CPPC-G: Fault-Tolerant Parallel Applications on the Grid

Daniel Díaz, Xoán C. Pardo, María J. Martín, Patricia González,
Gabriel Rodríguez, Juan Touriño, and Ramón Doallo

Computer Architecture Group,
Department of Electronics and Systems,
University of A Coruña, SPAIN
{pardo,mariam,pglez}@udc.es

Abstract. With the maturity of the Grid, the community has made an important effort in developing middleware and tools to provide different functionalities, such as resource discovery, resource management, job submission, execution monitoring. Unfortunately, not so much attention has been paid in developing mechanism to provide fault-tolerance to execution of applications on a Grid. This paper addresses the design and implementation of an architecture based on services (CPPC-G) to manage the execution of fault tolerant parallel applications on Grids. The CPPC (Controller/Precompiler for Portable Checkpointing) framework is used to insert checkpoint instrumentation into the application code. Designed services will be in charge of submission and monitoring of the execution of the parallel application, management of checkpoint files and detection and automatic restart of failed executions.

1 Introduction

Parallel computing evolution towards cluster and Grid infrastructures has created new fault tolerance needs. As parallel machines increase their number of processors, so does the failure rate of the global system. This is not a problem while the mean time to complete an application execution remains well under the mean time to failure (MTTF) of the underlying hardware, but that is not always true on long running applications, where users and programmers need a way to ensure that not all computation done is lost on machine failures. Checkpointing has become a widely used technique to obtain fault tolerance on such environments. It periodically saves the computation state to stable storage, so that the application execution can be resumed by restoring such state. A number of solutions and techniques have been proposed [2], each having its own pros and cons.

Grid computing presents new constraints for checkpointing techniques. Its inherently heterogeneous nature makes it impossible to apply traditional state saving techniques, which use non portable strategies for recovering structures such as application stack, heap, or communication state. Therefore, modern checkpointing techniques need to provide strategies for *portable* state recovery, where

the computation can be resumed on a wide range of machines, from binary incompatible architectures to incompatible versions of software facilities, such as different implementations for communication interfaces.

CPPC (Controller/Precompiler for Portable Checkpointing) [7, 8] is a checkpointing tool focused on the insertion of fault tolerance into long-running message-passing applications. It is designed to allow for execution restart on different architectures and/or operating systems, also supporting checkpointing over heterogeneous systems, such as the Grid. It uses portable code and protocols, and generates portable checkpoint files while avoiding traditional solutions which add an unscalable overhead, such as process coordination or message-logging.

This paper introduces CPPC-G, a set of new Grid services[1], implemented on top of Globus 4 [3] using the Java API, able to manage the execution of CPPC-instrumented fault tolerant parallel applications (CPPC applications from now on). The designed set of services will be in charge of submitting and monitoring the parallel execution, as well as of managing and replicating the state files generated during the execution. It is widely accepted that the performance of an MPI application on the Grid remains a problem, caused by the communication bottleneck on wide area links. To overcome such performance problem, in this work it is assumed that all processes of a CPPC application are executed on the same computing resource (e.g. a cluster or an MPP machine with MPI installed). Upon a failure, the Grid services described in this work restart the parallel application from the last consistent state, in a completely transparent way. At the present stage of CPPC-G development, only failures in the computing resource where the CPPC application is executed are being considered.

The structure of the paper is as follows. Section 2 gives a overview of the CPPC framework. Section 3 describes CPPC-G, its architecture, implementation details and deployment. The operation of the CPPC-G tool is shown in Section 4. Section 5 concludes the paper.

2 The CPPC Framework

CPPC[2] appears to the user as a runtime library containing checkpoint-supporting routines, together with a compiler tool which automates the use of the library for message-passing codes. There are several issues to be solved in implementing checkpointing solutions for parallel applications, such as consistency, portability, memory requirements, or transparency.

Global consistency In CPPC, consistency of checkpoint files among different processes is achieved by means of spatial coordination, rather than temporal coordination. Checkpoints are taken at the same relative code points by all

[1] With the term Grid service we denote a Web service that complies with the Web Services Resource Framework (WSRF) specifications

[2] CPPC is an open source project, and its current release can be downloaded at `http://cppc.des.udc.es`.

the processes (assuming SPMD codes). To avoid problems caused by messages between processes, checkpoint directives must be inserted at points where it is guaranteed that there are no in transit [1], nor ghost [5] messages. These points are called *safe points*, and checkpoint files generated in such points are *strongly consistent* [5]. For an automatic identification of safe points, a static analysis of interprocess message flow is needed. This automatization is currently under development.

