European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
A Network of Excellence funded by the European Commission

Grid Checkpointing Architecture - Integration of low-level checkpointing capabilites with GRID

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

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

CoreGRID Technical Report
Number TR-0075
May 22, 2007

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

Grid Checkpointing Architecture - Integration of low-level checkpointing capabilites with GRID

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

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

CoreGRID TR-0075
May 22, 2007

Abstract

The Grid Checkpointing Architecture (GCA) aims to define the novel, Grid-embedded components and associated design patterns that will allow the Grids to utilize a variety of the existing and future low-level checkpointing mechanisms in a conscious way. The GCA introduces logical specialized components and their place within the Grid environment. The relations between the devised components and those pre-existing ones are defined as well. The emphasis has been put to make the GCA able to be integrated with other components and especially with the upper layer management services like for instance the Grid Broker or Workflow Analyzer. The current shape of the GCA differs from the previous one and seems to be more flexible and more adapted to WSRF-based Grid Services. Additionally, the current GCA proposal is extended with the support for emerging Virtual Machine Monitor technology that allows for suspending and later resuming the whole computing environment (together with the OS). The paper focuses on the general presentation of the proposed architecture and its individual components and on the essential usage scenarios.

1 Introduction

In short GCA is a specification of Grid Services, design patterns and tools that make it possible to integrate the already existing [2] [19] as well as the future checkpointing packages with the Grid-oriented computing environment. The cooperation with VMMes (Virtual Machine Manager) that allows to suspend and resume the state of the whole virtual machine state is also taken into account. An introduction to the low-level checkpointing techniques is presented in [2] and some basis concerning the VMM can be found in [12] [13] [14]. Because the legacy checkpointing packages as such do not conform to the complex Grid environment and greatly differ in functionality, requirements and interfaces, the undertaken task is difficult itself. There are not any standards or specifications concerning the low-level checkpointing techniques. Each checkpointing package can be completely different in its semantics. Additionally, regarding the future checkpointing packages we are not able to anticipate the all possible behaviours and semantics of those packages. The matter with innovative technologies such as VMMes is similar. Right now, we cannot expect

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

a unified interface to VMMes of different vendors. It all implies that GCA should be as flexible and as adaptable as possible. The previous papers [3] [4] describe the earlier versions of the GCA and reveal the sequence of the involved R&D work. While designing the architecture, we assumed that the potential implementation environment will be based on the toolkit compliant with WSRF [5] [6] [7] [8] [9] [10] (for example Globus Toolkit [11]), which is the most up-to-date technique recommended for designing and developing the Grid Services.

The next sections of the paper go into the following subjects. In order to enhance the clarity of the paper, section 2 presents some abstract concepts and definitions that are used throughout the paper. Section 3 introduces a more detailed description of GCA. It presents the individual components of the GCA and their relationships. The idea of utilizing the VMMes as low-level checkpointers is described in section 4. Section 5 is about the essential usage scenarios and demonstrates how the individual components act in a particular situation. The security issues are mentioned in section 6. Section 7 discusses the project which is similar in its interest to the GCA. Section 8 reveals some work that has to be done in the future, and finally section 9 presents some general conclusions.

2 Abstract concepts

Throughout the paper we often use expressions and concepts that can have ambiguous meaning. To avoid misunderstandings and improve the general clarity of the presented concepts the crucial ones used in the paper are explained here. There are not any official definitions but solely explanations of the authors' intentions.

Computing Resource. The *Computing Resource* is any type of hardware that in order to accomplish the user-submitted computation is able to provide CPU cycles and access to I/O devices. As it will appear in section 4, in context of GCA the virtual machines managed by VMMes are not considered as Computing Resources. The virtual machine reside in *Computing Resources* and provide the virtual bubbles that allow secure job separation as well as suspending and resuming the state of the whole "virtual bubble".

Execution Manager. From the point of view of GCA the *Execution Manager* is a pre-existing Grid Service that is in charge of receiving the request of executing jobs and executing them with help of the corresponding *Local Resource Manager* (e.g. LSF, PBS, Torque, SGE). The example of the *Execution Manager* is WS GRAM form Globus Toolkit [11]. We can say that the *Execution Manager* plays the role of a gateway to the *Computing Resource*.

