Scalable Computing: Practice and Experience
Volume 9, Number 2, pp. 101–109. http://www.scpe.org
ISSN 1895-1767
© 2008 SCPE

A FAULT TOLERANCE SOLUTION FOR SEQUENTIAL AND MPI APPLICATIONS ON THE GRID[*]

GABRIEL RODRÍGUEZ, XOÁN C. PARDO, MARÍA J. MARTÍN, PATRICIA GONZÁLEZ, AND DANIEL DÍAZ[†]

Abstract. The Grid community has made an important effort in developing middleware to provide different functionalities, such as resource discovery, resource management, job submission or execution monitoring. As part of this effort this paper addresses the design and implementation of an architecture (CPPC-G) based on services to manage the execution of fault tolerant applications on Grids. The CPPC (Controller/Precompiler for Portable Checkpointing) framework is used to insert checkpoint instrumentation into the code of sequential and MPI applications. Designed services will be in charge of submission and monitoring of the execution of CPPC-instrumented applications, management of checkpoint files generated by the fault-tolerant applications, and detection and automatic restart of failed executions.

Key words: fault tolerance, grid computing, Globus, MPI, checkpointing

1. Introduction.

Parallel computing evolution towards cluster and Grid infrastructures has created new fault tolerance needs. As parallel platforms increase their number of resources, 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 of the 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 [4], each having its own pros and cons. The advent of the Grid requires the evolution of checkpointers towards portable tools focusing on providing the following fundamental features:

(i) *OS-independence:* checkpointing strategies must be compatible with any given operating system. This means having at least a basic modular structure to allow for substitution of certain critical sections of code (e.g. filesystem access) depending on the underlying OS.

(ii) *Support for parallel applications with communication protocol independence:* the checkpointing framework should not make any assumption about the communication interface or implementation being used. Computational Grids include machines belonging to independent entities which cannot be forced to provide a certain version of the MPI interface. Even recognizing the role of MPI as the message-passing de-facto standard, the checkpointing technique cannot be theoretically tied to MPI in order to provide a truly portable, reusable approach.

(iii) *Reduced checkpoint file sizes:* the tool should optimize the amount of data being saved, avoiding dumping state which will not be necessary upon application restart. This improves performance, which depends heavily on state file sizes. It also enhances performance in case of process migration in computational Grids.

(iv) *Portable data recovery:* the state of an application can be seen as a structure containing different types of data. The checkpointing tool must be able to recover all these data in a portable way. This includes recovery of opaque state, such as MPI communicators, as well as of OS-dependent state, such as the file table or the execution stack.

CPPC (Controller/Precompiler for Portable Checkpointing) [3, 19, 20] provides all these features which are key issues for fault tolerance support on heterogeneous systems. CPPC[1] appears to the user as a runtime library containing checkpoint-supporting routines, together with a compiler tool which automates the use of the library. The CPPC Compiler automatically inserts checkpoint instrumentation, composed of CPPC Library calls and flow control code. The analyses and transformations performed by the compiler completely automate the instrumentation process.

[*] This work has been supported by the Spanish Ministry of Education and Science (TIN-2004-07797-C02, TIN2007-67537-C03-02 and FPU grant AP-2004-2695) and CYTED (Iberoamerican Programme of Science and Technology for Development) Project (ref. 506PI0293).

[†] Computer Architecture Group, University of A Coruña, SPAIN. (grodriguez@udc.es).

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

This paper introduces CPPC-G, a set of new Grid services[2] implemented on top of Globus 4 [5], which is able to manage the execution of CPPC-instrumented fault tolerant applications (CPPC applications from now on). The designed set of services will be in charge of submitting and monitoring the 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 an MPI application are executed on the same computing resource (e.g. a cluster or an MPP machine with MPI installed). Upon a failure, CPPC-G services will restart the application from the most recent 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 an 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 describes related work in the field. Finally, Section 6 concludes the paper.

2. The CPPC framework. 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 computers using incompatible versions of software facilities, such as different implementations for communication interfaces.