Portability A checkpoint file is said to be portable if it can be used to restart the execution on an architecture or OS different from those where the file was generated on. This means that checkpoint files should not contain hard machine-dependent state, which should be recovered at restart time using special protocols. CPPC recovers non-portable state by means of the re-execution of the code responsible for creating such opaque state in the original execution. Hence, the new code will be just as portable as the original code was. Moreover, in CPPC the effective data writing will be performed by the selected writing plugin, using its own format. This enables the restart on different architectures, as long as a portable dumping format is used for code variables. Currently, a writing plugin based on HDF5 [6] is provided.

Memory requirements Checkpointing of large-scale real applications may lead to a great amount of stored state, the cost being so high as to become impractical. CPPC reduces the amount of data to be saved by including, in its compiler, a live variable analysis in order to identify those variable values that are needed upon restart. Besides, the HDF5 library can accommodate data in a variety of ways, including a compressed format based on the ZLib library [4]. A multithreaded dumping option is also provided to improve performance when working with large datasets.

Transparency As said before, the CPPC tool appears to the user as a compiler tool and a runtime library. The compiler tool aims to automate the use of the library. The user must insert only one compiler directive into the original application (the `cppc checkpoint` pragma) to mark points in the code where the relevant state will be dumped to stable storage into a checkpoint file. The compiler will perform a source-to-source transformation, automatically identifying both the variables to be dumped to the checkpoint file and the non-portable code to be re-executed upon restart; and inserting the necessary CPPC functions, as well as flow control code.

3 CPPC-G Design

In this work the design and implementation of a set of Grid services for remote execution of CPPC applications is described. The new Grid services must provide different functionalities such as resource discovery, remote execution and

monitoring of applications, detection and restarting of failed executions, etc. This section discusses the most relevant design and implementation issues to achieve all these features.

3.1 Some preliminary questions

Before going into details of the proposed CPPC-G architecture, some general questions, that are common to all its services, must be introduced. They are related to security issues, management of long operations, chained invocations among services, resource creation requests and provided clients.

Security issues The developed services depend on the underlying GSI security infrastructure of Globus for authentication, authorization, and credential delegation. Credential delegation can be performed directly by the user, or automatically by a service that itself has access to a delegated credential. The standard Globus services follow the principle of always making the user delegate the credentials himself beforehand, never resorting to automatic delegation. This is more cumbersome for the user, but allows a greater control over where the credentials will end up. The developed CPPC-G services also try to follow that same principle whenever possible. However an exception is made in situations in which the service to be invoked in not known beforehand because it is dynamically selected. Automatic delegation has been used in these cases.

Managing long operations Using a simple invocation of a Grid service to implement an operation that takes a long time to complete lacks flexibility, since there is no way to cancel or interrupt it. Instead, a approach based on the factory pattern is preferred. In this approach, widely used in the implementation of Globus, an operation is started by invoking a factory service that creates an instance resource in an instance service using a given factory resource as template. In the following, the terms resource and service will be used to refer to resource and service instances. The newly created resource is responsible for tracking the progress of the operation and it will be possible to query its state or subscribe in order to receive notifications when it changes. Furthermore, resources created in this way can be explicitly destroyed at any time or be automatically destroyed when a termination time specified by the user expires. It is responsibility of the user to extend the resource lifetime if it was not long enough to complete the operation. Resource lifetimes are a means to ensure that resources will be eventually released if communication with the user is lost.

Chained service invocations Most operations are implemented by invoking a service that itself invokes other services. It is usual to end up with several levels of chained invocations. With more than two levels, some difficulties arise with the delegation of credentials. In Globus, services that require user credentials publish the delegation factory service EPR (endpoint reference) as a resource

property of their corresponding factory service. The user must use that EPR to delegate his credentials before being allowed to create resources in the service. Services that invoke other services that also require the delegation of credentials must publish one delegation factory service EPR for each invoked service. In the following, the term delegation set will be used to refer to the set of delegation factory service EPRs where the user must delegate his credentials before using a service. Once the user has delegated his credentials to the proper delegation services, delegated credential EPRs are passed to invoked services as part of resource creation requests, that will be explained later in this section. In the following, the term credential set will be used to refer to the set of delegated credential EPRs.