Application, job, process, task. In the HPC, HPT and GRID terminology we can encounter the notions of application, job, process, task or program. All these notions describe a computing activity performed on a given *Computing Resource*. In principle, we can try to point out the differences between them, but in the paper we treat those notions in a very intuitive way and hardly ever distinguish them. From the checkpointing point of view we have to be able to store and restore the state of computing activity performed by the *Computing Resource* and we do not pay attention to the abstract name referring to the activity as such.

Image. The image is persistent data that is stored in non-volatile memory (e.g. on disk) and that contains all information required in order to restore the state and activity of the computing process. We assume that the incremental way of producing the images is allowed.

To take checkpoint. *To take checkpoint* means to trigger somehow (it does not matter how) the activity of collecting and storing all the data required to produce image. The expression *to checkpoint* where the word *checkpoint* is used as a verb has the same meaning.

Checkpoint. *A checkpoint* is considered as a point in time when the checkpoint was taken. Sometimes when we say *checkpoint* we also mean the image that was created when the involved checkpoint was taken.

Checkpointer. The term *checkpointer* refers to any tool, library or another package that allows *taking checkpoints* of programs of a particular type. Later on, when the individual components of the GCA are presented, the *checkpointers* are named as *Core Services*.

Checkpointer class. The notion of a *checkpointer class* is used to collect the *checkpointers* of a similar function-

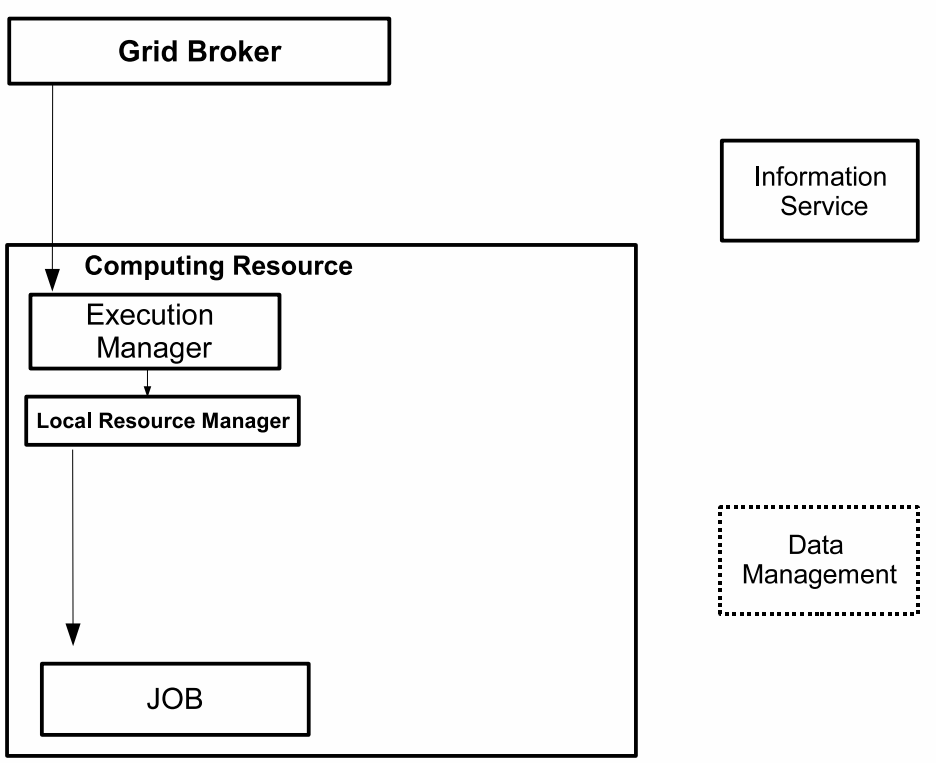

Figure 1: Simplified Grid computing environment.

ality together into a group. The *checkpointing classes* (more precisely - a set of functionality that makes up the class) can be defined in an arbitrary way and each *checkpointer* can belong to any number of *checkpointing classes*. So, the *checkpointing class* is only a logical name used to name all *checkpointers* that adhere to the assumed requirements or, in other words, to the assumed functionality.

Checkpointable application. The *checkpointable application* is every application of which *checkpoint* can be taken. Application can be *checkpointable* because the programmer has embedded such functionality into it, or because some *checkpointers* that can deal with the application are available.

Grid Resource Broker. The *Grid Resource Broker* is a component that is able to coordinate job submissions in a Grid. It is usually invoked with a job description and communicates with the *Information System* of the Grid to find available resources. Relying on this information, it tries to find the best resource for the user job.