CPPC (Controller/Precompiler for Portable Checkpointing) 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 section details various aspects of CPPC design associated with these major issues.

2.1. Portability. A state file is said to be portable if it can be used to restart the computation on an architecture (or OS) different from that where the file was generated on. This means that state files should not contain hard machine-dependent state, which should be recovered at restart time using special protocols. The vast majority of checkpointing research has focused on systems implemented either inside the OS kernel or immediately above the OS interface. This kind of solutions generally become locked into the platform for which they were originally developed. For instance, when checkpointing parallel communication APIs, such as MPI, the typical approach has been to modify the existing implementations of such APIs. The problem arises when, for this approach to be practical, it becomes necessary to adopt the modified libraries in real systems, that use already highly tuned and optimized communication libraries. Other solutions store in the checkpoint file the outcome of the APIs functions, becoming dependent of its implementation.

The solution used in CPPC is to recover non-portable state by means of the re-execution of the code responsible for creating such opaque state in the original execution. Moreover, in CPPC the effective data writing will be performed by a user-selected writing plugin, each using its own format. This enables the restart on different architectures, as long as a portable dumping format is used. Currently, a writing plugin based on HDF5 [17] is provided.

2.2. Memory requirements. The solution of large real scientific problems may need the use of large computational resources, both in terms of CPU effort and memory requirements. Thus, many scientific applications are developed to be run on a large number of processors. The full checkpointing of this kind of applications will 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 only needed upon restart. Besides, a compressed format based on the ZLib library [6] is included. This does not only help saving disk space and network transfers (if needed), but also can improve performance when working with large datasets with high compression rates. A multithreaded dumping option is also provided to improve performance when working with large datasets. If a failure occurred

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

in the checkpointing thread, inconsistent checkpoint files would be created. CPPC generates a CRC-32 for the checkpoint file. This CRC-32 is checked upon restart to ensure file correctness.

2.3. Global consistency. When checkpointing parallel applications, special considerations regarding message-passing have to be taken to ensure that the coordination implicitly established by the communication flow between processes is not lost when restarting the application. If a checkpoint is placed in the code between two matching communication statements, an inconsistency will occur when restarting the application, since the first one will not be executed. If it is a send statement, the message will not be resent and becomes an in-transit message. If it is a receive statement, the message will not be received, becoming a ghost message. Checkpoint consistency has been well-studied in the last decade [4]. Approaches to consistent recovery can be categorized into different protocols: uncoordinated, coordinated and message-logging. In uncoordinated checkpoint protocols the checkpoint of each process is executed independently of the other processes, leading to the so called domino effect (processes may be forced to rollback up to the beginning of the execution). Thus, these protocols are not used in practice. An important drawback, both of coordinated protocols and message-logging solutions, is their scalability. Increasing the number of processors will multiply the number of flying messages, thus enlarging the computation per process needed to be protocol-compliant.

CPPC avoids the overhead caused by coordination and message-logging by focusing on SPMD parallel applications and using a spatially coordinated approach. Checkpoints are taken at the same relative code locations by all processes, without performing interprocess communications or runtime synchronization. To avoid problems caused by messages between processes, checkpoints must be inserted at points where it is guaranteed that there are no in-transit, nor ghost messages. These points will be called safe points. Checkpoints generated in safe points are transitless and consistent, both being conditions for a checkpoint to be called strongly consistent [10]. Safe point identification and checkpoint insertion is automatically performed by the CPPC compiler.

3. CPPC-G design. In this section the design and implementation of a set of Grid services for the remote execution of CPPC applications is described. They have been implemented on top of Globus 4 using the Java API. 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. System architecture. Figure 3.1 shows the proposed CPPC-G architecture that comprises a set of five new services that interact with Globus RFT [8] and WS-GRAM services [9]. A `FaultTolerantJob` service is invoked to start the fault-tolerant execution of a CPPC application. `CkptJob` services provide remote execution functionalities. Available computing resources hosting a `CkptJob` service are obtained from a `SimpleScheduler` service. `StateExport` services are responsible for tracking local checkpoint files periodically stored by the CPPC application. And finally, `CkptWarehouse` services maintain metadata and consistency information about stored checkpoint files. 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. 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. `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. When the replication is finished 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 End-Point Reference (EPR) of a `CkptWarehouse` service and a `StateExport` resource description element, that will be used as a template to create `StateExport` resources. The `CkptJob` service can be configured to register useful information with an MDS index service in the form of a sequence of tags. These tags are specified by the administrator in a configuration file and used to indicate particular properties (e.g. that MPI is installed in the computing resource).

