Grid API (e.g. GridFTP in the case of Ninf-G) will not include user-provided instructions for performing some API initialization or disposal tasks.

6.3. Gridification granularity

Granularity is a software metric that attempts to quantify the size of the individual components** that make up a software system. Large components (i.e. those including much functionality) are commonly called *coarse-grained*, whereas those components providing little functionality are usually called *fine-grained*. For example, with Service-oriented Architectures (SOA) [71], applications are built in terms of components called *services*. In this context, component granularity is determined by the amount of functionality exposed by services, which may range from small (e.g. querying a database) to big (e.g. a facade service to a travel business).

Notwithstanding granularity being usually associated with the size of application components from a user's point of view, the concept can also be applied to get an idea of how granular the runtime components of a gridified application are. We define *gridification granularity* as the granularity of the individual components that constitute an executing gridified application from the point of view of the Grid middleware. Basically, these Grid-enabled components are execution units like jobs or tasks to which the Grid directly provides scheduling and execution services. Note that 'conventional' granularity does not necessarily determine gridification granularity. Clearly, this is because the former is concerned with the size of the components *before* an application is transformed to run on a Grid setting. For example, during gridification, a single coarse-grained service might be partitioned into several more granular services to achieve scalability.

Similar to conventional granularity, gridification granularity takes continuous values ranging from the smallest to the largest possible component size. As shown in the taxonomy of Figure 12, we divided the spectrum of gridification granularities into three discrete values:

- *Coarse-grained*: A running application is composed of a number of 'heavy' execution units. Typically, the application execution is handled by only one runtime component. This level of granularity usually results from employing solutions such as GEMICA, GRASG, GriddLes and XCAT, which adapt the executable of an ordinary application to be executed as a single Grid-aware job. At runtime, a job behaves like a 'black box' that receives a pre-determined set of input parameters (e.g. numerical values, files, etc.), performs some computation and returns the results back to the executor. A similar case occurs with compiled versions of applications gridified with GrADS.

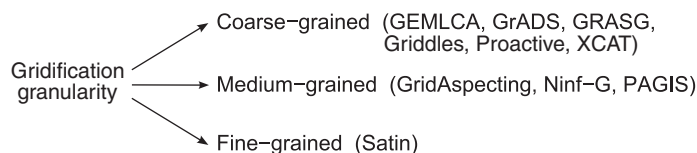


Figure 12. Gridification granularity taxonomy.

**The term 'component' refers to any single piece of software included in a larger system and should not be confused with the basic building blocks of the component-based programming model.

Coarse-grained gridification granularity suffers from two major problems. On the one hand, the application is treated by the middleware as a single execution unit. Therefore, unless refactored, it may not be possible for the individual resource-consuming parts of a running application to take advantage of mechanisms such as distribution, parallelization or scheduling to achieve higher efficiency. On the other hand, the middleware sees a running application as an indivisible unit of work. As a consequence, some of its portions that might be dynamically reused by other Grid applications (e.g. a data mining algorithm) cannot be discovered or invoked.

To a lesser extent, Proactive applications can also be considered as coarse grained. An ordinary Java application (i.e. its main class and helper classes) is gridified by transforming it into a self-contained active object. The user sees the non-gridified application as composed of a number of (medium-grained) objects. On the other hand, Proactive sees the gridified application as one big (active) object. When executing the application, the Proactive runtime performs scheduling and distribution activities on active objects rather than on plain objects. Nevertheless, Proactive is more flexible than the other approaches in this category, since it allows developers to explicitly manage mobility inside an active object, invoke methods from other active objects and externalize the methods implemented by an active object.