Local Resource Manager. It is a component that provides direct access to any *Computing Resource.* In most cases the *Local Resource Manager* is a queue system that schedules the jobs to be executed on a homogenous computing cluster. In the Grid environment the *Execution Manager* exposes a unified interface to different kinds of LRMs.

3 Architecture description

In this section a general outline of the GCA is exposed. 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.

3.1 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 1 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.

3.2 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 on any level (application level, user level, kernel level, hybrid approach). Each *Core Service* is able to checkpoint and restart programs that adhere to the *Core Service* specific constrains. For example, there can be limitation that only programs that do not use the network sockets are supported. The rule of thumb is that each *Core Service* is featured by a different interface and detailed functionality.

Even though the *Core Service* provides the greatly desired checkpointing functionality, in most cases it will be a low-level, legacy package that is not ready to cooperate with the Grid environment. On the one hand, the Grid is not aware that any particular *Core Service* resides on any *Computing Resource*. On the other hand, even if the Grid were aware of the existence of any *Core Service*, the specific interface of the given *Core Service* would not be known to it. It means that unfortunately the Grid is not able to employ the potential checkpointing functionality that is available on a given *Computing Resource*. Therefore, the potentially available checkpointing functionality is wasted in the Grid environment. Such a situation has motivated us to start working on the GCA that is intended to solve this problem.

3.3 Checkpoint Translation Service (CTS)

The partial remedy for the issues highlighted in the previous subsection is the *Checkpoint Translation Service*. It is the 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 not a general CTS but there is a set of CTSes, each of which is assigned to 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 figure 2.

The CTS is a component that among other things is responsible for handling the checkpoint requests. The request can be internal or external. The external requests are triggered by the *Grid Broker* or by any another external component while the inernal 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 sections 3.4 and 3.5. 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 if 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 identyfier, 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 has originally hosted the job. Additionally, all

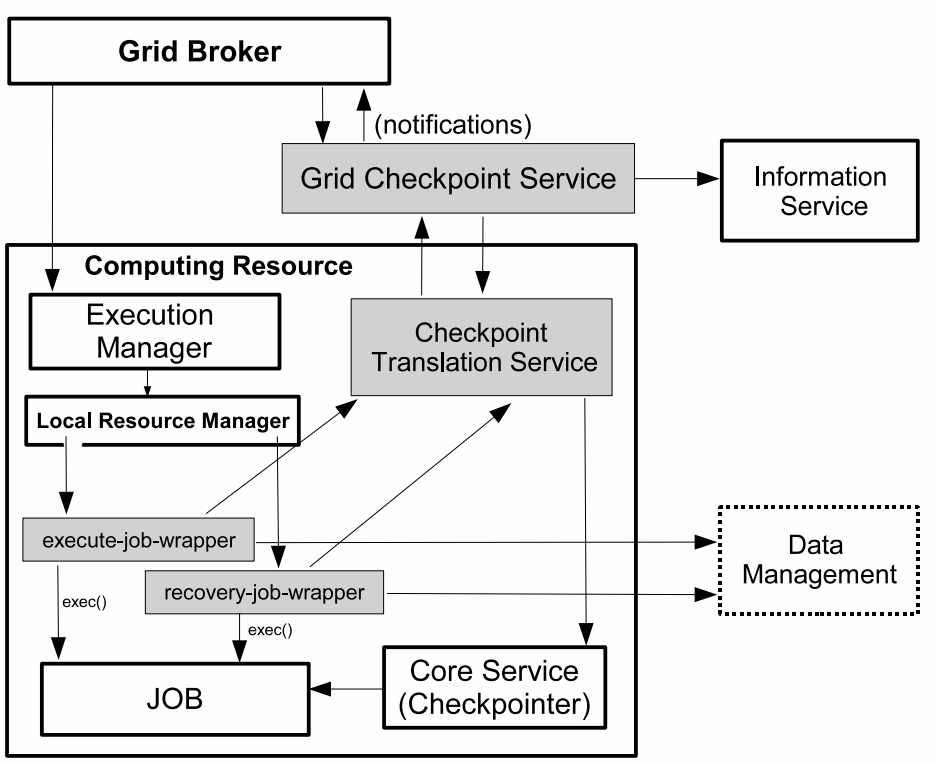

Figure 2: GCA components.