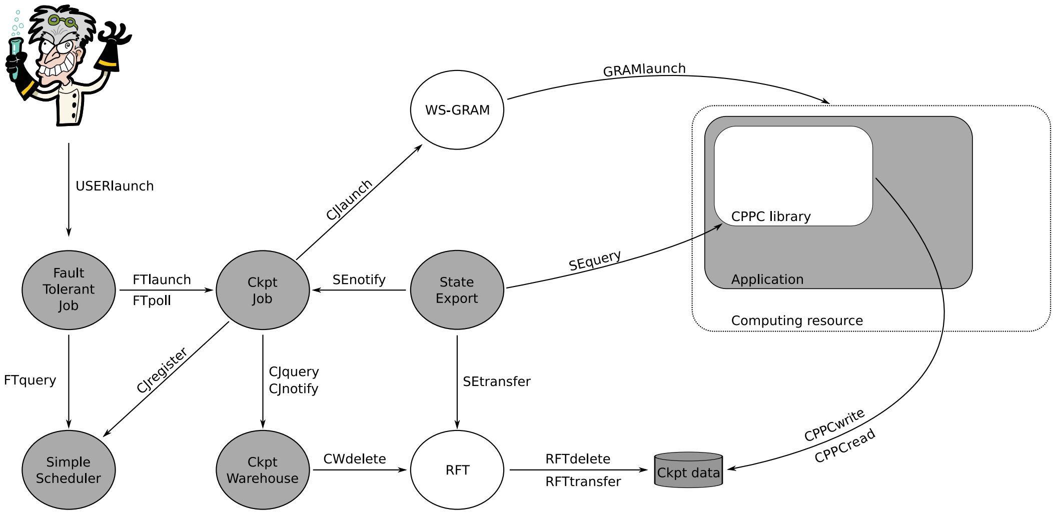

Fig. 3.1. *CPPC-G system architecture*

The CkptWarehouse service maintains metadata about checkpoint files generated by running CPPC applications. 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. 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).

The SimpleScheduler service keeps track of available computing resources hosting a CkptJob service. The service subscribes to an MDS 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 of 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 which 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. The monitoring of the execution is done by periodically polling the state of the computing resource. In case of failure, the execution is restarted automatically. In case the execution fails, FaultTolerantJob requests another node from the scheduler and tries to restart the application there from the last saved state. 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 the precise CkptJob service to be invoked to execute the application is known.

3.2. Component deployment. Figure 3.2 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, SimpleScheduler and FaultTolerantJob services will reside in other computing resources. It is enough that one instance of each of these last three services exists in a Grid for the system to work. 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 shown 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