When there are a large number of services involved in a chained invocation, the use of delegation sets becomes complicated for users and administrators. From the user's point of view, the delegation set is a confusing array of delegation EPRs to which he must delegate his credentials before invoking a service. To help users, XML entities have been defined to be used in the service WSDL file to describe the delegation sets in a hierarchical fashion. Once the WSDL is processed and stubs are generated, helper classes can be defined to handle delegation automatically. From the administrator's point of view, all the EPRs in the delegation set of a service must be specified in its configuration files, which is an error-prone task. To avoid this problem, a technique based on queries to build delegation sets dynamically has been implemented.

Resource creation requests In order to create a resource, factory services take as parameters the following creation request datatypes.

- *The initial termination time of the resource.*
- *The resource description.* This is the main component. It may include additional resource descriptions associated with chained invocations (e.g. WS-GRAM job descriptions include RFT transfer descriptions for file staging).
- *A credential set.* That is, the delegated credential EPR to be used by the resource, plus the delegated credential EPRs to be used in invocations to other services. This is separated from the resource description to potentially allow different requests to reuse the same credentials (e.g. repeated invocations to RFT with different source/destination URL pairs but with the same user credentials).
- *A refresh specification.* For each of the services the resource is expected to invoke a lifetime refresh period and a ping period are specified. Being the lifetime refresh period the amount of seconds to extend the lifetime of resources in invoked services; and the ping period the frequency with which the resources in invoked services are checked to be up and reachable. If no service invocations are expected, no refresh specification is needed.

Provided clients For each CPPC-G service two client programs are provided: a command-line client and a resource inspector. Command-line clients are used

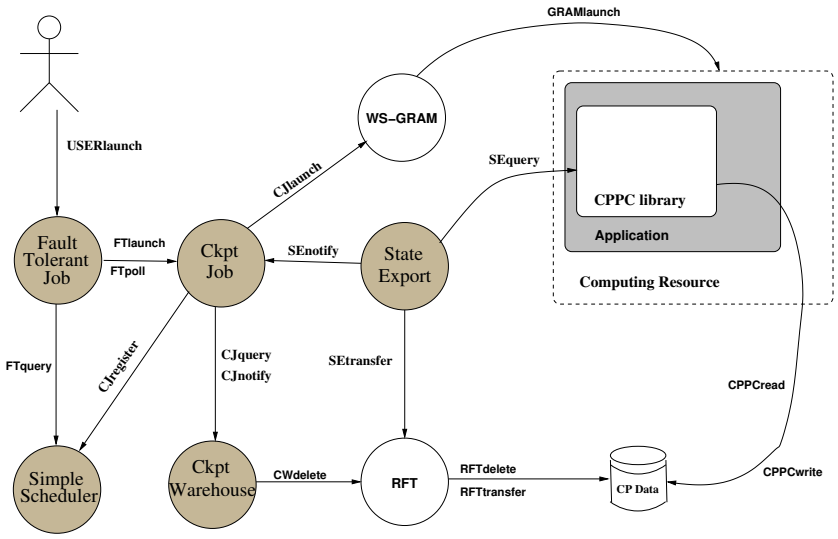

Fig. 1. System Architecture

for the creation of resources from their descriptions. They prepare creation requests, contact the proper factory services and return the EPRs of the created resources. Resource inspectors are used for interacting with the created resources. They have a graphical interface to monitor resource notifications, query resource properties and invoke service operations.

3.2 System Architecture

Figure 1 shows the proposed CPPC-G architecture that comprises a set of five new services that interact with Globus RFT and WS-GRAM services. A `FaultTolerantJob` service is invoked to start the fault-tolerant execution of a CPPC application. Available computing resources hosting a `CkptJob` service are obtained from a `SimpleScheduler` service. `CkptJob` services provide remote execution functionalities. Executing CPPC applications periodically stores checkpoint files. `StateExport` services are responsible for tracking local checkpoint files stored periodically by CPPC applications. And finally, `CkptWarehouse` services maintain metadata and consistency information about stored checkpoint files. Distributing the functionality into a number of separate services contributes both to modularity and reusability. In the following, the functionality of each service is described in depth.