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 3.5) 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*). The published properties concerning the underlying *Core Service* includes the following information:

- Does it support external checkpoints or take the checkpoints by itself?
- Does it allow for recovering the user's jobs on *Computing Resource* different from the one where the checkpoint was taken?
- Is it able to recover the user's job relying on whichever image or only on the recent one?
- The references to the *checkpointing classes* that the *Core Service* belongs to.
- The name of the *Core Service* itself (just to distinguish them).
- The list of functional properties of the *Core Service*. The properties take a form of the key-value pairs.

The published information that concerns the CTS itself includes:

- The name of the CTS.
- The EPR (WSRF-compliant End Point Reference) of the CTS.

- The EPR of the *Execution Manager* that executes the jobs which are to be checkpointed with help of the given CTS.
- The references to the auxiliary wrappers (described in subsection 3.4).
- The information about any additional parameters needed to execute or restore the job (the patterns with placeholders are possible).
- The information about any additional parameters passed to the auxiliary wrappers.

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 subsection 3.4) 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*.

3.4 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*.

3.4.1 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 3.5) 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()*.

3.4.2 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.

3.5 Grid Checkpoint Service (GCS)

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

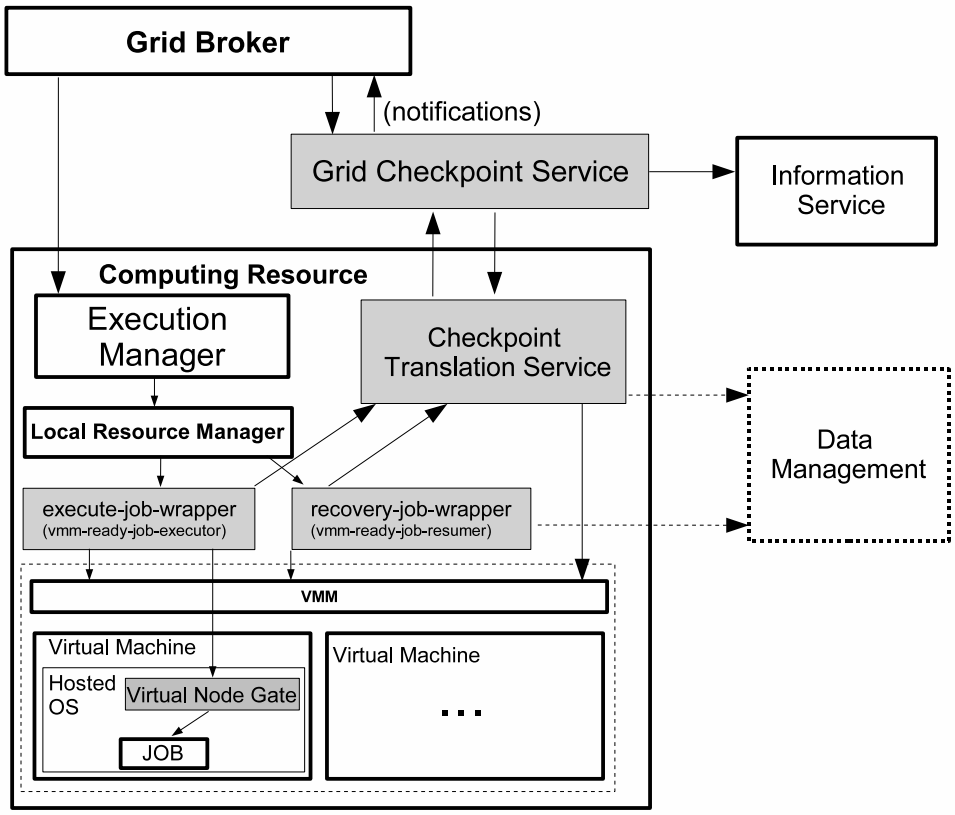

Figure 3: GCA with the VMM as the Core Service.

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 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 The VMM involvement

The main responsibility of VMM is providing and managing the *Virtual Machines*. Each *Virtual Machine* (VM) can be treated as a virtual bubble that can 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 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 Usage scenarios

The current section, basing on the elements described in section 3, introduces basic usage scenarios that can occur in the Grid environment that utilizes GCA. Keep in mind that due to great flexibility of the GCA the scenarios that involve different CTSes and *Core Services* do not have to exactly adhere to the ones presented below. It means that minor adaptations in the actual scenarios with the actual CTSes can occur.

5.1 Submitting checkpointable application

When the end user wants to submit to the *Grid Broker* a job that is to be checkpointable, he or she should indicate the appropriate checkpointer class or CTS or even the name of a particular *Core Service* that is able to deal with the given job. It is up to the end user to know the checkpointer class or CTS or *Core Service* that is able to support the just being submitted job. The scenario where the user indicates the exact set of properties that the adequate *Core Service* should posses and not the name of the *Core Service* or related CTS is also acceptable.