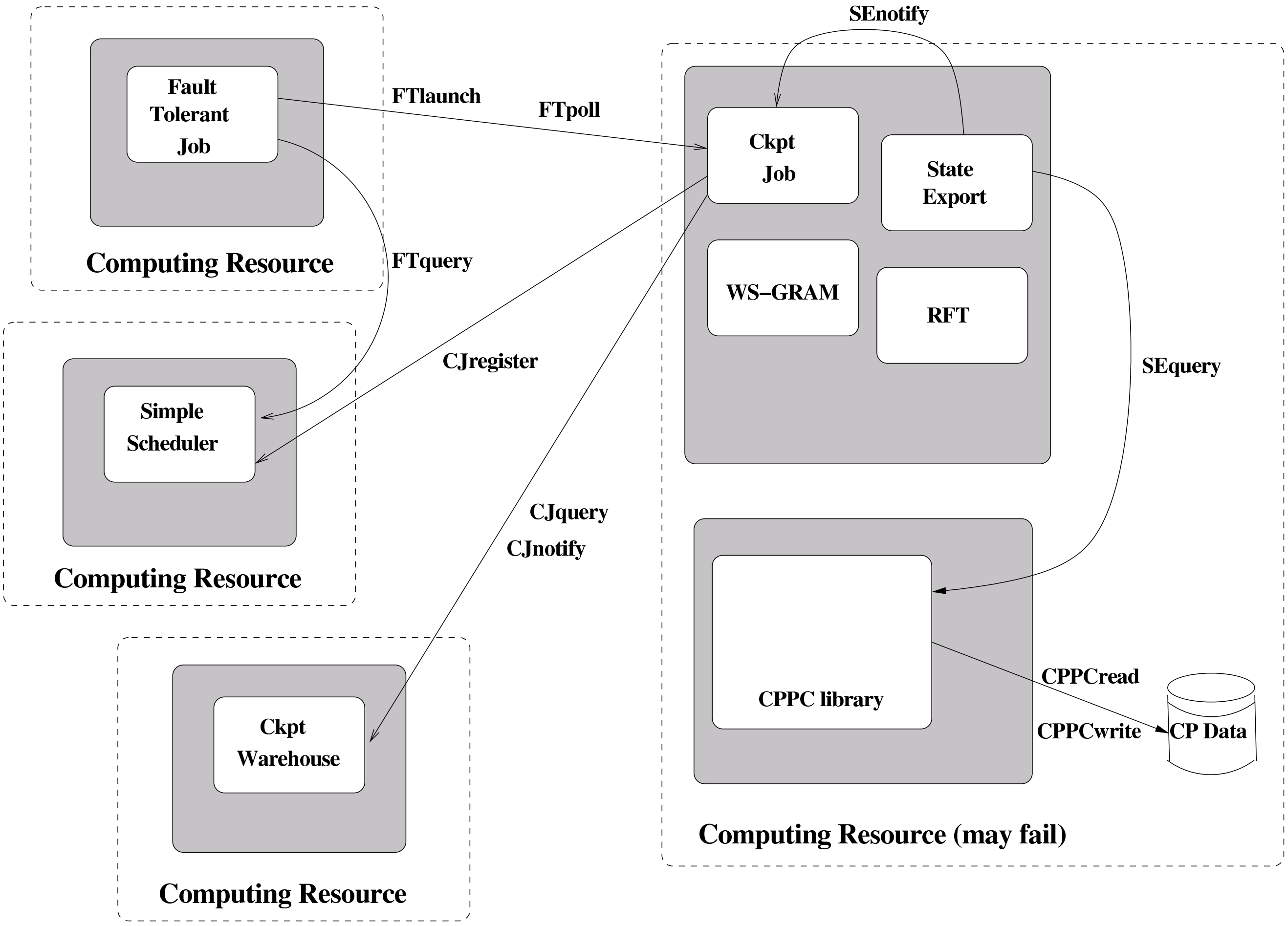

Fig. 3.2. *CPPC-G deployment*

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 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.

3.3. Some implementation issues. In this section some general questions, that are common to the implementation of all the services, are discussed. They are related to management of long operations, security issues, chained invocations among services, resource creation requests and provided clients.

3.3.1. 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, an 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 this paper, the terms resource and service are 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.

3.3.2. 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 is not known beforehand because it is dynamically selected. Automatic delegation has been used in these cases.

3.3.3. 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 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.

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

(i) *The initial termination time of the resource.*

(ii) *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).

(iii) *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).

(iv) *A refresh specification.* For each of the services the resource is expected to invoke, a lifetime refresh period and a ping period are specified. The lifetime refresh period is the amount of seconds to extend the lifetime of resources in invoked services. The ping period is 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.

3.3.5. 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 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.

