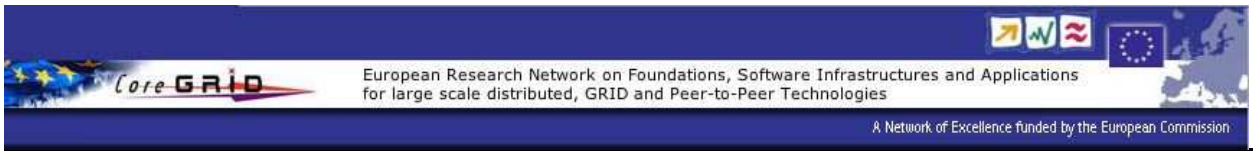

The Grid Checkpointing Architecture

Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak
{gracjan,radekj,rafal.mikolajczak}@man.poznan.pl
Poznan Supercomputing and Networking Center
61-704 Poznan, Noskowskiego 12/14, Poland

Jozsef Kovacs,
smith@sztaki.hu
Computer and Automation Research Institute of Hungarian Academy of Sciences
1111 Budapest Kende u. 13-17. Hungary

CoreGRID White Paper
Number WHP-0003
May 13, 2008

Institute on Grid Information, Resource and Workflow Monitoring Services

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

The Grid Checkpointing Architecture

Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak
{gracjan,radekj,rafal.mikolajczak}@man.poznan.pl
Poznan Supercomputing and Networking Center
61-704 Poznan, Noskowskiego 12/14, Poland

Jozsef Kovacs,
smith@sztaki.hu
Computer and Automation Research Institute of Hungarian Academy of Sciences
1111 Budapest Kende u. 13-17. Hungary

CoreGRID WHP-0003
May 13, 2008

Abstract

Introduction of checkpointing into the Grid environment is difficult. The difficulty of integrating the checkpointing technology is imposed by the distributed and heterogeneous nature of the Grid environment and by the close links of the checkpointing tools to the hardware. This paper scrutinizes the most important problems that have to be solved to prepare a fully functional environment that is able to utilize the full potential of the checkpointing technology for improving fault tolerance and system management in the Grids. The scope of considered issues covers both the high level issues concerning placement of the checkpointing in the Grid architecture with regard to the role this service plays in the Grid, and more technical problems as well. The solutions for all the mentioned problems are proposed, taking into account the complexity of the implementation on the one hand, and delivered functionality on the other.

The general architecture of the Grid Checkpointing Service is presented. The interconnections and interactions between internal modules and external Grid services are described taking into account the experience gained during the research and experiments on proof-of-concept implementations. The proposed model of the grid-enabled checkpointer should allow for seamless integration of different checkpointers with the Grid environment.

The intended reader of the paper is a person willing to have a checkpointing service in the operational Grid environment or someone willing to gain knowledge about the checkpointing on both legacy and Grid-level. The basic knowledge on topics related to the checkpointing technology issues and Grid internals is required.

1 Introduction

Contemporary Grid deployments feature a growing number of nodes and increasing capabilities of each node. This rapid growth results in an increasing performance of the Grid computing infrastructure which is a good thing, however, along with the growing computing power grows the complexity of the system management and the computation infrastructure itself. Each crash of the system, even every scheduled downtime introduces significant costs in terms of user computation time and this costs should be reduced to minimum.

Because of the sheer size of such an installation the failure rate grows along with the number of elements in the infrastructure. As a result, the fault-tolerance features are getting more and more important and are becoming a necessity in modern, large scale computing infrastructures. Due to the reasons mentioned above, it may be hard to assure an adequate level of reliability of the environment to guarantee that the computation will end in a desired time

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

frame, which is one of the key factors from the user's point of view when deciding whether to use a certain solution or not.

There are a few possible ways to enhance the level of fault tolerance and robustness in the computing environment. One of such solutions is the checkpointing mechanism. This mechanism has been well known for some time, and there are even some solutions already available. However, the existing implementations can not be directly deployed in the Grid infrastructure. The variety of approaches to the implementation of the checkpointing mechanism together with varying functionality and interface are the main obstacles when it comes to use these tools in the Grid. To make the subject even more difficult, the checkpointing mechanism is a low-level feature (from the point of view of Grid architecture), which is tightly bound with the underlying operating system and does not provide grid-related interfaces, whereas the mechanism should be accessible by the high-level services such as brokers. This "gap" or, in other words, lack of middleware services is a very important reason why such a desired mechanism lacks proper support in the contemporary grid solutions. This white paper strives to depict the most meaningful difficulties one has to overcome to provide the Grid with the checkpointing functionality. The presented issues follow from the experience gained during the work on the specification and the proof-of-concept, partial implementation of the Grid Checkpointing Architecture (GCA).

When introducing a new functionality to an existing environment, the first step is to prepare an architecture that will blend with the existing environment's architecture to provide a seamless interaction with the current components. The prepared architecture should be general enough to fit into any implantation and, on the other hand, give a clear suggestion what components will be needed and what the tasks and responsibilities of each element of the architecture.

Because the tests and proof-of-concept installations were mostly based on the Globus Toolkit 4 [17], it is a point of reference for all the remarks regarding the Grid services in this paper. The architecture in general, however, is not bounded with any specific middleware and does not impose any restrictions on what software is used to provide required functionality, therefore other Grid solutions such as Unicore can be used as well.

2 Overview of current approaches

This section presents the key implementation methods of checkpointing, both on the local and on the Grid levels.

2.1 Low level checkpointers

Up to now there have been a few approaches to the implementation of checkpointing mechanisms. Those mechanisms were implemented to checkpoint different application types, and are characterized by a completely different set of supported mechanisms and interfaces. The checkpointing mechanisms were implemented as a part of the operating system kernel (system or kernel level checkpointers); as statically or dynamically linked libraries (user level checkpointers); or as part of the application itself (application level checkpointer). Each of the mentioned approaches, and in fact even different checkpointers of the same type, handles storing and restoring of the application state in a different way and supports applications that are using various resources and mechanisms.

The kernel level checkpointing is characterized by the greatest level of flexibility. As long as the application sticks to the set of mechanisms supported by the checkpointer, it will be checkpointed and, most probably, successfully restarted. The application state can be stored regardless of the original programming language or availability of the source code, available libraries etc. The biggest drawback of this solution is a limited set of supported features, since some mechanisms, such as network connections, are very hard to handle in a general way. The psnC/R [12], chpox [13], CryoPID [16], OpenVZ [14] are examples of the implementation of such approach. There is also a separate software group that cannot be called a dedicated checkpointer but is providing the checkpointing functionality as one of its inherent features. The technology is based on the concept of isolated *Virtual Machines* (VM) which residing within an operating system provides the application with an illusion of running on the dedicated hardware with possibly different operating system. Usually this software is able to freeze the entire machine state to a file on the persistent storage. From our point of view it is an equivalent of the system level checkpointing with additional overhead on the image size as the entire OS must be stored along with the application. The XEN [6], KVM [7], VMware [18] can act as example applications for this software class.