When the *Grid Broker* receives the job submission, from the point of view of GCA the further scenario looks as follows:

1. As usual, the *Grid Broker* looks for *Computing Resources* (i.e. for EPRes of appropriate *Execution Managers*) that match the job's requirements. The resource that provides access to the *Core Service* is the CTS. Therefore during the resource matching process, the *Grid Broker* has to look for the *Computing Resources* with CTS that is the "driver" to the *Core Service* of a given name or checkpointer class. The *Grid Broker* can obtain the appropriate information about the CTSes from the *Information Service*. Depending on the information that the user has provided, the CTS that will deal with the given job can be determined basing on the CTS name itself, the checkpointer class or the name of the *Core Service*. In the case when only the set of properties of the potential *Core Service* has been provided, the appropriate CTS is looked for basing on those properties.

2. When the *Grid Broker* chooses an appropriate *Computing Resource* (accurately when the EPR of a given *Execution Manager* is known) and appropriate CTS, then basing on meta-data exported by given CTS to the GCS and further by the GCS to the *Information Service*, the *Grid Broker* finds out what is the name of *execution-job-wrapper* and what are optional additional parameters that have to be passed to the job and to the wrapper. Then the *Grid Broker* submits the request of executing the correct *execution-job-wrapper* to the *Execution Manager* connected with the chosen *Computing Resource*. The parameters passed to the wrapper include the GID assigned by the *Grid Broker* to the user job, the job itself, the user-defined parameters that will finally be passed to the job and mentioned before additional parameters obtained from the *Information Service*.

3. The *Execution Manager* executes the submitted *execute-job-wrapper* passing all the involved parameters to it.

4. The *execute-job-wrapper* registers at CTS the relation between the GID (passed to the wrapper by the *Grid Broker* as one of the parameters) and the local ID that is further used by the CTS to trigger the checkpoint of the given job. It depends on a particular *Core Service* what type of local ID is used, but the relation can be registered regardless of the form of the local ID. Then the CTS informs the GCS that from now on the CTS is in charge of forwarding checkpoint requests to the job of the given GID.

5. It is possible that some of the additional parameters take the form of expressions with placeholders and before they are passed to the actual job they have to be evolved into actual values. In such a case the help of CTS can be required. In the presented scenario we do not see the need for performing this step but it is described here to demonstrate the flexibility of the architecture and that if it is needed the individual components can be fitted to the current circumstances.

6. The *execute-job-wrapper* uses the *exec()* syscall to execute the actual user's job. All the parameters defined by the user and optional additional parameters introduced by a particular CTS are passed to the job.

7. From now on the user job is running.

8. If the *Core Service* allows triggering the checkpoints by the external requests, the CTS listens for checkpoint requests and forwards them to the underlying *Core Service*. In a case when a job takes checkpoints of itself, the CTS, if it is possible, detects (or is informed) that the checkpoint was taken, informs the GCS about it and it can further forward the information to the *Grid Broker* and to the *Information Service*. The main reason why CTS should be aware of checkpoints is for the bookkeeping purposes. Depending on the implementation and the given *Core Service*, after a checkpoint is taken, the CTS can perform additional actions like encrypting or replicating or archiving the image.

5.2 Issuing and handling checkpoint requests

This subsection describes the scenario where the user's job has already been submitted and started on the appropriate Computing Resource. The assumption is that the underlying Core Service supports the external checkpoint requests. The scenario of issuing the checkpoint request looks in the following way:

1. The *Grid Broker* decides to take checkpoint of the user's job of the given GID (it does not matter why and how the broker decided about it).

2. The Grid Broker sends the doCheckpoint(GID) request to the GCS.

3. The GCS, basing on data that were registered at it by CTS when the job of the given GID was started, finds out the EPR (*End Point Reference*) of CTS that is in charge of forwarding the checkpoint request to the appropriate *Core Service*.

4. The *Grid Broker* passes the *doCheckpoint(GID)* request to the adequate CTS. Depending on the semantics of a particular underlying *Core Service*, the CTS can block or postpone all further requests until the handling of the current request finishes.

5. The CTS basing on its own embedded knowledge and data that has been registered by the *execute-job-wrapper*, utilizes the underlying *Core Service* to take the checkpoint of the user's job. The connection between GID and local ID that is used to determine the user's job is obtained thanks to the data that was registered at CTS by the *execute-job-wrapper*.