4. CPPC-G operation. To initiate the execution of an application the following steps are taken in sequence. It is assumed that, before the user submits an application, all available `CkptJob` services are already registered with a `SimpleScheduler` service (`CJregister` in Figure 3.1):

1. In order to prepare the application submission, the user must create in advance an instance of the `CkptWarehouse` service and a credential set. This information will be included as part of the resource creation request used to submit the application.

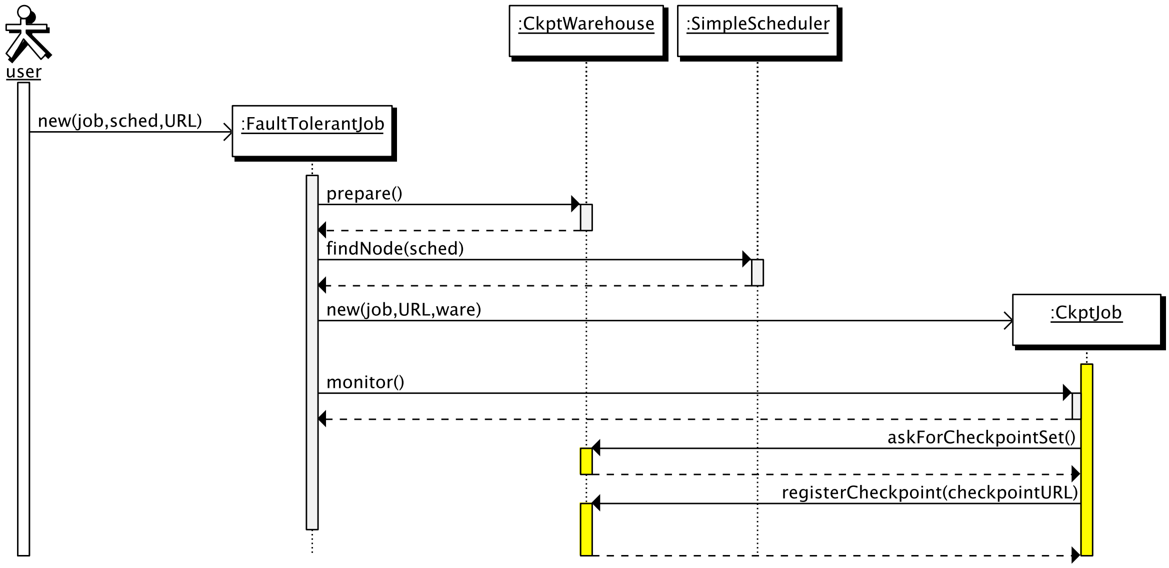

Fig. 4.1. *Sequence diagram for a fault-free execution*

2. The user submits the application to a `FaultTolerantJob` service (`USERlaunch` in Figure 3.1).
3. The `FaultTolerantJob` service invokes a `SimpleScheduler` service (`FTquery` in Figure 3.1), asking for an available computing resource.
4. The `FaultTolerantJob` service queries 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 Figure 3.1).
6. The `CkptJob` service queries the `CkptWarehouse` service to obtain the last consistent set of checkpoint files (`CJquery` in Figure 3.1). Checkpoint files will be moved, as part of the staging of input files, by WS-GRAM on the selected computing resource when the application was started.
7. The `CkptJob` service invokes WS-GRAM to initiate the application execution (`CJlaunch` in Figure 3.1).
8. The `CkptJob` service invokes also the `StateExport` service to initiate the exporting of checkpoints.

When a process of the CPPC application generates a checkpoint file (`CPPCwrite` in Figure 3.1) the following steps are taken in sequence:

9. The corresponding `StateExport` resource detects the presence of the newly created checkpoint file (`SEquery` in Figure 3.1) by using the technique based on monitoring 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 Figure 3.1).
11. Once the transfer is finished, the `StateExport` service notifies the `CkptJob` service (`SEnotify` in Figure 3.1) 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 Figure 3.1). 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 is composed, the `CkptWarehouse` service deletes obsolete files by using RFT (`CWdelete` in Figure 3.1).

A simplified sequence diagram showing a typical fault-free execution is shown in Figure 4.1.