CPPC applications running in a computing resource can store checkpoint files in locations not accessible to services hosted in different computing resources (e.g. the local filesystem of a cluster node). The `StateExport` service is responsible for tracking these local checkpoint files and move them to a remote site that could be accessed by other services. There must be a `StateExport` service for every computing resource where a CPPC application could be executed. One

`StateExport` resource is created for each process of a running CPPC application. This resource is responsible for finding checkpoint files the process generates. To help finding checkpoint files, the CPPC library has been slightly modified. Now processes of CPPC applications write, besides checkpoint files, metadata files in a previously agreed location in the computing resource filesystem. Each of these files contains the URL of the newly generated checkpoint file, the checkpoint file identifier number (*epoch*[3] from now on) and the name that will be used for the next metadata file. The name of the first metadata file is passed to the CPPC application as a parameter. `StateExport` resources periodically check (by using GridFTP) for the existence of the next metadata file in the series. When a new one is found, the resource parses it, locates the newest checkpoint file using the extracted information, and replicates it via RFT in a previously agreed backup site. The original checkpoint filename is not preserved and the copy is given a new Universally Unique Identifier (UUID). When the replication is finished, the original checkpoint file is deleted (also via RFT), and a notification is sent to the `CkptJob` service.

The `CkptJob` service provides remote execution of CPPC applications. This service coordinates with `StateExport` and `CkptWarehouse` to extend WS-GRAM adding needed checkpointing functionality. Job descriptions used by the `CkptJob` service are those of WS-GRAM augmented with the EPR of a `CkptWarehouse` service and a `StateExport` resource description element, that will be used as a template to create `StateExport` resources. As in WS-GRAM, the `CkptJob` factory service contains several factory resources, each of them tied to the corresponding WS-GRAM factory resource[4]. Users must specify which one to use when creating a `CkptJob` resource. The `CkptJob` service can be configured to register useful information with an MDS index service. Currently the information registered for each factory resource is composed by its EPR and a sequence of tags. These tags are specified by the administrator in a configuration file and used to indicate that the factory resource has a certain property (e.g. that MPI is installed in the computing resource).

The `CkptWarehouse` service maintains metadata about checkpoint files generated by running CPPC applications, namely: the URL of the checkpoint file, the identifier of the process that generated it, and its epoch. Each time a checkpoint file is successfully exported the proper resource in the `CkptWarehouse` service is notified, being responsible for composing sets of globally consistent checkpoint files (this functionality was originally provided by the CPPC library). When a new globally consistent state is composed, checkpoint files belonging to the previous state (if they exist) become obsolete and can be discarded (they are deleted by using RFT).

[3] In CPPC each checkpoint file is uniquely identified by a number. The same identifier on different processes refers to a checkpoint file created at the same relative execution point, thus allowing processes to coherently argue about valid restart points.

[4] Factory resources in WS-GRAM are used to transparently execute jobs using any local scheduler (i.e. fork, PBS, SGE, etc.).

The SimpleScheduler service keeps track of available computing resources hosting a CkptJob service. The service subscribes to an MDS GT4 index service that aggregates the information published by registered CkptJob services. In particular, the sequences of tags published by CkptJob services are used to select the proper computing resource that satisfies some given scheduling needs. As for now, the only supported scheduling need is the required presence of a given tag, but this mechanism could be used in future versions to support more complicated selections.

The FaultTolerantJob service is the one that the user invokes to start the execution of a CPPC application. One resource is created for each application, being responsible for starting and monitoring it by periodically checking its execution state. In case of failure, the execution is restarted automatically. Computing resources needed for executing the application are obtained by querying a SimpleScheduler service, so it is not possible to know beforehand where the application will be executed. As a consequence, credential delegation has to be deferred until it is known the precise CkptJob service to be invoked to execute the application. Descriptions of FaultTolerantJob resources are composed of a scheduling need and the description of a CkptJob resource. Failed executions are indicated by FaultTolerantJob resources by setting a resource property and sending a notification.

3.3 Component Deployment

Figure 2 shows the expected deployment of CPPC-G services in a Grid. As it was already mentioned, it is assumed that all processes of a CPPC application will be executed in the same computing resource for performance reasons. In a typical configuration of CPPC-G, a CkptJob service and a StateExport service will be present in that resource (as will be Globus WS-GRAM and RFT services). The CkptWarehouse service will be hosted in the same storage resource to which checkpoint files are moved, and the SimpleScheduler and FaultTolerantJob services will reside in other computing resources. It must be noted that the use of SimpleScheduler and FaultTolerantJob is optional. They must be present only if automatic restart of failed executions is wanted. The rest of services can be used on their own if only remote execution of CPPC applications is necessary. In this case it will be responsibility of the user to manually restart a failed execution.