- *Medium-grained*: The running application has a number of execution units of moderate granularity. Systems following this approach are GridAspecting, Ninf-G and PAGIS. In the former and latter cases, the user identifies those tasks within the application code that can be executed concurrently. Then, they are mapped by the middleware to semi-granular runtime task objects. Similarly, a running Ninf-G application is composed of several IDL-interfaced processes that are distributed across a network. Unlike GridAspecting and PAGIS, this approach affords an opportunity for dynamic component invocation, as a Ninf-G application might perform calls to the functions exposed by the components of another Ninf-G application.
- *Fine-grained*: This category represents the gridification granularity associated with runtime components generated on the invocation of a method/procedure. A representative case of fine-grained granularity is *Satin*. Basically, a middleware-level task is created after every single call to a spawnable method, regardless of whether calls are recursive or not. From the application point of view, there is a better control of parallelism and asynchronism. However, a running application may generate a large number of tasks that should be handled efficiently by the underlying middleware. This fact suggests the need for a runtime support providing sophisticated execution services smart enough to efficiently deal with task scheduling and synchronization issues.

It is worth noting that, in some cases, the user may indirectly adjust (e.g. by refactoring code) the gridification granularity to fit specific application needs. For example, a set of medium-grained tasks could be grouped into one bigger task in order to reduce communication and synchronization overhead. Conversely, the functionality performed by a task could be decomposed into one or more tasks to achieve better parallelism. Nonetheless, this process can be cumbersome and sometimes counterproductive. For example, Proactive applications can be restructured by turning standard objects into active objects, but then the programmer must explicitly provide the code for handling active object lookup and coordination. Similarly, gridification granularity of Ninf-G applications can be reduced by increasing the number of server-side programs. However, this could cause the application to spend more time communicating than doing useful computations.

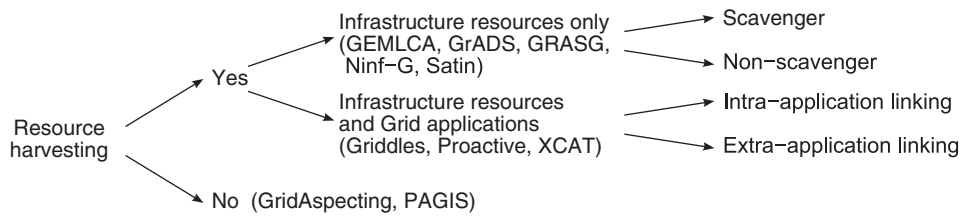


Figure 13. Resource-harvesting taxonomy.

6.4. Resource harvesting

The resource harvesting taxonomy describes, in a general way, the kind of Grid resources to which access is made transparent by each gridification method. The utmost goal of Grid Computing, as explained at the beginning of this article, is to virtualize distributed resources so that they can be transparently used and consumed by ordinary applications. Certainly, gridification tools play a fundamental role in achieving such a transparency. The resource-harvesting taxonomy is depicted in Figure 13.

Surprisingly, some gridification methods do not pursue resource virtualization. Specifically, solutions such as GridAspecting or PAGIS aim at preserving the integrity of the application logic during gridification and make them independent of a specific Grid platform or middleware. In this way, users have the flexibility to choose the runtime support or middleware that better suits their needs. However, as these approaches do not offer facilities for using Grid services, the burden of providing the ‘glue’ code for interacting with the Grid is entirely placed on the application developer, which clearly demands a lot of programming effort.

Most gridification methods, however, provide some form of Grid resource leveraging, along with a minimal or even no effort from the application developer. Basically, these are integrated solutions that offer services for gridifying ordinary applications as well as accessing Grid resources. Depending on the type of resource they attempt to virtualize, these methods can be further classified as follows:

- *Infrastructure resources only*: Applications resulting from applying the gridification process are not concerned with providing services to other Grid applications. Applications are simply ported to the Grid to transparently leverage middleware-level services (e.g. resource brokering, load balancing, mobility, scheduling, parallelization, storage management, etc.) that virtualize and enhance the capabilities of computational resources such as processing power, storage, bandwidth, etc. Moreover, some approaches are more focused on harnessing idle CPU power (the so-called ‘scavengers’; GrADS, Ninf-G, Satin), whereas others also include simple abstractions and easy-to-use services to deal with data management on the Grid (GEMLCA, GRASG). In any case, the emphasis is put solely on taking advantage of Grid resources, rather than on using Grid services *and* services implemented by other Grid applications.
- *Infrastructure resources and Grid applications*: The goal of these approaches is to simplify the consumption of both Grid services and functionality offered by gridified applications. At the middleware level, gridified applications are treated just like any other individual Grid

Table II. Comparison between gridification tools leveraging both Grid resources and applications.