Currently two general types of execution failures are being considered: failures in the CPPC application execution, or failures in the `CkptJob` service. In both cases the `FaultTolerantJob` service is finally aware of the failed execution and a restart is initiated going back to step 3. The process ends when the execution terminates successfully. All resources are released when the user acknowledges the finished execution.

5. Related work. Important effort has been made in developing middleware to provide Grids with functionalities related to application execution. However, support for fault tolerant execution is either lacking or limited. WS-GRAM [9], the execution manager of the Globus Toolkit, handles file transfer before and after an application execution, but offers no handling of the checkpoint files generated while the execution is underway. Other fault tolerance-related functionalities are absent too.

GridWay [11, 12] is a Grid-level scheduler provided with the Globus Toolkit (interfacing with other Grid systems is also possible). It handles the search for nodes that are suitable to the needs of the user. It offers checkpointing support for jobs, and allows job migration. The checkpointing system of Gridway does not cover message-passing applications that perform distributed checkpointing, which requires global checkpoint consistency determination and garbage collection of obsolete checkpoint files.

Regarding checkpoint file storage, the usual solution in computational Grids consists in storing several replicas of files in dedicated servers managed by replica management systems [1, 18]. For opportunistic Grids, a middleware that provides reliable distributed data storage using the free disk space from shared Grid machines is presented in [2].

Several approaches for the implementation of fault tolerance in message-passing applications exist. MPICH-GF [21] is a checkpointing system based on MPICH-G2 [14], a Grid-enabled version of MPICH. It handles checkpointing, error detection, and process restart in a manner transparent to the user. But, since it is a particular implementation of MPICH, it can not be used with other message-passing frameworks. The checkpointing is performed at a data segment level, that is, it stores the entire application state, thus generating non-portable files. To achieve global consistency MPICH-GF uses process coordination, which is a non-scalable approach.

There have been a number of initiatives towards achieving fault tolerance on Grids. The Grid Checkpoint and Recovery (GridCPR) Working Group [7] of the Global Grid Forum was concerned with defining 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. A key feature of Grid Checkpoint Recovery service was recoverability of jobs among heterogeneous Grid resources.

The CoreGrid checkpointing work group of the CoreGrid Network of Excellence has proposed an alternative Grid checkpointing architecture [13, 15]. The difference with the GridCPR proposal is that the latter assumes that the checkpointing tool should be a part of the application, and would be tightly connected to various Grid services such as communication, storage, etc. In the CoreGrid proposal, checkpointers are system-level and external to the applications.

MIGOL [16] is a fault-tolerant and self-healing grid middleware for MPI applications built on top of the Globus Toolkit. MIGOL supports the migration of MPI applications by checkpointing and restarting the application on another site. However, as for now the current version of the middleware depends of locally stored checkpoints, which have to be accessible after an execution failure to enable auto-recovery. No checkpoint replication is performed. This means that if the machine goes down or becomes otherwise inaccessible, application execution must start from the beginning.

6. 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 sequential and MPI processes in a portable manner. The new Grid services ask for the necessary resources; start and monitor the execution; make backup copies of the checkpoint files; detect failed executions; and restart the application. All this is done in a completely transparent fashion.

The proposed Grid services 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`. Also, the CPPC framework could be replaced by any other checkpoint framework provided that it generates the necessary metadata files.

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 for 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 itself fault-tolerant. 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 finding of potential checkpoint backup repositories over the Grid by querying a MDS index service.

REFERENCES

[1] A. L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf, *Performance and Scalability of a Replica Location Service*, in HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, 2004, IEEE Computer Society, pp. 182–191.

[2] R. Y. de Camargo, F. Kon, and R. Cerqueira, *Strategies for Checkpoint Storage on Opportunistic Grids*, IEEE Distributed Systems Online, 7 (2006), p. 1.

[3] D. Díaz, X. Pardo, M. J. Martín, P. González, and G. Rodríguez, *CPPC-G: Fault-Tolerant Parallel Applications on the Grid*, in 3rd Workshop on Large Scale Computations on Grids (LaSCoG'07), vol. 4967 of Lecture Notes in Computer Science, Springer, 2008.