6. The just taken checkpoint should be assigned a unique identifier. The CTS has to remember the connection between the checkpoint identifier and the just created image. It is not defined what such a connection should look like. It is only important that in the future CTS should be able to locate and utilize the given image basing on the checkpoint identifier (unfortunately sometimes the Core Grid semantics can support restoring jobs basing only on the recent image).

7. Right now the CTS can perform a lot of customized actions. For example, the image can be encrypted, replicated or archived.

8. The CTS has to register with GCS the checkpoint identifier and connect it with the following information: the GID of the job whose checkpoint was just taken, time when the checkpoint was taken, the name of the CTS that was involved into taking the checkpoint, the sequential number of the checkpoint and EPR of the *Execution Manager* that was used to execute the just checkpointed job.

9. The blocking or postponing of the incoming checkpoint request can be disabled.

10. The checkpoint identifier is returned to the GCS and GCS further returns the checkpoint ID to the *Grid Broker*.

5.3 Recovering the job

We do not care about the way the *Grid Broker* decides that the user's job has to be restarted (the jobs monitoring is out of scope of the GCA). However, when the *Grid Broker* decides that the job of the given GID has to be recovered, the following steps describe the example scenario:

1. The *Grid Broker* decides that the job of the given GID has to be recovered.

2. The *Grid Broker* basing on its own history knowledge or on data available from the *Information Service* finds out what CTS was recently associated with the job of the given GID.

3. The *Grid Broker* tries to find the *Computing Resource* on which the job will be recovered. The choice is limited to the sites that have installed the CTS of the same type as the one that was chosen when the job was executed for the first time. In some cases the choice can be even limited to exactly the same *Computing Resource* that was employed when the job was executed for the first time.

4. If the given CTS allows it, the *Grid Broker* can choose the image identifier that will be used to recover the job. The list of all images (more accurately of all image identifiers) associated with the given GID and all their properties are available from the *Information Service*. It is also possible to rely on a relative identifier denoting the recent available image.

5. As at the moment the involved CTS is known, the *Grid Broker* gains the name of the associated *recovery-job-wrapper* from the *Information Service*.

6. The *Grid Broker* submits the request of executing the *recovery-job-wrapper* (the one established in the previous step) to the chosen *Execution Manager*. The optional additional parameters and image identifier are submitted as well.

7. When the *Execution Manager* executes the *recovery-job-wrapper* it is passed the GID of a job being recovered and identifier of the image that has to be used.

8. The *recovery-job-wrapper* basing on its own embedded knowledge, on the passed GID and image identifier obtains the appropriate image (if it is required).

9. The *recovery-job-wrapper* recovers the job in a way specific for the *Core Service* and basing on the just obtained image.

10. The *recovery-job-wrapper* also has to register at CTS a new relation between GID and the local ID used to identify the job when checkpoint is taken. It depends on a particular *Core Service* how and exactly when the mentioned local ID is obtained. Here we assume that the underlying *Core Service* provides a special recovery command that creates a new process that finally becomes the recovered job and that the local ID that we need is the PID of the recovered process. In such a case the registration that connects the GID and PID is performed after the recovery command finishes its work.

11. The CTS, when receives the information about the relation between GID and local id from the *recovery-job-wrapper*, additionally registers the relation between GID and itself at GCS. From now on the GCS is aware which CTS handles the checkpoint requests addressed to the job of a given GID.

12. Actually the *Execution Manager* is not aware that it has just executed the *recovery-job-wrapper* (instead of executing the user's job). As it thinks that the user's job has been executed, it may wish to interact with it by means of Unix signals. Therefore, in the described case, the *recovery-job-wrapper* has to stay in memory as a demon that receives all signals and further forwards them to the recovered job. However, if the *Core Service* allowed exchanging the memory space of the *recovery-job-wrapper* with the memory space of the job being recovered, the *Execution Manager* would interact directly with the recovered job.

5.4 Registering new checkpointing class

The checkpointing classes serve to collect the *Core Services* of a similar functionality in groups. When the end user submits the job that is to be checkpointable, he or she has to tell the *Grid Broker* what checkpointing class is capable of dealing with the given job. To define a new checkpointing class the related functionality has to be indicated. It is done in a form of a sequence of freely defined properties. The association of the given CTS with the given checkpointing class takes place during CTS registration. The GCS exposes a list of registered CTSes, checkpointing classes, their properties and relations in a form of WS-ResourceProperties. There is an assumption about one simple preexisting checkpointing class: the basic class. The basic class provides support for checkpointing jobs of the following properties: one process, one thread, input and output data are placed in regular files, network and other inter process communication techniques are not used.