Tool	Interface description	Communication protocol	Application discovery
GriddleS	Implicit (file-based)	Sockets	No
Proactive	Explicit (WSDL)	SOAP	Yes (lookup by active object identifier)
XCAT	Explicit (WSDL)	SOAP or XML-based implicit notification	No

resource: an entity providing special capabilities that can be used/consumed by other applications by means of specialized Grid services. Note that this is a desirable property for a gridification tool, since reusing existing Grid applications may improve application modularity and drastically reduce development effort [72].

Linking together Grid applications requires the underlying middleware to provide, in principle, mechanisms for communicating applications. These mechanisms may range from low-level communication services such as those implemented by GriddleS to high-level, interoperable messaging services like SOAP. In addition, mechanisms are commonly provided to describe the interface of a gridified application in terms of the internal services that are made accessible to the outside, and also to discover existing Grid applications. For example, popular technologies for describing and discovering Grid applications are WSDL [73] and UDDI [74], respectively. Table II briefly compares the tools that support application linking by showing how they deal with application interface description, communication and discovery.

There are basically two forms to connect applications: *extra-application* and *intra-application*. In the extra-application approach, existing Grid applications can be reused by combining and composing them into a new application. For example, XCAT conceives gridified applications as being indivisible components that can be combined—with little coding effort—into a bigger application, but no binding actions are ever carried out from inside any of these components. Another example of a gridification tool following this approach is GMarte [75], a high-level Java API that offers an object-oriented view on top of Globus services. With GMarte, users can compose and coordinate the execution of existing binary codes by means of a (usually small) new Java application. On the other hand, in the intra-application linking approach, users are not required to implement a new ‘container’ application, since binding to existing Grid applications is performed within the scope of a client application. GriddleS and Proactive are examples of approaches based on intra-application linking.

Gridification approaches oriented towards consuming Grid resources are engaged in finding ways to make the task of porting applications to use Grid services easier. On the other hand, approaches seeking to effortlessly take advantage of Grid resources and existing applications generalize this idea by providing a unified view over Grid resources in which applications not only consume but also offer Grid services. It is important to note that this approach shares many similarities with the service-oriented model, where applications may act both as clients and as providers of services. In fact, many global Grid standards, such as OGSA and WSRF, have already embodied the convergence of SOA and Grid Computing technologies.

7. DISCUSSION

Table III summarizes the approaches discussed so far. Each cell of the table corresponds to the taxonomic value associated with a particular tool (row) with respect to each of the taxonomies presented in the previous section.

There are approaches that let users to gridify applications without modifying a single line of code. Solutions belonging to this category take the application in their binary form, along with some user-provided configuration (e.g. input and output parameters and resource requirements), and wrap the executable code with a software entity that isolates the complex details of the underlying Grid. It is important to note that this approach has both advantages and disadvantages. On the one hand, the user does not need to have a good expertise on Grid technologies to gridify his/her applications. Besides, applications can be plugged into the Grid even when the source code is not available. On the other hand, the approach results in extremely coarse-grained gridified applications; thus, users generally cannot control the execution of their applications in a fine-grained manner. This represents a clear tradeoff between ease of gridification and flexibility to control the various runtime aspects of a gridified application.

A remarkable result of the survey is the diversity of programming models existing among the analyzed tools: procedural and message passing (GrADS), AOP (GridAspecting, PAGIS), workflow-oriented (GriddLes), RPC (Ninf-G), component-based (PAGIS, Proactive, XCAT), object-oriented (Proactive, Satin), just to name a few. This evidences the absence of a widely adopted programming model for the Grid, in contrast to other distributed environments (e.g. the Web) where well-established models for implementing applications are found [76].

Another interesting result is the way in which technologies such as Java, Web Services and Globus have influenced the development of gridification tools within the Grid. Specifically, many of the surveyed tools are based on Java or rely on Web Services, and almost all of them either build on top of Globus or provide some integration with it. Nevertheless, this result should not

Table III. Summary of gridification approaches.

Tool	Application re-engineering	Compilation unit modification	Gridification granularity	Resource harvesting
GEMLCA	No	No	Coarse grained	Grid resources
GrADS	Yes (compilation units only)	Instruction insertion	Coarse grained	Grid resources
GRASG	No	No	Coarse grained	Grid resources
GridAspecting	Yes (structure only)	No	Medium grained	No
GriddLes	No	No	Coarse grained	Grid resources and applications
Ninf-G	Yes (structure and compilation units)	Call replacement	Medium grained	Grid resources
PAGIS	Yes (structure only)	No	Medium grained	No
Proactive	Yes (compilation units only)	Code conventions	Coarse grained	Grid resources and applications
Satin	Yes (compilation units only)	Code conventions	Fine grained	Grid resources
XCAT	No	No	Coarse grained	Grid resources and applications

be surprising for several reasons. Java has been widely recognized as an excellent choice for implementing distributed applications mainly because of its ‘write once, run anywhere’ philosophy, which promotes platform independence. Web Services technologies enable high interoperability across the Grid by providing a layer that abstracts clients and Grid services from network-related details such as protocols and addresses. Lastly, Globus—baptized by Ian Foster as the ‘Linux of the Grid’—has become the *de facto* standard toolkit for implementing Grid middlewares, since it provides a continuous evolving and robust API for common low-level Grid functionalities, such as resource discovery and monitoring, job execution and data management.

8. CONCLUSIONS

Grid Computing enables users to effortlessly take advantage of the vast amounts and types of computational resources available on the Grid by simply plugging applications into it. However, given the extremely heterogeneous, complex nature inherent in the Grid, adapting applications to run on a Grid setting has been widely recognized as a very difficult task. So comes the challenge to provide appropriate methods to *gridify* applications, that is, semi-automatic and automatic methods for transforming conventional applications to benefit from Grid resources. In this sense, a number of gridification approaches have been proposed in an attempt to reality to catch up with this ambitious dream.

Unfortunately, current approaches to gridification cope only with a subset of the problems that are essential to truly achieving gridification, while not addressing the others. Ideally, ordinary applications should be made Grid-aware without the need for manual code refactoring, modification or adaptation. Besides reducing development effort, this would enable even the most novice Grid users to quickly and easily put their applications to run on the Grid. Similarly, users should also be able to take advantage of Grid resources and existing Grid applications with little, or eventually non-, coding effort. Last but not least, the gridification process should also take into account the runtime characteristics of the applications being gridified to provide mechanisms by which users can easily adjust the granularity of application components so as to produce Grid-aware applications that can be efficiently executed. Consequently, there is a need for new approaches capable of effectively coping with all these issues.

Recently, SOAs have appeared as an elegant approach to tackle down some of the problems suffered by current gridification methods. SOAs provide the basis for *loose coupling*: interacting applications that know little about each other in the sense that they discover the necessary information to use external services (protocols, interfaces, location, etc.) in a dynamic fashion. This frees developers from explicitly providing a code for connecting applications together and accessing resources from within an application. Moreover, SOAs enable application modularity, interoperability, reusability and various application granularities. As a matter of fact, it is not clear where to draw the line between Grid Services and Web Services technologies [18]. Furthermore, current Grid standards are actively promoting the use of SOAs and Web Services for materializing the next generation architectures and middlewares for the Grid [72].

Finally, although the analysis in this paper has been explicitly centered around the notion of gridification as the process of transforming the source code of an application to run on the Grid, an aspect that deserves special attention is the amount of configuration that may be necessary to truly make this transformation happen. In a broader sense, gridifying an application is concerned not only

with making conventional source code Grid-aware but also with supplying some Grid-dependent configuration in order to run the adapted application, which usually ranges from application-specific parameters (e.g. expected execution time and memory usage) to deployment information (e.g. number of nodes to use). Sadly, this demands developers' knowing in advance of many platform-related details before an application can take advantage of Grid services.

As gridification methods evolve, difficulties in gridifying ordinary applications seem to move from adapting source code to configuring and deploying Grid-aware applications. For example, this fact is evident in those approaches (e.g. GEMICA, GRASG, XCAT) where code modification is not required but deployment becomes difficult. Nevertheless, the problem of simplifying the deployment of Grid applications has been acknowledged by some of the current gridification tools. For instance, a Proactive application can be executed on several Internet-connected machines by configuring and launching the application at a single location. Another incipient work towards this end can be found in [77], a middleware with the goal of easing both programming and deployment of conventional Java applications.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and suggestions to improve the quality of the paper.

REFERENCES

1. Foster I, Kesselman C (eds.). *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann: San Francisco, CA, U.S.A., 2003.
2. Foster I. The grid: Computing without bounds. *Scientific American* 2003; **288**(4):78–85.
3. Foster I, Kesselman C, Tuecke S. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications* 2001; **15**(3):200–222.
4. Sarmienta LFG, Hirano S, Bayanihan: Building and studying volunteer computing systems using Java. *Future Generation Computer Systems, Special Issue on Metacomputing* 1999; **15**(5–6):675–686.
5. Distributed.net. The distributed.net project. <http://www.distributed.net> [20 July 2007].
6. Folding@home. The folding@home project. <http://folding.stanford.edu/> [20 July 2007].
7. Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D. SETI@home: An experiment in public-resource computing. *Communications of the ACM* 2002; **45**(11):56–61.
8. Loewe L. Evolution@home: Observations on participant choice, work unit variation and low-effort global computing. *Software—Practice and Experience* 2007; DOI: 10.1002/spe.806.
9. Natrajan A, Humphrey MA, Grimshaw AS. The Legion support for advanced parameter-space studies on a Grid. *Future Generation Computer Systems* 2002; **18**(8):1033–1052.
10. Thain D, Tannenbaum T, Livny M. Condor and the Grid. *Grid Computing: Making the Global Infrastructure a Reality*, Berman F, Fox G, Hey A (eds.). Wiley: New York, NY, U.S.A., 2003; 299–335.
11. Foster I. Globus toolkit version 4: Software for service-oriented systems. *IFIP International Conference on Network and Parallel Computing*, vol. 3779. Springer: Berlin, 2005; 2–13.
12. Chien A, Calder B, Elbert S, Bhatia K. Entropia: Architecture, performance of an enterprise desktop Grid system. *Journal of Parallel and Distributed Computing* 2003; **63**(5):597–610.
13. Levine D, Wirt M. Interactivity with scalability: Infrastructure for multiplayer games. *The Grid 2: Blueprint for a New Computing Infrastructure*, Foster I, Kesselman C (eds.). Morgan Kaufmann: Los Altos, CA, 2003; 167–178.
14. Sun Microsystems. Sun n1 grid engine 6. <http://www.sun.com/software/gridware/> [20 July 2007].
15. OGSA-WG. Defining the Grid: A roadmap for OGSA standards. <http://www.gridforum.org/documents/GFD.53.pdf> [20 July 2007].
16. OASIS Consortium. Web services resource framework (WSRF)—primer v1.2. committee draft 02. <http://docs.oasis-open.org/wsrp/wsrp-primer-1.2-primer-cd-02.pdf> [20 July 2007].
17. Vaughan-Nichols SJ. Web services: Beyond the hype. *Computer* 2002; **35**(2):18–21.

18. Stockinger H. Defining the Grid: A snapshot on the current view. *Journal of Supercomputing* 2007; DOI: 10.1007/s11227-006-0037-9.
19. Foster I. What is the Grid? a three point checklist. *Grid Today* 2002; **1**(6).
20. Taylor IJ. From P2P to web services and Grids: Peers in a client/server world. *Computer Communications and Networks*. Springer: Berlin, 2005.
21. Chetty M, Buyya R. Weaving computational Grids: How analogous are they with electrical Grids? *Computing in Science and Engineering* 2002; **4**(4):61–71.
22. Bal H, Casanova H, Dongarra J, Matsuoka S. Application-level tools. *The Grid 2: Blueprint for a New Computing Infrastructure*, Foster I, Kesselman C (eds.). Morgan Kaufmann: Los Altos, CA, 2003; 463–489.
23. Kielmann T, Merzky A, Bal H, Baude F, Caromel D, Huet F. Grid application programming environments. *Future Generation Grids*. Springer: Berlin, 2006; 286–306.
24. Baker M, Buyya R, Laforenza D. Grids and Grid technologies for wide-area distributed computing. *Software—Practice and Experience* 2002; **32**(15):1437–1466.
25. Venugopal S, Buyya R, Ramamohanarao K. A taxonomy of data Grids for distributed data sharing, management, and processing. *ACM Computing Surveys* 2006; **38**(1):1–53.
26. Krauter K, Buyya R, Maheswaran M. A taxonomy and survey of Grid resource management systems for distributed computing. *Software—Practice and Experience* 2002; **32**(2):135–164.
27. CERN. The GridCafé project. <http://gridcafe.web.cern.ch/gridcafe/> [20 July 2007].
28. Arnold A, Gosling A. *The Java Programming Language*. Addison-Wesley: Reading, MA, U.S.A., 1996.
29. GRIDS Laboratory. The GridBus project. <http://www.gridbus.org> [20 July 2007].
30. Foster I, Kesselman C. Concepts and architecture. *The Grid 2: Blueprint for a New Computing Infrastructure*, Foster I, Kesselman C (eds.). Morgan Kaufmann: Los Altos, CA, 2003; 37–63.
31. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press: Cambridge, MA, 1994.
32. Dongarra J, Walker D. MPI: A standard message passing interface. *Supercomputer* 1996; **12**(1):56–68.
33. Bryan Downing T. *Java RMI: Remote Method Invocation*. IDG Books Worldwide: Foster City, CA, U.S.A., 1998.
34. Karonis N, Toonen B, Foster I. MPICH-G2: A Grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing* 2003; **63**(5):551–563.
35. Nakada H, Matsuoka S, Seymour K, Dongarra J, Lee C, Casanova H. A GridRPC model and API for end-user applications. *Technical Report*, GridRPC Working Group, July 2005.
36. Johnson RE. Frameworks = (components + patterns). *Communications of the ACM* 1997; **40**(10):39–42.
37. Codenie W, De Hondt K, Steyaert P, Vercammen A. From custom applications to domain-specific frameworks. *Communications of the ACM* 1997; **40**(10):71–77.
38. Globus Alliance. The Java CoG kit. http://wiki.cogkit.org/index.php/Java_CoG_Kit [20 July 2007].
39. Allen G, Davis K, Dolkas KN, Doulamis ND, Goodale T, Kielmann T, Merzky A, Nabrzyski J, Pukacki J, Radke T, Russell M, Seidel E, Shalf J, Taylor I. Enabling applications on the Grid: A GridLab overview. *International Journal of High Performance Computing Applications, Special issue on Grid Computing: Infrastructure and Applications* 2003; **17**(4):449–466.
40. Allen G, Davis K, Goodale T, Hutanu A, Kaiser H, Kielmann T, Merzky A, van Nieuwpoort RV, Reinefeld A, Schintke F, Schott T, Seidel E, Ullmer B. The Grid application toolkit: Towards generic and easy application programming interfaces for the Grid. *Proceedings of the IEEE* 2005; **93**:534–550.
41. Goodale T, Jha S, Kaiser H, Kielmann T, Kleijer P, von Laszewski G, Lee C, Merzky A, Rajic H, Shalf J. SAGA: A simple API for Grid applications—high-level application programming on the Grid. *Computational Methods in Science and Technology* 2006; **20**(1):7–20.
42. Goux J-P, Kulkarni S, Linderoth J, Yoder M. An enabling framework for master–worker applications on the computational Grid. *IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, Pittsburgh, PA, U.S.A. IEEE Computer Society: Silver Spring, MD, 2000; 43–50.
43. Berman F, Wolski R, Casanova H, Cirne W, Dail H, Faerman M, Figueira S, Hayes J, Obertelli G, Schopf J, Shao G, Smallen S, Spring N, Su A, Zagorodnov D. Adaptive computing on the Grid using AppLeS. *IEEE Transactions on Parallel Distributed Systems* 2003; **14**(4):369–382.
44. Ferreira JF, Sobral JL, Proenca AJ. JaSkel: A Java skeleton-based framework for structured cluster and Grid computing. *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, Washington, DC, U.S.A. IEEE Computer Society: Silver Spring, MD, 2006; 301–304.
45. Ho Q, Ong Y, Cai W. Gridifying aerodynamic design problem using GridRPC. *Grid and Cooperative Computing—GCC 2003 (Lecture Notes in Computer Science, vol. 3032)*, Li M, Sun X, Deng Q, Ni J (eds.). Springer: Berlin, 2003; 83–90.
46. Wang B, Xu Z, Xu C, Yin Y, Ding W, Yu H. A study of gridifying scientific computing legacy codes. *Grid and Cooperative Computing—GCC 2004 (Lecture Notes in Computer Science, vol. 3251)*, Jin H, Pan Y, Xiao N, Sun J (eds.). Springer: Berlin, 2004; 404–412.
47. Kolano PZ. Facilitating the portability of user applications in grid environments. *Distributed Applications and Interoperable Systems, 4th IFIP WG6.1 International Conference (Lecture Notes in Computer Science, vol. 2893)*, Stefani J, Demeure IM, Hagimont D (eds.). Springer: Berlin, 2003; 73–85.

48. Delaitre T, Kiss T, Goyeneche A, Terstyanszky G, Winter S, Kacsuk P. GEMLCA: Running legacy code applications as Grid services. *Journal of Grid Computing* 2005; **3**(1–2):75–90.
49. Kacsuk P, Sipos G. Multi-Grid, multi-user workflows in the P-GRADE Grid portal. *Journal of Grid Computing* 2005; **3**(3–4):221–238.
50. Vadhiyar S, Dongarra J. Self adaptability in Grid computing. *Concurrency and Computation: Practice and Experience (Special Issue on Grid Performance)* 2005; **17**(2–4):235–257.
51. Wolski R, Spring N, Hayes J. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems* 1999; **15**(5–6):757–768.
52. Ho Q, Hung T, Jie W, Chan H, Sindhu E, Ganesan S, Zang T, Li X. GRASG—A framework for ‘gridifying’ and running applications on service-oriented Grids. *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*. IEEE Computer Society: Silver Spring, MD, 2006; 305–312.
53. W3C Consortium. SOAP version 1.2 part 0: Primer. W3C Recommendation. <http://www.w3.org/TR/soap12-part0/> [20 July 2007].
54. Allcock B, Bester J, Bresnahan J, Chervenak AL, Foster I, Kesselman C, Meder S, Nefedova V, Quesnel D, Tuecke S. Data management and transfer in high-performance computational Grid environments. *Journal of Parallel Computing* 2002; **28**(5):749–771.
55. Maia PHM, Mendonca NC, Furtado V, Cirne W, Saikoski K. A process for separation of crosscutting Grid concerns. *Proceedings of the ACM Symposium on Applied Computing*, New York, NY, U.S.A. ACM Press: New York, 2006; 1569–1574.
56. Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J, Irwin J. Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-oriented Programming*, vol. 1241, Aksit M, Matsuoka S (eds.). Springer: Berlin, Heidelberg, New York, 1997; 220–242.
57. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W. Getting started with AspectJ. *Communications of the ACM* 2001; **44**(10):59–65.
58. Kommineni J, Abramson D. GridLeS enhancements and building virtual applications for the Grid with legacy components. *Advances in Grid Computing—EGC 2005 (Lecture Notes in Computer Science*, vol. 3470), Sloat PMA, Hoekstra AG, Priol T, Reinefeld A, Bubak M (eds.). Springer: Berlin, 2005; 961–971.
59. Takemiya H, Shudo K, Tanaka Y, Sekiguchi S. *Constructing grid applications using standard grid middleware*. *Journal of Grid Computing* 2003; **1**(2):117–131.
60. Pope AL. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley: Boston, MA, U.S.A., 1998.
61. Webb D, Wendelborn AL. The PAGIS Grid application environment. *International Conference on Computational Science (Lecture Notes in Computer Science*, vol. 2659), Sloat PMA, Abramson D, Bogdanov AV, Dongarra J, Zomaya AY, Gorbachev YE (eds.). Springer: Berlin, 2003.
62. Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M, Quilici R. Programming, deploying, composing, for the Grid. *Grid Computing: Software Environments and Tools*. Springer: Berlin, 2006; 205–229.
63. van Nieuwpoort RV, Maassen J, Kielmann T, Bal HE. Satin: Simple and efficient Java-based Grid programming. *Scalable Computing: Practice and Experience* 2005; **6**(3):19–32.
64. van Nieuwpoort RV, Maassen J, Wrzesinska G, Hofman R, Jacobs C, Kielmann T, Bal HE. Ibis: A flexible and efficient Java based Grid programming environment. *Concurrency and Computation: Practice and Experience* 2005; **17**(7–8):1079–1107.
65. Gannon D, Krishnan S, Fang L, Kandaswamy G, Simmhan Y, Slominski A. On building parallel and Grid applications: Component technology and distributed services. *Cluster Computing* 2005; **8**(4):271–277.
66. Armstrong R, Gannon D, Geist A, Keahey K, Kohn S, McInnes L, Parker S, Smolinski B. Toward a common component architecture for high-performance scientific computing. *IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society: Silver Spring, MD, 1999; 115–124.
67. Jugravu A, Fahringer T. JavaSymphony, a programming model for the Grid. *Future Generation Computer Systems* 2005; **21**(1):239–247.
68. Neary MO, Cappello P. Advanced eager scheduling for Java-based adaptive parallel computing. *Concurrency and Computation: Practice and Experience* 2005; **17**(7–8):797–819.
69. Huedo E, Montero RS, Llorente IM. A framework for adaptive execution in Grids. *Software—Practice and Experience* 2004; **34**(7):631–651.
70. Englander R. *Developing Java Beans*. O’Reilly & Associates, Inc.: Sebastopol, CA, U.S.A., 1997.
71. Huhns MN, Singh MP. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing* 2005; **9**(1):75–81.
72. Atkinson M, DeRoure D, Dunlop A, Fox G, Henderson P, Hey T, Paton N, Newhouse S, Parastatidis S, Trefethen A, Watson P, Webber J. Web service Grids: An evolutionary approach: Research articles. *Concurrency and Computation: Practice and Experience* 2005; **17**(2–4):377–389.
73. W3C Consortium. Web services description language (wsdl) version 2.0 part 1: Core language. W3C Candidate Recommendation. <http://www.w3.org/TR/wsdl20/> [20 July 2007].

74. OASIS Consortium. Uddi version 3.0.2. UDDI Spec Technical Committee Draft. http://uddi.org/pubs/uddi_v3.htm [20 July 2007].
75. Alonso JM, Hernández V, Moltó G. GMarte: Grid middleware to abstract remote task execution. *Concurrency and Computation: Practice and Experience* 2006; **18**(15):2021–2036.
76. Johnson R. J2EE development frameworks. *Computer* 2005; **38**(1):107–110.
77. Mateos C, Zunino A, Campo M. JGRIM: An approach for easy gridification of applications. *Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications* 2007; DOI: 10.1016/j.future.2007.04.011.