Other configurations besides the one showed in the figure are possible. Although it is usual for a CkptJob service to invoke the StateExport service hosted in the same Globus container, it is not mandatory. The StateExport and CkptJob services could reside in different computing resources provided that GridFTP servers are present in both of them. In any case, the StateExport service must be configured to inspect the same computing resource where the CkptJob service submits the CPPC application, otherwise no checkpoint files will ever be found. It is usual for a CkptJob service to invoke WS-GRAM in the same Globus container,but it is also possible to host them in different computing resources. In a similar way, it is not mandatory for the CkptWarehouse service

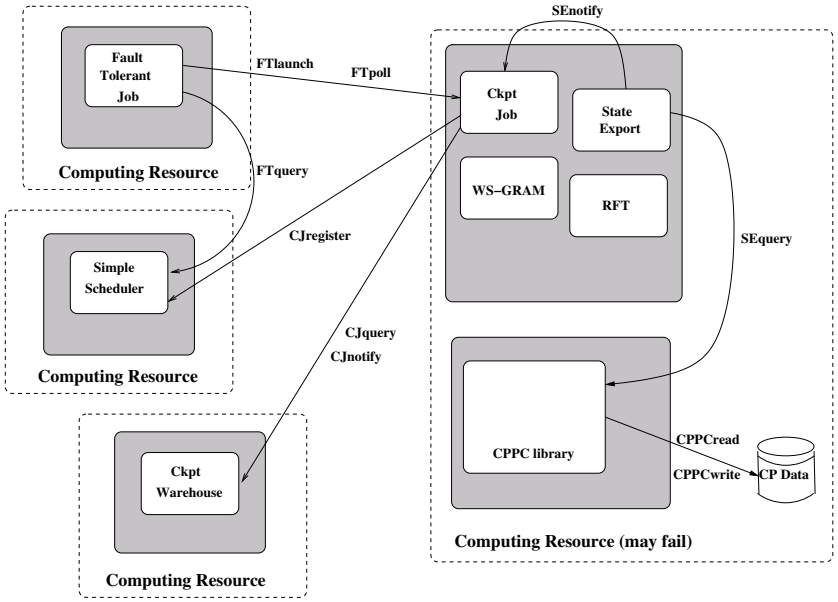

Fig. 2. CPPC-G deployment

to reside in the same resource in which checkpoint files are stored. They can be remotely accessed in any resource containing RFT or GridFTP (or both). All these alternatives provide more flexibility to administrators when deploying the system.

4 CPPC-G Operation

In this section some of the activities involved in executing fault-tolerant CPPC applications with the set of services proposed are described. It is assumed that, before the user submits an application, all available `CkptJob` services are already registered with a `SimpleScheduler` service (`CJregister` in Figure s1 and 2).

To initiate the execution of an application the following steps are taken in sequence:

1. In order to prepare the application submission, the user must create in advance an instance of the `CkptWarehouse` service and a credential set. Typically the `CkptWarehouse` instance will reside in the storage resource the user wants to use as the backup site for checkpoint files. The credential set will be used by services acting on behalf of the user (i.e. the `FaultTolerantJob` service, the `CkptWarehouse` service and RFT). This information will be included as part of the resource creation request used to submit the application.
2. The user submits the application to a `FaultTolerantJob` service (`USER-launch` in Figures 1 and 2).

3. The `FaultTolerantJob` service invokes a `SimpleScheduler` service (`FTquery` in Figures 1 and 2), asking for an available computing resource.
4. The `FaultTolerantJob` service invokes the `CkptJob` service on the selected computing resource to get its delegation set. The `CkptJob` service builds the delegation set dynamically by querying the services hosted in the computing resource (i.e. the `StateExport` service, WS-GRAM and RFT). With this delegation set and one of the delegated credentials of the user, the `FaultTolerantJob` service creates a credential set that will be included as part of the resource creation request used to start the CPPC execution.
5. The `FaultTolerantJob` service invokes the `CkptJob` service on the selected computing resource to start a CPPC execution (`FTlaunch` in Figures 1 and 2).
6. The `CkptJob` service queries the `CkptWarehouse` service to obtain the last consistent set of checkpoint files (`CJquery` in Figures 1 and 2). If there is none, then the execution must begin from scratch. If one is found, that means a previously failed execution is being restarted. Checkpoint files will be moved, as part of the staging of input files, by the WS-GRAM on the selected computing resource when the application was started. In any case, the `CkptWarehouse` service removes any existing checkpoint files from previous attempts.
7. The `CkptJob` service invokes WS-GRAM to initiate the application execution (`CJlaunch` in Figures 1 and 2). A modified WS-GRAM job description is used as parameter of the invocation: a) in the `Environment` element the desired number of processes to be spawned is added; b) in the `StageIn` and `CleanUp` elements the information needed to handle the transfer of checkpoint files is added; and c) information about metadata files to be created by processes is added (i.e. the common directory where to create the files and the name of the first file in the series for each process). Additionally, the administrator may have defined beforehand, in a configuration file, additional information to be added to the job environment.
8. The `CkptJob` service invokes also the `StateExport` service to initiate the exporting of checkpoints. A `StateExport` resource is created for each process of the CPPC application. After that, the `CkptJob` service subscribes to receive notifications from `StateExport` resources indicating the presence of new checkpoint files.

When a process of the CPPC application generates a checkpoint file (`CPPC-write` in Figures 1 and 2) the following steps are taken in sequence:

9. The corresponding `StateExport` resource detects the presence of the newly created checkpoint file (`SEquery` in Figures 1 and 2) by using the technique based on metadata files already explained in Section 3.2.
10. The `StateExport` service uses RFT to export the checkpoint file to a previously agreed backup site (`SEtransfer` in Figures 1 and 2).
11. Once the transfer is finished, the `StateExport` service notifies the `CkptJob` service (`SEnotify` in Figures 1 and 2) about the existence of a new checkpoint file.

12. After receiving the notification, the `CkptJob` service notifies the `CkptWarehouse` service in its turn (`CJnotify` in Figures 1 and 2). The notification includes information about the process that generated the checkpoint file.
13. When, upon arrival of more recent checkpoint files, a new consistent state in composed, the `CkptWarehouse` service deletes obsolete files by using RFT (`CWdelete` in Figures 1 and 2).

Currently two general types of execution failures are being considered: failures in the CPPC application execution, or failures in the computing resource. In both cases the `FaultTolerantJob` service is finally aware of the failed execution and a new restart is initiated going back to step 3. Note that the new execution can be potentially started in a different computing resource but it has to use the same `CkptWarehouse` service.

The process ends when the execution terminates successfully. In that case, the corresponding `StateExport` resources are deactivated and the `CkptJob` resource changes its status to *Finished*. All resources are released when the user acknowledges the finished execution.

5 Conclusions and Future Work

Services for fault tolerance are essential in computational grids. The aim of this work is to provide a set of new Grid services for remote execution of fault-tolerant parallel applications. CPPC is used to save the state of MPI processes in a portable manner. The new Grid services automatically ask for the necessary resources; start and monitor the execution; make backup copies of the checkpoint files; detect failed executions; and restart the application automatically.

The Grid services proposed are loosely coupled, up to the point that it is not necessary for them to reside in the same Globus container. Distributing the functionality into a number of separate services improves both modularity and reusability. Also, it allows to easily replace current services by new ones with desirable features. For instance, other scheduler service can be used instead of `SimpleScheduler`.

The functionality of already existing Globus services is harnessed whenever possible: CPPC-G uses WS-GRAM as job manager and to monitor the applications; RFT to transfer the state files; GridFTP to detect new state files; MDS to discover available computing resources; and GIS to authentication, authorization and credential delegation. Additionally, the modifications made to the existing CPPC library have been kept to a minimum.

At the moment, the CPPC-G architecture is not fault-tolerant itself. In the future it is planned to use replication techniques for the `FaultTolerantJob`, `SimplerScheduler` and `CkptWarehouse` services. Other future direction will be to automate the instantiation of `CkptWarehouse` by querying to a MDS index service.

Acknowledgments

This work has been supported by the Ministry of Education and Science of Spain (ref: TIN-2004-07797-C02), Galician Government (ref: PGIDIT04TIC105004PR) and CYTED Program (ref: 506PI0293).

References

1. K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.
2. E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.
3. I. T. Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, July 2006.
4. J. Gailly and M. Adler. ZLib Home Page. http://www.gzip.org/zlib/.
5. J.M. Hélary, R.H.B. Netzer, and M. Raynal. Consistency issues in distributed checkpoints. *IEEE Transactions on Software Engineering*, 25(2):274–281, 1999.
6. National Center for Supercomputing Applications. HDF-5: File Format Specification. http://hdf.ncsa.uiuc.edu/HDF5/doc/.
7. G. Rodríguez, M. J. Martín, P. González, and J. Touriño. Controller/Precompiler for Portable Checkpointing. *IEICE Transactions on Information and Systems*, E89-D(2):408–417, February 2006.
8. G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo. Portable checkpointing of MPI applications. In *Proceedings of the 12th Workshop on Compilers for Parallel Computers (CPC'06)*, pages 396–410, A Coruña, Spain, January 2006.