5.5 Deploying new CTS

To integrate a new *Core Service* with the GCA someone has to implement the dedicated CTS. Before the GCA is able to employ a new *Core Service*, first the dedicated CTS has to be deployed. There are two stages of deploying a new CTS.

The first stage of deploying the CTS encompasses informing the GCS about the functionality of the new CTS and its main properties (like CTS name, associated checkpointing classes, names of *execute-* and *recovery-job-wrapper*, and so on). To accomplish this task the CTS provider has to prepare a special CTS descriptor (an xml-based text file) that is further used to register all required information at GCS.

The second stage of deploying CTS encompasses configuration, installation and execution of the CTS on a particular *Computing Resource*. An administrator who is installing the CTS has to edit a configuration file that includes such information as: EPR of GCS, EPR of an associated *Execution Manager* as well as the name and EPR of CTS itself. When the CTS is started up, it reads the configuration file and registers itself (together with accompanying properties) at GCS defined in the same file. According to the model described in the Security Issues section the CTS has to posses a digital certificate that is accepted by GCS. From now on, the Grid Broker has access to CTS through the GCS.

5.6 Deploying the GCS

According to the model described in the Security Issues section, when GCS is deployed it has to be configured to accept the *Grid Broker's* request and the *Grid Broker* has to be configured to accept potential GCS's notifications. When the GCS is started up it should register itself at the *Information Service*. Thanks to that the *Grid Broker* will be able to find the EPR of the GCS.

6 Security Issues

There is an assumption that secure communication between individual components of the GCA is ensured by the *Grid Security Infrastructure* (GSI). A *Grid Broker* has to be configured to accept the messages from GCS (GCS can send notifications to the *Grid Broker*). The GCS must accept messages from the *Grid Broker* and from all CTSes and the CTSes have to accept messages from GCS. The model of mutual acceptance can be based on GSI-derived digital certificates.

As regards the security of images, the CTS may provide additional mechanismes and solutions to ensure adequate level of security itself. For example, it is up to the given CTS if images are transferred over the network and if they are encrypted.

7 Related work

Comparing with the GCA, the most similar project is the GridCPR which "defines a user-level API and associated layer of services that will permit checkpointed jobs to be recovered and continued on the same or on remote Grid resources". More accurately the GridCPR works on a specification that future applications will have to adhere to in order to make them checkpointable in a Grid environment and in order to make the Grid capable of taking advantage of this checkpointing technique. The GridCPR project is managed by the Open Grid Forum Grid Checkpoint Recovery Working Group. Information about the project and its progress can be found on the pages: http://gridcpr.psc.edu/GGF/ and https://forge.gridforum.org/sf/projects/gridcpr-wg. The main difference between GCA and GridCPR is that GCA focuses mainly on legacy checkpointing packages and on those that are not Grid-aware as such. The GCA tries to ansewer the question how to incorporate the already existing checkpointing packages into the GRID whilst the GridCPR aims to define new checkpointing related services and API from scratch. Of course both approaches are rational and have application in different situations depending on developers and users requirements.

8 Future work

The presented shape of GCA is the result of feedback collected after publishing the early GCA version. In the current state it is more flexible and adaptable to cooperate with the *Grid Broker* or any another manager. However, further consultations and discussions are anticipated, and any more enhancements that will improve the overall level of scalability, flexibility and integratebility are possible.

9 Conclusions

Basing on feedback from the previous proposition of the GCA and with the aid of our new partners we finally defined a new shape of GCA. The main change focuses on the range of responsibility of individual components and on the way the individual components interact with each other. The architecture is to more extent driven by the WSRF specification. Providing that the surrounding environment is also based on WSRF compliant Grid Services, the new approach ensures more modularity and independence as the previous one. Additionally the possibility of embedding the VMM-based environments has been mentioned explicitly.

To summarize, the GCA consists of a few components which when embedded into the GRID environment allow employing low-level checkpointing packages in this environment. The individual GCA components, the internal and external relationships and related design rules have been presented in the paper. Nevertheless, just in a few words the GCA consists of set of Checkpoint Translation Services and special recovery- and execute-job-wrappers that are dedicated for given Core Service (i.e. low-level checkpointer). Simplifying things we can say that Checkpoint

Translation Service provides interface to the Core Service while the mentioned wrappers implement this interface. Finally, the component that mediates between the set of CTSes and the other GRID components (including the Grid Resource Broker) is Grid Checkpoint Service.