[4] 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 (2002), pp. 375–408.

[5] I. Foster, *Globus Toolkit Version 4: Software for Service-Oriented Systems*, Journal of Computer Science and Technology, 21 (2006), pp. 513–520.

[6] J. Gailly and M. Adler, *ZLib Home Page*. `http://www.gzip.org/zlib/`.

[7] Global Grid Forum, *Grid Checkpoint Recovery Working Group*. `http://forge.ogf.org/sf/projects/gridcpr-wg`

[8] Globus Alliance, *RFT: Reliable File Transfer Service*. `http://globus.org/toolkit/docs/4.0/data/rft/`

[9] ———, *WS-GRAM Execution Management Service*. `http://globus.org/toolkit/docs/4.0/execution/wsgram/`

[10] J. Hélary, R. Netzer, and M. Raynal, *Consistency Issues in Distributed Checkpoints*, IEEE Transactions on Software Engineering, 25 (1999), pp. 274–281.

[11] E. Huedo, R. S. Montero, and I. M. Llorente, *The Grid Way Framework for Adaptive Scheduling and Execution on Grids*, Scalable Computing: Practice and Experience, 6 (2005), pp. 1–8.

[12] ———, *A Modular Meta-Scheduling Architecture for Interfacing with pre-WS and WS Grid Resource Management Services*, Future Generation Computing Systems, 23 (2007), pp. 252–261.

[13] G. Jankowski, R. Januszewski, R. Mikolajczak, and J. Kovacs, *Grid Checkpointing Architecture—a Revised Proposal*, Tech. Report TR-0036, Institute on Grid Information, Resource and Workflow Monitoring Systems, CoreGRID—Network of Excellence, May 2006.

[14] N. T. Karonis, B. Toonen, and I. Foster, *MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface*, Journal of Parallel and Distributed Computing, 63 (2003), pp. 551–563.

[15] J. Kovacs, R. Mikolajczak, R. Januszewski, and G. Jankowski, *Application and Middleware Transparent Checkpointing with TCKPT on Clustergrid*, in Distributed and Parallel Systems—Cluster and Grid Computing, Proceedings of 6th Austrian-Hungarian Workshop on Distributed And Parallel Systems (DAPSYS), P. Kacsuk, T. Fahringer, and Z. Németh, eds., Springer Verlag, 2007, pp. 179–189.

[16] A. Luckow and B. Schnor, *Migol: A Fault-Tolerant Service Framework for MPI Applications in the Grid*, Future Generation Computer Systems—The International Journal of Grid Computing: Theory, Methods and Applications, 24 (2008), pp. 142–152.

[17] National Center for Supercomputing Applications, *HDF-5: File Format Specification*. `http://hdf.ncsa.uiuc.edu/HDF5/doc/`.

[18] M. Ripeanu and I. Foster, *A Decentralized, Adaptive Replica Location Mechanism*, in HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, 2002, IEEE Computer Society, p. 24.

[19] G. Rodríguez, P. González, M. J. Martín, and J. Touriño, *Enhancing Fault-Tolerance of Large-Scale MPI Scientific Applications*, in PaCT, V. E. Malyshkin, ed., vol. 4671 of Lecture Notes in Computer Science, Springer, 2007, pp. 153–161.

[20] G. Rodríguez, M. J. Martín, P. González, and J. Tourio, *Controller/Precompiler for Portable Checkpointing*, IEICE Transactions on Information and Systems, E89-D (2006), pp. 408–417.

[21] N. Woo, H. Jung, H. Y. Yeom, T. Park, and H. Park, *MPICH-GF : Transparent Checkpointing and Rollback-Recovery for Grid-enabled MPI Processes*, IEICE transactions on information and systems, 87 (2004), pp. 1820–1828.

Edited by: Dana Petcu, Marcin Paprzycki
Received: April 30, 2008
Accepted: May 16, 2008