The application level checkpointing can be perceived as the opposite solution to the system approach it is a part of the application. Since the checkpointing is "hard coded" into the application itself, it is treated as the additional feature of the application. Due to integration with the application, this solution is characterized by the best knowledge

of the state of the application and is able to store the application image in the most efficient way. There is also one unique feature that only the application level checkpointing is able to provide, namely the possibility of restarting the application in the heterogeneous environment. Such functionality is a trade-off for extremely limited scope of supported applications. Implementation of this mechanism in one application cannot be reused in other. Additionally, a very detailed knowledge of the application and access to the source codes is required in order to be able to implement this mechanism, and the effort has to be repeated for each application separately. The Gaussian may be an example of application with the checkpointer implemented as inner feature [15].

The next approach, that is the user or library level checkpointing, can be perceived as a hybrid of kernel and application level approaches. In most cases the checkpointing tool is provided as a library that has to be statically or dynamically linked against the application. This approach shares both the advantages and disadvantages of the two previously mentioned solutions. For example, it is possible to support preservation of the network connections state exactly like in the application approach, and on the other hand, use one library to handle many applications similarly to the system-level solution. However, it is programming language dependent and sometimes requires changes in the source code of the checkpointed application. In some cases it may be impossible to use the library because of the way the application was compiled or linked. The library level checkpointing was implemented e.g. in Condor [9], Libckpt [10] and psncLibCkpt [11].

2.2 Grid level checkpointers

There are at least two different approaches to the checkpointing in the Grid environment. The first one relies on the mechanisms implemented into the application to checkpoint its state as a reply to a request from the management service or independently, according to some internal policies, e.g. in a fixed time interval. This solution is pretty similar to the user and application level approach mentioned in the previous section, except that the library provides a set of commands that allow for the interaction between the grid services and the application in order to store the checkpoint images and grants access to the meta data associated with the checkpoint images. This approach is developed by the Open Grid Forum body, and is described in detail in: "An Architecture for Grid Checkpoint and Recovery Services" [1]. The advantages and drawbacks of this architecture of the Grid checkpointing are the same as the ones mentioned for the library approach for non-grid enabled checkpointers in the previous section.

Contrary to the previously mentioned model, there is also an approach focused on re-using the existing low-level checkpointers. It can be done by exposing functionality of the low-level checkpointers to higher level services such as resource managers or brokers that should be able to utilize the underlying features to help system management or to reduce the cost of the system or hardware failures to a minimum. An obvious advantage of such an approach is re-using the existing software to enhance the functionality of the Grid environment instead of going through the painful process of writing the most basic software bricks from scratch. It is, however, not an easy task since so far it has been impossible to integrate such modules in a straightforward way, especially if a fully integrated environment that is able to utilize the full potential of the checkpointing technology is the goal we are striving for.

It is also possible to prepare a solution that will use some low-level checkpointers to handle applications running on the computing resource in a way that the grid will be "half-aware" of the checkpointing mechanism. An example of such a solution may be the implementation presented [4], where the Grid broker interacts not directly with the checkpointer to but exploits the functionality of a local resource manager that allows for the interaction with the locally installed checkpointer for fault tolerance purposes. The logic behind the checkpoint and restart scenario was encoded in the description of the application workflow. Therefore, such solution cannot be treated as a fully integrated computing environment. In spite of the differences, all the solutions share a similar set of requirements that must be fulfilled to make the checkpointer able to handle Grid applications. A more detailed insight into the issues related to the full integration, along with possible solutions, will be mentioned in the following sections.

3 Research context

The checkpointing service is by its nature a low-level, hardware or application-bound feature with various ways of implementation, and possibly with different functionality. Therefore, the work on the introduction of this mechanism in the Grid system requires solving many problems, the most important of which are briefly presented below. The overview covers the general problems such as the right placement of the service in the Grid environment as well as the practical problems that have to be solved to make the system serve its purpose. The paper focuses on the problems

caused by the Grid environment nature and does not discuss the problems related to the checkpointing itself, such as the consistency problems, preserving of communication channels etc.

The checkpointing service can be perceived in several ways, depending on the role it fulfills in the environment. It can be treated as a low-level operating system level feature that ensures that once the application is submitted, it will finish the calculation against any odds that may happen during the execution time. This approach can hardly be called a grid checkpointing as the checkpointer is neither able to use any of the Grid services, nor plays any significant role in the Grid management. It is, however, very simple to deploy such a configuration, and it is useful, even if such installation provides limited functionality.

In order to extend the functionality and enable the features such as fault tolerance on the Grid level, the migration of applications between the computing resources for the management and load balancing purposes, some additional work has to be done. This approach requires a lot more effort to achieve the desired functionality. The benefits, however, make it worth working on it.

3.1 Checkpointing in the Grid environment

All entities building up the Grid infrastructure are either resources, resource properties or services, so the grid checkpointer has to be either of those. In the original concept presented as the Grid Checkpointing Architecture in [2] and [3], the checkpointing service is, as the name implies, a Grid service that is registered and accessed exactly like any other service in the Grid. The overall approach is based on the paradigm of layered division of responsibility. The set of reference layers has been defined in the GCA specification. These layers are depicted in the figure 1. It is noteworthy that, according to the current functionality of individual Globus components and Grid Brokers, the actual partial or complete implementation of GCA could not comply with the reference model in every single point. Following the ideas formed in the GCA, in order to perform a checkpoint or restart, the client of the checkpointing service should call it and the high level Grid service will delegate the request to the appropriate low-level checkpointer, providing all the necessary parameters and ensuring all the conditions required by the checkpointer.

The original concept proved to be very hard to implement without a vast number of changes in the Grid architecture as the contemporary grid resource brokers do not offer suitable features required to implement the desired functionality.

Because the scope of action of the local checkpointer is in most cases only a physical machine on which the checkpointer is installed and not all the checkpointers are able to handle all applications, the information about the installed checkpointers has to be taken into consideration by the resource broker while dispatching the job to the concrete computing resource. Unfortunately, the contemporary Grid Resource Brokers fail to do the matching of the multiple resource types for a single application. Currently, the Broker is looking for the appropriate computing resource, which is the only resource type it is able to look for that all the requirements stated in the job description. It is not able to look for the node that, apart from other required parameters has also a desired service installed. This proved to be a conceptual problem however, as the scope of our work was focused on reusing the already existing checkpointers which by nature are bound to the computing node, the solution was to include a description of the locally installed checkpointer as another resource such as the operating system or the CPU. This approach simplified the general implementation but was not 100% in line with the original idea.

3.2 Resource description

Regardless of the general placement of the checkpointing service in the Grid architecture, at some point it is necessary to prepare a description of the capabilities of a local checkpointer that will make it usable by the higher level services. As mentioned in section 2.1, the low-level checkpointers may vary from each other in a significant degree, therefore each of them has to be precisely described. For each single checkpointer the Grid must be provided with the information regarding the following subjects: the identification information such as name, version etc., a description of supported mechanisms and requirements put on the machine where the application will be restarted. The innate diversity among the checkpointers is the reason why it is hard to find an unambiguous method of description of the mentioned parameters. In this section we will take a closer look at each of the subjects mentioned above.

In order to make the Grid checkpoint-enabled, the description of the low-level part has to be unified. The description of the supported mechanisms proved to be especially difficult. This information is crucial for the checkpointing service since creating an image with a tool that is not able to store all the information of the application state will most probably lead to crash during the restart or, worse still, it can be the cause of the results corruption. The more

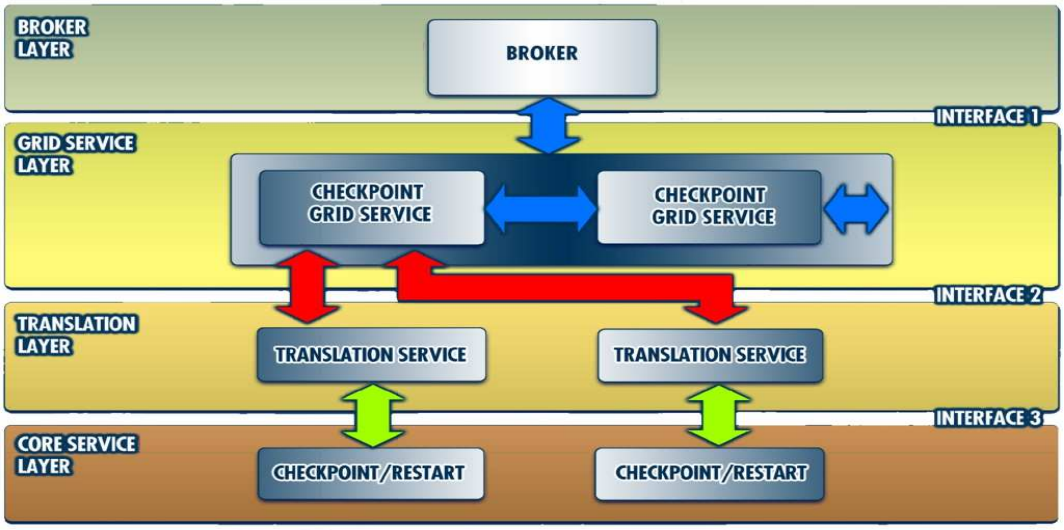

Figure 1: Grid Checkpointing Architecture

information on the checkpointer features and the application internals we have, the more accurate decisions can be made.

The name of the checkpointer is the most trivial information from the set of data that describes the checkpointer. The only requirement is that the name should be unique to allow for the identification of the checkpointer.

A question arises when we have to decide how to describe the information on the mechanisms the checkpointer is able to handle. A simple list of the supported applications could be an obvious approach, which is certainly easy to use and human-readable. On the other hand, however, it is not very flexible and extensible because it does not show whether an application that is not present on the list is not supported or has not been tested. An alternative way may take the form of a list of the low-level application-related information that is stored in during the checkpoint process. Such solution gives more exact and more flexible information, but the number of users that are able to use the information in this form is very small because it requires in-depth knowledge on the application and sometimes even of the underlying hardware. As a trade-off between functionality and usability, a special requirement can be added to the job description. This parameter describes in an aggregated way what the checkpointer is capable of. Every checkpointer is assigned to one of the classless with self-describing names such as “parallel job”, “mpi application”, etc. The most complex part of the low-level checkpointer-related metadata is the part that describes the requirements put on the machine that will be used for restarting the checkpointed job. The metadata must grasp all the dependencies between the operating system, the environment configuration, the checkpointer and the application.

The requirements can be divided into two groups. The first part is a set of requirements that have to be taken into consideration by the Grid broker when looking for the node that will run the restarted application. This information includes the requirements on the installed libraries, the operating system version, the hardware type etc. Such information is a simple extension of the normal job description because it relates to resources that may be considered by the broker even without checkpointing. The second group of the dependencies consists of the conditions that have to be considered locally on the destination node. The environment variables or special parameters passed to the checkpointer are examples of such information. This group encompasses the part of the actions that in the GCA a checkpoint or restart script takes care of. Because of the possible variety of the requirements and the required flexibility, we did not point out any particular solution. In the deployment installation we used a checkpointer that required the same operating system, the node architecture and checkpointer in the same version as the one that was used on the original node [4]. This approach proved to be sufficient for our purposes but is not as general as we would wish for. During the study on the checkpointing subject we did several proof-of-concept installations but even the most advanced ones required the user to provide a special parameter in the job description. A desired solution should, however, not require a special input indicating the fact that a checkpointer should be used. The fault tolerance should be taken into consideration by the dispatching broker and it should be done without the user consideration. The information on the class of the application or, in other words, the checkpointer that should be used, should be possible to include in the job description as a hint for the scheduler. Additionally, the resource broker must be able to take into consideration multiple resource

types rather than a single resource type such as computing resource, as it is the case with the contemporary brokers.

3.3 Application image handling

There are several conditions that have to be fulfilled for the checkpoint to serve its purpose. One of the most basic is the accessibility of application images. In a non-grid environment this condition was fulfilled if the data survived the crash of the computer and the application was resumed after the machine was up again. In case of the Grid environment it is not so simple, as the images have to be accessible even if the original computing resource is still unavailable. To ensure the survival and usability of the images, one has to deal with several issues such as finding a right place to store the files, efficiently transfer data and register meta-data about the images in the information service. In other words, there is a need for a reliable grid storage service as well as globally accessible information service able to store data dynamically during the application execution. But it is not enough. So far there has been no grid storage service available that would grant the required storage space in a similar way to the computing power. Therefore, the storage space has to be provided individually for each deployment to be able to transfer the images by means of the already accessible Grid tools like the gridftp. We have to remember that the images can be very large and the process of sending them to the distant nodes can introduce unacceptable costs in terms of network bandwidth, especially when the storage is placed on a distant resource.

Creating the image of the application state is not enough to assure that the application could be restarted properly. In order to make the computing environment more fault tolerant, the image has to be stored in a globally accessible storage and the event of image creation has to be stored as a metadata of the application. The lack of grid awareness from the low-level checkpointer side together with the different scenarios of image creation (the creation of the images may be triggered [4] or independent) may lead to a situation where the checkpoint will be created properly, but the Grid broker will not be notified and subsequently not able to determine whether the restart is possible or not. To make the images usable, there has to be a service that would register the data on each checkpoint when the images are created. In the test deployments we have used an OGSA-DAI -based solution to ensure that the information can be accessed in fashion that allows both for reading and writing, and ensures data consistency.

The management of the life-cycle of the images is another problem that has to be solved. The files containing the state of the application may have a significant size. Therefore, proper image handling can spare a lot of storage. If the images are stored locally on the node where the application was running and finished successfully, the decision whether remove the image or not can be made locally, but again, if the removal was not issued from the higher management service, it may destroy the information needed to migrate the application. The image management should be done if not by the Broker itself, then by a service notified by the Broker of the state of the application. Because the application images are valid only for a certain amount of time, which is at most the application lifetime, there should be a mechanism that will remove the image after its lifetime is over. Such a mechanism working independently of the broker ensures that the storage space will not be wasted in case of problems with the Broker.

3.4 Executing the checkpoint

There are several issues that have to be addressed to execute the checkpoint of the job running in the Grid. The most important ones will be discussed in this section.

In the most common checkpoint usage scenario, the resource management service, according to some internal policy, issues the checkpoint of the managed application.

The original GCAs idea how the checkpointer should be accessed was based on a logically central checkpointing service that should act as a dispatcher of any request related to the checkpointing functionality. The first issue that has to be addressed is to call a non-grid checkpointer from a Grid service. The problem with accessing the non-Grid-enabled checkpointers is well described in: [2] and [3]. The proposed solution of preparing a service that, using a internal interface, connects with the wrapper that exposes a unified checkpointer interface for other Grid Services seemed to be a simple and straightforward solution until the first approach to create real-life implementation. The proposed solution worked fine with the computing resources such as large SMP machines, but got complicated when a job was submitted to a cluster of nodes where the only node exposed to the Grid was the access node, and the computing nodes were configured with a private set of addresses. Because the computing nodes cannot expose any services, direct implementation of the GCA architecture does not come in question. Such situation is not uncommon because, due to security and maintenance reasons, the worker nodes are not exposed to have access to the external network and therefore neither can use the external Grid services (e.g. to register images or access point to the service)

nor be accessed from Grid services such as brokers. Providing a general solution that will make such node accessible without compromising the security of the cluster can be a challenging task and has to be done individually for each deployment and cannot be solved at the architecture level. The implementation we prepared handled this subject by introducing additional functionality to the wrapper that was able to do all the necessary operations remotely on any node and collect the results. Unfortunately, this approach introduced the additional overhead on some of the operations (see section 3.3).

The process of creating an image requires not only access from the Grid to the local resource but also vice versa. Some of the external Grid services have to be accessed, e.g. to register the information about the checkpoint. Therefore the communication channel must be established between any computing node and the rest of the Grid environment. Those issues were presented in the previous section.

As mentioned in section 2.1, the low-level checkpointers are characterized by different functionality and different sets of parameters, which is an undesired state from the Grid point of view. Even if there is an unified way of accessing the local checkpointers realized, e.g. in a way presented earlier, there are still problems with parameters that may require some knowledge that is not available on the high Grid level (such as the Broker) and we cannot assume that the lower layers expose the desired information to the other services. An example of such information may be the local process identifiers required by some checkpointers. In the GCA we proposed that in order to get information a submitted job would be executed not directly but in a sort of a sandbox. In practical terms this means that for each checkpointer there is a customized binary (sort of wrapper) that registers all the relevant information prior to running the actual application. This binary should be executed prior the actual application and the most obvious way of doing this is to replace the applications binary with the wrapper and passing the application name to the wrapper for later execution. The drawback of changing the original application binary name is the need for changes in the job description file. This is inconvenient for the user and it is possible to make such changes transparently, e.g. by the broker replacing the binary name in the job description with the desired wrapper binary name. It requires, however, implementation of this feature in the Broker.

3.5 Restarting the application

Restarting the application in a non-Grid environment is, apart from technological issues, a simple task. It boils down, in most cases, to execute either a special command (like resume in case of PSNC/R) or re-execute the original binary with some special parameter (LibCkpt is an example of such solution). Providing the images were correctly created, the application will resume computation from the point of time when the checkpoint image was created. The Grid environment implies some additional problems related to the distributed and heterogeneous nature of such environment. The application may be restarted on a distant node, therefore some additional conditions may occur that must be fulfilled in order to ensure that the restarted application will work properly. The most basic requirement is that the restart must be performed by the same checkpointing toolkit as the one that created the images. Unfortunately, due to the lack of standards in the area of checkpointing toolkits any further requirements depend strongly on the checkpointer that was used to take the checkpoint of the images. The new computing resource may have to have installed an application that was checkpointed, the application has to be provided with appropriate input, output files along with the images containing the intermediate computation results. Running a Gaussian application with its internal checkpointing mechanism may be an example of an internal checkpointer that requires to fullfill all the conditions listed above. There are, however, many more scenarios that require fulfilling multiple additional requirements. One of the most commonly encountered issues is the need to have the new destination nodes compatible with the original one on the hardware level because most of the checkpointers are very dependent on the processor architecture. This is the case for both the system and the user level checkpointers. The set of possible elements that have to be identical on the old and new computing resource consists of (but is not limited to) the operating system, both type and version, environment variables, file system topology, various resource identifiers such as network addresses, process identifiers, shared resource identifiers etc. All those requirements have to be fulfilled by the grid services responsible for the restart of the application so it is crucial to have an exact description of the specific checkpointing toolkit coupled together with a description of the application and the images, just to be able to prepare a sufficient environment. There are some solutions such as KVM, VMware, XEN which, when used, will solve the environment compatibility issue by introducing a special container abstractions that provide a separate execution environment. Using those solutions adds a requirement on having the same solution installed on the node the application will be restarted on.

Because some of those products offer a possibility of suspending the state of the entire virtual machine they may be treated as a special type of system-level checkpointer. The complete information about requirements set have to be

included in the checkpointers metadata (see section 3.2) and taken into consideration by the resource broker when it finds a node where the computation should continue.

Because the restart of the application in the Grid environment is in fact a standard submit scenario modified in a way that the application's binary name is exchanged by the binary name of the restarting command and there may be a need to perform an additional task locally on the destination node. Therefore it is handy to prepare a restarting script that is a tool customized for each checkpointer that will do all the necessary operations, such as downloading the images of the application, prior to running the actual application. This approach follows the guidelines from the GCA documents and proved to be a flexible solution.

3.6 Other issues

The introduction of the checkpointing service to the Grid requires interaction between several basic Grid services. Because the different parts that build up the Grid environment may significantly differ from each other in terms of the implemented features, the task of integrating may vary, depending on which tools are selected to build up the environment. An example of such a problem may be the case of integrating the GRMS [8] broker with the checkpointing service. As mentioned earlier, the GCA designed the architecture where the client asks the checkpointing service to perform a desired operation and the service will delegate the request. This model proved to be incompatible with the one implemented in the GRMS where the Broker should have been informed about an entry point of a checkpointer that should be used to checkpoint the job.

The difficult task of introducing the checkpointing in the Grid becomes even harder because the Grid core services do not always provide the necessary functionality. The initial GCAs assumption on how the Grid *Information Service* works proved to be wrong as the GIS in fact provides the read-only data, while the requirement stated in the GCA indicated that there is a need for some grid equivalent for the database with the read-write access. The need for the read-write access to a globally accessible repository is required in order to ensure an on-line reflection of the changes of the application state. The OGSA-DAI provided the required functionality. However, there are still issues in the area of integrating it with the certificates used by the Globus services.

4 Architecture description

In this section a general outline of the GCA is exposed. The presented architecture takes into account the problems mentioned in the previous sections and presents either a possible solution or a workaround for the mentioned issues. First, the operational environment that the GCA is intended to work in is presented. Next the individual components which constitute the GCA and their mutual relationships and the related design patterns are presented. Actually, the architecture presented in this section is part of Integrated Framework Architecture for the Grid Information, Resource and Workflow Services that is described in the D.IRWM.04 CoreGRID deliverable.

4.1 Operational, reference environment

The GCA is not a self-contained technology. It is intended to work and cooperate with a number of already existing Grid Services. Figure 2 shows the simplified Grid environment within which the GCA is to be embedded. One of the components that is shown in the figure is the *Grid Broker* that is in charge of finding the *Computing Resource* that meets the given job's requirements. The mentioned requirements are specified by the user and are placed in the job. Each *Computing Resource* is equipped with a set of resources (i.e. the CPUs, the available libraries, storage capabilities, RAM or some particular tools). The resources and their properties are published in the *Information Service*. The job descriptor contains a list of resources and their properties that the given job requires in order to be executed.

To find out the *Computing Resource* with a set of particular resources the *Grid Broker* enquires the *Information Service*. Next, the *Grid Broker* sends the user job to the chosen *Computing Resource*, and more accurately to the *Execution Manager* that provides access to the given *Computing Resource*. In fact, the *Execution Manager* is the interface to any Local Resource Manager (i.e. LSF, SGE, Torque, fork()) that is installed on the given *Computing Resource*. The *Grid Broker* or the *Execution Manager* can take advantage of any *Data Management Services* in order to prepare the conditions for executing the job or in order to preserve the results of the job. For instance, the input files can be staged in before the actual job is started. The *Grid Broker* assigns a unique Global Identifier (GID) to each submitted job. The GID distinguishes the given job and is used to all further interactions with it.

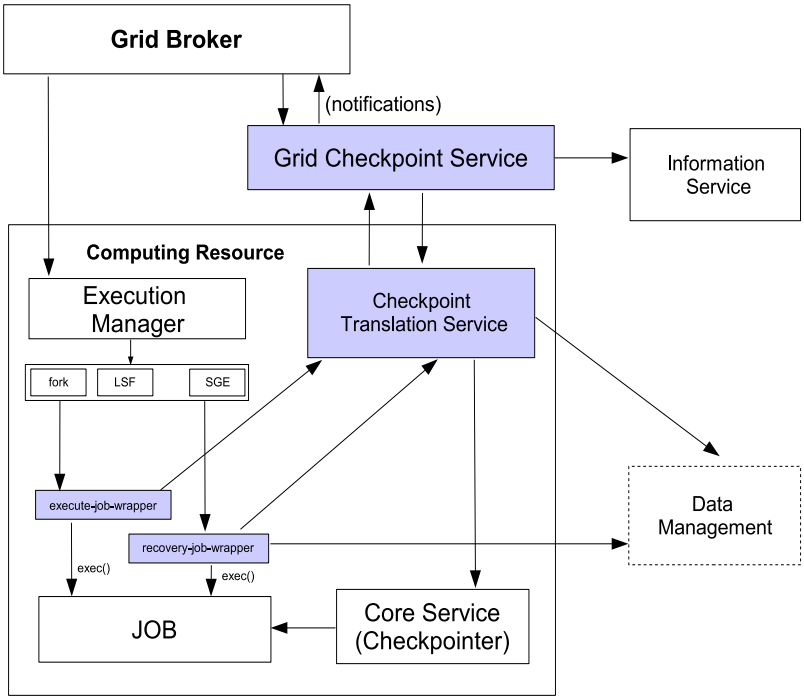

Figure 2: Grid Checkpointing Architecture

4.2 Core Service

The name Core Service refers to any checkpointing package that can be available on a given *Computing Resource*. In fact it is not the strict inner GCA element but rather any pre-existing low-level checkpointing package. The logical place of the Core Service within the GCA is depicted in Figure 2. The checkpointing functionality of a given Core Service can be implemented in various ways offering different functionality (see section 2.1).

4.3 Checkpoint Translation Service

The Checkpoint Translation Service (CTS) is a GCA-derived component and can be considered as the “driver” to the given Core Service. The CTS exposes to the Grid environment a uniform interface and is customized in respect of the underlying Core Service. There is no general CTS but rather a set of CTSes, each assigned and developed especially for a particular Core Service. The knowledge how to use the related Core Service is embedded into the CTS. The logical relation between the CTS and other GCA components is presented in the Figure 2.

The CTS is a component that, among other things, is responsible for handling the checkpoint requests, both internal and external ones. The external requests are triggered usually by the *Grid Broker* or by some other external component while the internal requests are triggered by the CTS or Core Service itself in a periodical way. How the external request is routed to the given CTS and how the internal checkpointing is triggered is presented in the sections 4.4 and 4.5. To handle the checkpoint request the CTS depends on its internal knowledge how to use the underlying Core Service. When checkpoint is finished, the CTS can store the image in any Data Replication Service. No matter whether the image is stored externally or not, the CTS has to assign the image a logical name or a unique identifier. The format of the logical name or the identifier is not imposed. It is merely required that the given CTS should be able to deal with it (for example with help of external Storage or Data Management services). If the given CTS, basing on the logical image name or identifier, is able to fetch the image from the remote locations, then during recovery activity it is possible to recover the job on the

Computing Resource different from the one that originally hosted the job. Additionally, all the information that is needed to properly perform and manage the job recovery stage has to be logged. Therefore, after the checkpoint is taken the CTS registers at Grid Checkpoint Service (see section 4.5) the following information: the unique identifier of the checkpoint, the sequence number of the checkpoint, the GID of the job being checkpointed, the type of the

involved CTS, date of the checkpoint action and the logical image name. In case of incremental images, log can contain additional information imposed by the semantics of incremental images.

The CTS can be considered as one of the resources that are available on a given *Computing Resource*. When the CTS is deployed it has to inform the Grid environment about its existence, its main properties and of the underlying Core Service (it means that this information has to be ultimately published in the *Information Service*).

Since the CTS is treated as a kind of the resource, the published properties can be used by the *Grid Broker* to match the given CTS with the requirements defined in a particular job descriptor.

Obviously the user may or may not put the checkpointing requirements into the job descriptor. If the checkpointing requirements are omitted, the job will simply not be checkpointed and does not need any CTS on the *Computing Resource* to be executed.

To make the user's life easier, the exact knowledge of all CTSes and related Core Services is not required. The user can define that his or her job is to be checkpointed by the checkpointer that belongs to the particular checkpointer class. All checkpointers that are members of the given checkpointer class have to adhere to the functionality (or, in other words, to the properties) imposed by this checkpointer class. From the point of view of GCA, the relation between the Core Service and the checkpointer class is defined on the level of CTSes. In other words, to add the given Core Service to the particular checkpointer class, the actual CTS that provides access to that Core Service has to be added.

The user, dealing with the checkpointer classess, focuses on the desired checkpointer functionality instead of particular implementations of CTSes or Core Services. From the point of view of the *Grid Broker*, coping with the checkpointer classes instead of the CTSes directly gives the opportunity to select a *Computing Resource* from a potentially larger pool. Finally, the job can be submitted to any *Computing Resource* that has deployed any CTS that has been registered as a member of the given checkpointer class.

It seems to be evident that to recover a job, the same type of Core Service that was used to take checkpoint has to be used. The GCA makes this rule even more restricted. The job can be recovered only on the *Computing Resource* that is equipped with the CTS of the same type that the one that was handling the checkpoint request. In some cases, due to a particular Core Service features, the recovery action can be undertaken only on the same *Computing Resource* on which the job was executed originally. If it is the case, then the published properties of the CTS related with the Core Service have to reveal this semantics.

The next responsibility of CTS is revealed during the recovery activity. The CTS is in charge of cooperating with the *recovery-job-wrapper* (described in section 4.4) in order to fetch the job image to the local node. The way the functionality is implemented does not matter. It is only important that the *recovery-job-wrapper* together with related CTS basing on the valid logical image name or on the image identifier have to be able to fetch the image. It is also possible that all the "image fetching" functionality is implemented only in *recovery-job-wrapper* or only in the CTS, depending on particular implementation of CTS and on demands of the underlying Core Service.

4.4 Auxiliary wrappers

Next GCA elements that are noticeable in Figure 2 are auxiliary wrappers. They fall into *execute-job-wrapper* and *recovery-job-wrapper* and are described in the subsequent subsections. Their main role is to perform some specific actions just before executing and recovering the job in order to keep the GCA in the coherent state. The assumption is that auxiliary wrappers are closely related and provided together with the CTSes. In some implementations it is possible that the wrappers functionality can be achieved with the help of standard mechanisms provided by the involved Execute Manager or by its underlying Local Resource Manager.

4.4.1 *execute-job-wrapper*

When the *Grid Broker* is about to submit the checkpointable job, basing on the list of available CTSes (the list can be obtained from the *Information Service*) and on the user supplied job's descriptor, it can match the job with the *Computing Resource* that is equipped with the CTS that is able to checkpoint the given job. The selected *Computing Resource* is accessible through the *Execution Manager* which receives the jobs submissions. However, if a job is to be checkpointed, then instead of submitting the job, a special *execute-job-wrapper* has to be submitted. Which execute-job-manager will be used depends on the involved CTS and can be obtained from the *Information Service* (the CTS had to publish it during the deployment activity). The *execute-job-wrapper* is passed the following arguments: the original job together with its original arguments, the GID assigned by the *Grid Broker* to the job, and any additional arguments according to the information published in the *Information Service*.

The main task of the *execute-job-wrapper* is to provide the GCA with the information about the relation between the GID of the user job and the EPR of the related CTS. It is important because another GCA component, basing only on the job GID will later be in charge of forwarding the doCheckpoint() request to the appropriate CTS. Additionally, the CTS has to know the mapping between the job GID and the identifier used locally by the given Core Service to trigger checkpoint of a particular job. In some cases it is sufficient to provide the CTS merely with information that will allow it to find out the actual identifier used by the Core Service later (when the doCheckpoint() request is handled). The CTS has to know this local identifier in order to properly instruct the Core Service which process has to be checkpointed.

To accomplish these tasks the first action the *execute-job-wrapper* performs is registering at the related CTS the relation between the GID and the identifier used locally by the Core Service. The GID is passed to the wrapper as one of its arguments but the knowledge how to obtain the local identifier used by the Core Service is embedded into the wrapper. However, in most cases the local identifier is the regular process identifier (PID) of the process that constitutes the job and the wrapper PID and the job's process PID are the same. The CTS remembers the registered relation and next informs the Grid Checkpointing Service (see section 4.5) that from now on the given CTS is responsible for handling all the external doCheckpoint() requests addressed to the job of the given GID.

Next the *execute-job-wrapper* has to convert itself into the actual job. To do so, the wrapper uses the exec() syscall and the parameters that were passed to the wrapper by the *Grid Broker*. Thanks to that the PID of the job and the wrapper are the same, so the relation between PID and GID that has been registered by the wrapper at CTS remains valid. From now on the job is executed in an ordinary way unless the related CTS receives the doCheckpoint().

4.4.2 *recovery-job-wrapper*

When a job is to be recovered, first the *Grid Broker* has to find the *Computing Resource* to which the job will be submitted. The rule is that the job can be recovered only on the *Computing Resource* with the deployed CTS of the same type that the one that was used to checkpoint the job. After the target *Computing Resource* is already known, the recovering process is almost as simple as resubmitting the given job to the *Execution Manager*. But instead of the original job the *recovery-job-wrapper* is submitted. The proper *recovery-job-manager* is recognized in a similar way to the *execute-job-wrapper* reorganization. The list of arguments of the *recovery-job-wrapper* includes the GID, the job itself, the original job's arguments and the identifier of the image that is to be used in the recovery process.

The first task of the *recovery-job-wrapper* is to fetch the job image. To accomplish this task the wrapper may (but does not have to) cooperate with the related CTS. It is the internal knowledge of the wrapper and related CTS how to obtain the image. The assumption is that the image is uniquely identified by the ID of the checkpoint image.

Next, the *recovery-job-wrapper* has to register at CTS the new relation between the GID and the identifier used locally by the Core Service. It is similar to the task that the *execute-job-wrapper* had to do. Finally, the *recovery-job-wrapper*, basing on its internal knowledge and with help of the given Core Service recovers the original job. From now on, until the CTS receives the doCheckpoint() request, the job is executed in an ordinary way.

But here we can find a pitfall. The *recovery-job-wrapper* cannot simply use the exec() syscall to exchange itself with the recovered job. It means that the semantics of some Core Services can disturb the assumption that the PID of the *recovery-job-wrapper* and the recovered job are the same. There are two possible solutions in such a case. First, the wrapper can register the relation between itself and the GID, and the CTS basing on its internal knowledge will be able to find out the correct identifier later (when the external doCheckpoint() request is handled). Second, the wrapper also registers the relation between its own PID and the job GID but additionally the wrapper will have to remain in the memory for the whole period during which the recovered job is executed.

Thanks to that the wrapper will be able to receive and forward all doCheckpoint() requests to the actual job. In most cases it means that the wrapper will have to forward all UNIX signals to the recovered job.

4.5 Grid Checkpoint Service

The headquarters of the GCA is the Grid Checkpoint Service (GCS). In the previous subsections it was expressed in a very general way that CTS has to inform the GCA about its existence and the main properties of the underlying Core Service. More precisely, the CTS registers all the information just at the GCS which further exposes a part of the information to the *Information Service*. The GCS is also the place where all the bookkeeping data (logs concerning the performed checkpoints) are registered and from where they are further published. Additionally, when the checkpointable job is being started by the *execute-job-wrapper* or by the *recovery-job-wrapper*, the related CTS registers

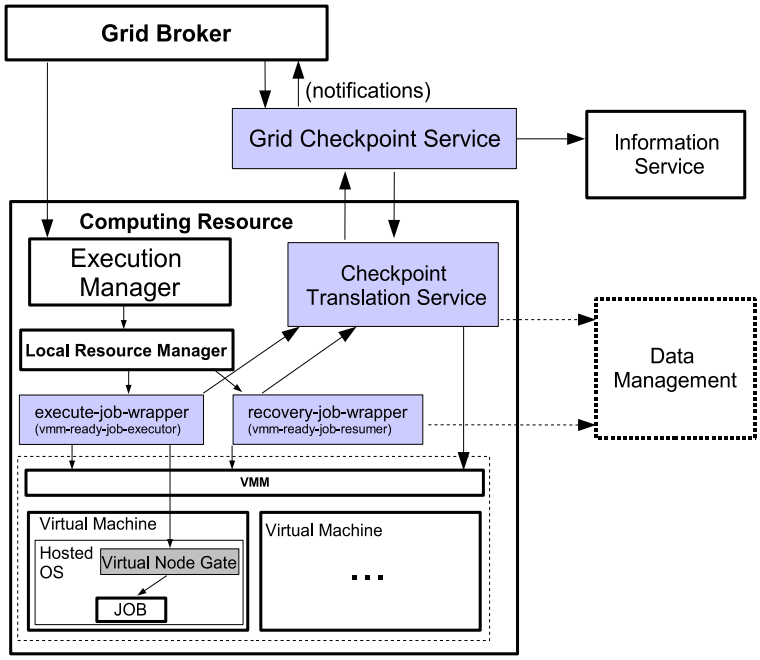

Figure 3: Grid Checkpointing Architecture with VMM

the mapping between the EPR of the CTS and the GID of the job just at the GCS. Thanks to that the GCS is a component that is able to forward the doCheckpoint() request to the appropriate CTS. So, the *Grid Broker*, to trigger the checkpoint of the job of the given GID, sends an appropriate request to the GCS which further knows how to route the request to the proper CTS.

Since the GCS is the central component through which all data flow goes, the external components can subscribe at it for event- or data-driven notifications. For example, the *Grid Broker* can be interested in being informed each time that the checkpoint of the given job is taken. It can be meaningful when the checkpoints are triggered by the Core Service itself instead of being triggered by the external doCheckpoint() request. In such a case,the GCS is able to notify the subscriber about the checkpoints because the adequate CTS is in charge of logging each checkpoint in the GCS. As it is shown in Figure 2 and as it results from the above description, the GCS interacts with the *Grid Broker* and CTSes. Additionally the GCS shares the part of its knowledge with the *Information Service*. As the *Grid Broker* has to interact directly with the GCS, the location of the GCS has to be known to the *Grid Broker*. Therefore, the GCS advertises its EPR through the commonly available *Information Service*.

4.6 Involvement of the *Virtual Machines*

The first approach of using the *Virtual Machines* (VM) int the GCA was to treat is as any other checkpointer but later study revealed that the differences imposed by this mechanisms require to introduce some dedicated changes in the architecture.The main responsibility of Virtual Machine Manger (VMM) is providing and managing the *Virtual Machines*. Each Virtual Machine can be treated as a virtual bubble that can , apart from other provided fuctionality, be suspended and later, basing on the saved image, resumed. There is an assumption that all the functionality that is required to suspend and later resume the VM is embedded into the VM or VMM. Thanks to that, when the Virtual Machine state is dumped to the persistent memory, the user jobs that were running on the VM are also “frozen”. Consequently, resuming the VM also causes resuming the user jobs. As it was mentioned in the introduction subsection, the GCA is very flexible. The assumption is that the CTS as well as the *execute-job-wrapper* and the *recovery-job-wrapper* can utilize any auxiliary tools or even Grid Services in order to provide the desired functionality. Therefore, since the VMM technology reveals new checkpointing possibilities and simultaneously the GCA is so flexible, we decided to present the way the VMM can cooperate with the GCA.

The concept of utilizing the VMM-based environment together with the GCA is depicted in Figure 3. The figure

can be considered as a particular scenario of a more general Figure 2. The Core Service has been replaced with the VMM that is in charge of managing the *Virtual Machines* (VM). The execute-job-manager and the *recovery-job-manager* are named *vmm-ready-job-executor* and *vmm-ready-job-resumer* respectively and a separate component that is named *Virtual Node Gate* appears. From the point of view of GCA the *Virtual Node Gate* can be considered as an auxiliary tool associated with the *execute-job-wrapper*.

The *vmm-ready-job-executor* is responsible for submitting the user job to a particular Virtual Machine. The way the *vmm-ready-job-executor* will find out or provide the Virtual Machine that will host the user job is not defined. There can be a pool of already running *Virtual Machines* or an individual Virtual Machine can be started up for each job separately. The *vmm-ready-job-executor* is able to execute the job on the Virtual Machine thanks to the *Virtual Node Gate*. The *Virtual Machines* (more precisely the Operating Systems deployed on these *Virtual Machines*) has to be appropriately configured to ensure that the *Virtual Node Gate* is automatically started up together with the Operating System. The *Virtual Node Gate* can be a piece of software dedicated for a given configuration or it can even be any preexisting software that will allow the *vmm-ready-job-executor* to execute the job on the Virtual Machine. We took an implicit assumption that the *Virtual Node Gate* can communicate with the *vmm-ready-job-executor* via network connections and, if it is not the case, we assume that VMM provides some way to communicate with the hosted Operating Systems.

When CTS receives the request to do checkpoint of the job of given GID, it has to forward the request to the appropriate VMM. Then the VMM will save the image of the Virtual Machine that is hosting the user job. In order to determine which VM has to be frozen, the CTS uses the information that was earlier provided by the *vmm-ready-job-executor*. As it was stated in a general description of GCA, the CTS can perform any additional actions with the image of the Virtual Machine. For example, if the CTS provides for jobs migration, the image can be archived or replicated with the help of any external Data Management services.

As it was described in the previous subsection, in order to restart a user's job in the GCA-based environment the *Grid Broker* has to submit a request to execute the *recovery-job-wrapper* to the appropriate *Execution Manager*.

In case of VMM involvement, the *recovery-job-wrapper* is called the vm-ready-job-resumer and it is in charge of resuming the appropriate Virtual Machine. To accomplish that task the *vmm-ready-job-resumer* has to be familiar with the interface of involved VMM and additionally has to be able to fetch the appropriate image (in case the recovering is performed on a different *Computing Resource* than the original job was executed).

5 Conclusions

The checkpointing is an important and desired functionality allowing for seamless functioning of the large installations reducing to a minimum the risk of losing the computation time due to the failures of different origin. The task of introducing the checkpointing functionality is not an easy one and requires a lot of knowledge, on both the Grid infrastructure itself and the checkpointing specific problems. According to experience gained during the work on GCA I in this paper we presented issues along with possible solutions pointing out the advantages and disadvantages of each approach and we hope that this information will help anyone willing to deploy a checkpoint/restart functionality in his Grid environment, which is not an easy task, but the result makes the effort worth it.

One of the biggest problems one has to solve during deployment of presented architecture is limited support from contemporary Grid services. Without basic services such as fully functional storage service or information service it is hard to implement the functionality imposed by the GCA and requires a lot of work during implementation stage, to provide the lacking functionality or to prepare a non-general workarounds. Therefore the advent of more functional basic services would make the deployment work a lot more straightforward.

The presented checkpointing architecture imposes more or less a user driven checkpointing. This means that the user, or the interface that the user is using, have to specify that the job should be run in a more fault-tolerant way. However, that is not a desired solution as the checkpointing should be a feature of the computing environment and all the steps required to achieve the desired quality of service should be done in the background by the Grid services. This disadvantage of the proposed solution is the result of the shortcomings in support for multi-criteria resource management in the contemporary resource brokers and very limited support for the checkpointing. Extending the functionality of the brokers by introducing more flexible job descriptors that could be modified dynamically during the job execution, or implementing support for checkpointing specific scenarios in the brokering software itself is a prerequisite for further development of the checkpointing as a Grid service.

The growing popularity of the *Virtual Machines* technology and its unique capabilities both in the resource man-

agement and fault tolerance areas are making this it a almost ideal checkpointer that is solving one of the biggest problems with the checkpointing: the lack of well maintained products. Solutions like KVM or XEN are either in the mainstream kernel range or are available as a external branches of the Linux kernel tree. This guarantees sufficient widespread in the Grid environment to allow for implementation of services like run-time migration of the applications. The qualities of the VM deployed in the Grid environment make the Grid Checkpointing Service even more functional than before and are outlining the use of this technology as a direction for future research.

References

[1] GFD-I.93 Derek Simmel, Thilo Kielmann, “An Architecture for Grid Checkpoint and Recovery Services”

[2] Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak, Jozsef Kovacs “Towards Grid Checkpoint Architecture”

[3] Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak, Jozsef Kovacs, “Grid Checkpointing Architecture - a revised proposal”

[4] Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak, Jozsef Kovacs, “Improving the fault-tolerance level within the GRID computing environment - integration with the low-level checkpointing packages”

[5] Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak, Maciej Stroinski, Jozsef Kovacs, Atilla Kertesz, “Grid Checkpointing Architecture - Integration of low-level checkpointing capabilites with GRID”

[6] http://www.xensource.com

[7] http://sourceforge.net/projects/kvm

[8] http://www.gridge.org/content/view/18/99/

[9] http://www.cs.wisc.edu/condor/

[10] James S. Plank, Micach Beck, Gerry Kingsley, “Libckpt: Transparent Checkpointing under Linux”

[11] http://checkpointing.psnc.pl/Progress/psncLibCkpt/

[12] http://checkpointing.psnc.pl/Progress/psncCR/

[13] http://directory.fsf.org/project/chpox/

[14] http://en.wikipedia.org/wiki/OpenVZ

[15] http://www.gaussian.com/

[16] http://cryopid.berlios.de/

[17] http://www.globus.org/toolkit/

[18] http://www.vmware.com/