Because GCA is intended to cooperate with as many existing and future *Core Services* as possible, the components like CTS and *execute-* and *recovery-job-wrapper* are defined in a very general way. In most cases, the GCA defines the responsibility of those components rather than the exact actions or algorithms that they should do. It makes it difficult to describe the architecture but at the same time provide in more extent desired level of flexibility. Trying to grasp the spirit of the GCA, it is important to understand that a lot of responsibility is left up to the CTSes and related *execute-* and *recovery-job-wrappers*. For example, it is up to the CTS if the images are replicated, archived or encrypted. It is also up to CTS (and related wrappers) if and how to obtain the image when the job is recovered on the *Computing Resource* other than the one where checkpoint was taken. When the *recovery-job-wrapper* is told to recover job of the given GID, the wrapper has to perform all the actions required to obtain the proper image (with help of CTS and optionally any other external services) and employ the *Core Service* to perform the actual job recovering. It should also be made clear that every CTS and related auxiliary tools (such as execute- and recovery-job-wrapper) are implemented separately for each Core Service. Nevertheless, it is possible that multiple implementations of CTSes provide access to the same *Core Service*. The conditions that have to be met when the next CTS is developed are: the imposed interface that CTS exposes to the GCS and the format of the involved WS-ResourceProperties. Even though the actual way of implementation of CTS is not defined, the main features of the given CTS should be known to the *Grid Broker*. The features are enumerated in the CTS descriptor and exported in a form of WS-ResourceProperties. To adapt a Core Service to the GCA one must provide the specialized CTS and associated wrappers while the GCS is general and can deal with a variety of CTSes, *Core Services* and *Grid Brokers*. Therefore, the integration of new *Core Services* requires some development work to be done and the work has to be based on rules and principles presented in the paper.

Concluding the considerations about GCA, it is worth spottimg the advantages of the architecture for the Grid environment. The emerging of productive implementation of GCA can significantly increase the range in which the checkpointing functionality is utilized in Grid-based computing environments. Furthermore, we assume that the advantages of checkpointing functionality and related possibility of migrating jobs and improving the level of fault-tolerance are obvious and we do not describe them here.

References

[1] http://www.coregrid.net/

[2] A Survey of Checkpointing/Restart Implementations, Eric Roman, Lawrence Berkley National Laboratory, CA.

[3] Gracjan Jankowski, Jozsef Kovacs, Norbert Meyer, Radoslaw Januszewski and Rafal Mikolajczak, Towards Checkpointing Grid Architecture, PPAM2005 proceedings.

[4] Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikoajczak, Jozsef Kovacs, CoreGRID Technical Report, Number TR-0036 May 30, 2006.

[5] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf

[6] Web Services Resource Framework (WSRF) - Primer v1.2, Committee Draft 02 - 23 May 2006, http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf.

[7] Web Services Resource Properties 1.2 (WS-ResourceProperties), OASIS Standard, 1 April 2006, http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf.

[8] Web Services Resource Lifetime 1.2 (WS-ResourceLifetime) OASIS Standard, 1 April 2006, http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-os.pdf.

[9] Web Services Service Group 1.2 (WS-ServiceGroup), OASIS Standard, 1 April 2006, http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf.

[10] Web Services Base Faults 1.2 (WS-BaseFaults), OASIS Standard, April 1 2006, http://docs.oasis-open.org/wsrf/wsrf-ws_base_faults-1.2-spec-os.pdf.

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

[12] Enhanced Virtualization on Intel Architecture based Server, White Paper.

[13] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Mrtins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, Larry Smith, Intel Virtualization Technology, Computer, May 2005, pp. 48-56.

[14] P. Baham et al., ”Xen and the Art of Virtualization”, Proc. 19th ACM Symp. Operating Systems Principles, ACM Press, 2003, pp. 164-177.

[15] T. Garfinkel et al., ”Terra: A Virtual Machine-Based Platform for Trusted Computing”, Proc. 19th ACM Symp. Operating Systems Principles, ACM Press, 2003, pp. 193-206.

[16] http://gridcpr.psc.edu/GGF/

[17] http://www.gridforum.org/

[18] Jos’e Carlos Sancho, Fabrizio Petrini et al., Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance.