Rosa Badia, IBM Barcelona
Robert Hood, NASA
Thilo Kielmann, Vrije Universiteit
Christine Morin, INRIA Rennes
Stephen Pickles, University of Manchester
Massimo Sgaravatto, INFN Padova
Paul Stodghill, Cornell University (editor)
Nathan Stone, Pittsburgh Supercomputing Center
Heon Y. Yeom, Seoul National University
*fixme: August ??, 2004*

GridCPR
Request for Comments: GWD-XXX-00x-4
Obsoletes: number
Category: informational

# Use Cases for Grid Checkpoint and Recovery

## Status of This Memo

This memo provides information to the Grid community regarding use case scenarios for Grid Checkpointing and Recovery. It does not define any standards or technical recommendations. Distribution is unlimited. This is a DRAFT document and continues to be revised.

## Abstract

This document describes use cases to be addressed by the Grid Checkpoint and Recovery Working Group (GridCPR WG). The scenarios are also used to determine a set of requirements for these standards.

## Full Copyright Notice



## Intellectual Property Statement

The GGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the GGF Executive Director.

# Contents


*fixme: Usecase and use case are used inconsistently. Should it be use-case?*
*fixme: check for consistent usage of "application" and "job".*

# 1 Introduction

One of the goals of the Grid Checkpointing and Recovery Working Group (GridCPR WG) is to,

> define 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. [ 29]

In order to understand the requirements that these API and services must meet, it is necessary to understand the situations in which they will be used.

The purpose of this document is to enumerate usage scenarios that the GridCPR WG have decided must be addressed by its specifications. These scenarios will be used to derive a set of requirements for the GridCPR API and services.

Note that the scenarios described below are not intended to partition jobs. That is, a single application may use checkpoint and restart in a variety of different ways. For example, in the RealityGrid [ 27, 23] project there are applications that use checkpoint and restart for fault-tolerance (Section 3), job migration (Section 4) and debugging (Section 5) simultaneously.

# 2 Terminology

Checkpointing and Recovery (CPR) can occur at many different levels of granularity. At the most basic level, CPR can be provided for a single sequential process on a single machine. Providing CPR for multiple processes on a parallel machine or even tasks executing on different machines is far more complex.

In this document, we refer to the basic computational processing unit that can be checkpointed as a *job*. The mandate of the Certain requirements, such as quality of service, will be made on intra-job CPR, but not inter-job CPR.

*fixme: An* application *is program executable. A job is an individual instance of the execution of an application.*

This usecases document does not preclude applications using the GridCPR API for inter-job CPR. In fact, existing workflow applications use job-level CPR and checkpoint meta-data in order to provide inter-job fault-tolerance. However, while checkpointing each of the constituent jobs falls within the GridCPR charter, the inter-job mechanisms and APIs is out of our scope.

# 3 Fault-Tolerance

## 3.1 Introduction

One of the most common reasons for using CPR is to provide some amount of fault-tolerance or failure recovery to a job.

There are two primary customers in this scenario, the system provider and the user. The *system provider* is the individual or institution that is providing a computational resource (e.g., workstation, cluster, supercomputer) for other parties to use. The *user* is the party that is interested in using this resource to run a job.

## 3.2 Terminology

*fixme: hoist to Section 2?*

Before proceeding further, it is necessary to describe the *fault model* that we wish this scenario to address. First, we will assume that only hardware failures are addressed by this scenario. In particular, we will not attempt to detect or handle application bugs or hacker attacks.

Second, we will assume that these failures can be modeled as *fail-stop* failures [22][1]. We do not consider the more general class of Byzantine failures[2].

Third, we will assume that, once any part of the computational resource has failed, it is impossible for the job to continue execution without initiating some global form of recovery. That is, we do not consider scenarios where the job continues executing on fewer processor nodes[3].

Instead of enumerating a single or small set of scenarios, we will consider a class of fault-tolerance scenarios. The basic structure of these scenarios is as follows:

**Checkpointing** While the job is running, the state of the job is saved.

**Failure Processing** In the event of failure, the job will be resubmitted to a computational resource to continueexecution.

**Recovery** Upon execution, the state of the job is restored, and the job continues from its previously saved state.

## 3.3 Checkpointing

There are two fundamental approaches to saving the state of a job.

In *System-Level Checkpointing (SLC)*, the job's system state is captured and saved. This may mean saving the processor's registers, stack, and memory at the point that the checkpoint is taken. MOSIX [2] and Condor [21] are two well-known system-level CPR systems, but many others have been developed as well [32, 25].

[1] Certain non-application software failures, such as operating system failures, may appear and be treated as hardware failures using this model.

[2] Certain hardware failures, such as transient network failures or memory corruption, may not cause any processor to fail and stop. However, as long as these failures can be detected, they can handled by causing the processor to shutdown.

[3] It may be possible for the job to *resume* execution on fewer nodes.

In *Application-Level Checkpointing (ALC)*, state-saving is done within the job's application source code. In this case, the critical program variables and data structures are saved. Very often, ALC is implemented directly by the developer [10], but it can also be added to application source code automatically using compiler-based tools [5]or programming environment [14] .

In parallel jobs, it is necessary to obtain a consistent global snapshot of the each process and the state that is shared between them (e.g., messages in flight, shared global memory). For data-parallel jobs, very often, it is possible to save all of the state with respect to a global or application point of view [28]. For other styles of parallelism (i.e., SPMD, MIMD), more complicated protocols may be required in order to ensure that a correct snapshot is saved. See [11] for a survey of these techniques.

## 3.4 Failure Processing

The first requirement for processing a failure is to recognize that a failure has occurred. Very often, either the user or system provider can manually determine when a failure has occurred by monitoring the computational resources. There are a number of heartbeat mechanisms [15, 13, 9] that have been proposed for detecting fail-stop failures automatically and providing notification services.

Once a failure has been detected, the job must be resubmitted to a computation resource. For this scenario, we will assume that the job is resubmitted to the same (albeit repaired ) computational resource. Issues involved with recovering on a different computation resource are discussed in Section 4.

Other issues involving failure processing are outside the scope of this working group.

## 3.5 Recovery

Once the job has started executing again after resubmission, its state must be restored to the onesaved during the Checkpointing phase. The details of this are dependent upon how the checkpoint was taken in the first place (e.g, SLC or ALC, sequential or parallel). However, one requirement of these approaches is that the state that was saved during the Checkpointing phase must be available during the Recovery phase.

## 3.6 Accounting

System administrators may wish to encourage their users to use CPR systems by limiting the refund that is made in the event of a system failure. For example, if a system's refund policy is such that only three hours are credited to a user in the event that the machine fails during a 24 hour job, then the user may be inclined to checkpoint their job every three hours in order to prevent being charged for lost work in the event of failure.

## 3.7 Example systems

The checkpoint system for the Pittsburgh Supercomputer Center's Terascale Computing System [30] allows for the automated recovery of jobs following both machine failures and scheduled maintenance periods. As an added feature, this system allows that any time lost by the user process because of machine failure between the time of the failure and the time of the last checkpoint can be automatically credited back to their allocation.

CPR in the European DataGrid [12] is used to provide some form of fault-tolerance to jobs, which is particularly important for long-running jobs, such as those in High Energy Physics. In this system, the developer is responsible for determining the job state that must be saved and restored. The system is responsible for noticing failures and automatically resubmitting jobs for further execution.

The RealityGrid [27, 23] project provides support for jobs that contain ALC functionality. This support enables jobs to implement fault-tolerance and to implement strategies for long running computations to save state at the end of a fixed length batch run. In either case, jobs can be restarted from checkpoints in subsequent batch allocations.

MPICH-GF [19] supports user-transparent SLC for fault tolerance of MPI jobs running within a homogeneous computing environment. MPICH-GF is provided as a library that is linked with the unmodified application source code. The system provides checkpointing and message logging and a job management system that monitors the job, periodically sends checkpoint signals to the job, and restart the job if a failure occurs.

The $C^3$ system is a precompiler that can be used to add ALC to existing application source code. The developer adds directives to the application source code to indicate the points at which checkpoints can be taken, and the $C^3$ uses

program transformations to augment the application with code to save critical program variables and data structures. $C^3$ also uses compiler optimizations to reduce the size and overhead of taking checkpoints. The $C^3$ system also provides features to ensure that consistent checkpoints of MPI jobs [4, 6].

The Kerrighed system [1] provides a single system image OS than can be run within and across clusters. Its goal is to provide SLC of sequential and parallel jobs, and to enable transparent failover and migration of such jobs within a cluster federation. Loosely-coupled distributed jobs are handled by forcing processes to checkpoint when certain communication occurs and by affixing certain causality information to messages.

# 4 Job Migration

In this scenario, the user would like to save the current state of the job in order to resume execution at some point in the future on the same or different computational resource.

In the Fault-Tolerance scenario, we assumed that a job would restart on the same machine on which it was initially executed. With Job Migration (and, in fact, with Fault-Tolerances in practice), the job may be resumed on a different machine than its initial execution. This new machine may have a different number of processing nodes and a different processor architecture. "Portable" checkpoints can be generated using over-decomposition and by using architecture independent encoding methods (e.g., XDR [31] and HDF5 [16]).

*Over-decomposition* is a technique whereby a job's work is divided into more units than there are physical processors. That is, if a job is run on $P$ processors, than instead its work being divided into $P$ units, it may be divided into $cP$, where $c$ is a factor greater than 1. While this approach can introduce some amount of overhead to an job's execution, it does provide flexibility in how work units are assigned to processors. For example, if the job is restarted on $Q$ processors, then work units can be repartitioned so that each processor is assigned $cP/Q$ units.

The mechanisms for implementing CPR for job migration are similar to those discussed in Section 3. For example, Condor [21] provides transparent migration of sequential jobsin a homogeneous computing environment. Dome [3] is an example of an application framework that enables heterogeneous checkpointing and job migration via over-decomposition. AMPI [18] is a runtime system that enables a job to automatically migrate between machines with a different number of processing nodes. PORCH [26] is an example of a compiler system that enables a job to automatically migrate between machines with different architectures. All of these systems are potential customers of the GridCPR APIs.

For jobs that support ALC, the RealityGrid [27, 23] and GridLab [**?**] projects provides support for job migration in heterogeneous computing environments. RealityGrid does not directly provide heterogeneous checkpointing capability: the responsibility of generating portable and repartitionable checkpoints lies with the developer.

# 5 Debugging

CPR can be used for debugging applications. One example of how this can be done is with the *replay debugger* [33], which enable the developer to run a job apparently in reverse. This is done by periodically checkpointing the job. When the developer wishes to run the job in reverse, from breakpoint $B$ to breakpoint $A$, the job is restored to the a taken prior to $A$ checkpoint and then allowed to run forward until it reaches $A$.

An example of a debugging system that uses CPR is the O'Caml debugger [20]. CPR is also being added to the P2D2 debugging system [17].

The RealityGrid [27, 23] project enables jobs to provide "wind back" capabilities that are based upon the CPR system. Wind back can be used not only for debugging, but for computational steering as well [7, 8]. A common use involves the user running a computationally intensive simulation in a mode that produces low-resolution results. The user can then wind back the simulation to a point of interest and rerun it with options to produce high-resolution results only in the region of interest.

# 6 Parametric Applications

There are many scientific applications that explore a certain space of parameters or that make certain non-deterministic decisions. Often, a scientist will want to explore a set of parameter values or consider different non-deterministic

decisions. Each point in this execution at which one of these values is set represents a branch in the execution of the program. Since there can be several such points during execution, in the general case, this exploration can result in a tree of possible execution paths.

CPR can be used to make this exploration more efficient. For instance, if the job takes a checkpoint just before a branch value is chosen, then on a subsequent execution the user can resume the job from the checkpoint and force a different value to be chosen. This eliminates the need to repeat all of the computation that occurred prior to the checkpoint to be repeated.

The RealityGrid [27, 23] project enables this sort of parametric exploration by supporting "checkpoint trees" [24]. That is, the RealityGrid system enables checkpoint files to be linked together in a manner that encodes the causal relationship between them. This enables a user to construct and manage exploration trees. Combined with the visualization tools wind back features discussed in the previous section, these capabilities provide the computational scientist with very powerful tools for scientific discovery.

# 7 Functional Requirements for GridCPR

A GridCPR system will consist of a number of services that provide the functionality. These services will be enumerated and described in a future architecture document. However, in order to build portable applications and tools that use these services a number of features of these systems must be standardized.

In this section we identify the requirements that are imposed by the preceding scenarios.

**Checkpoint Storage** A necessary requirement of this scenario is a means to create, write, read and destroy checkpoints on stable storage. This ability must be provided in the form of an API so that the developer can exploit it directly (manual ALC). This API may also be used by automatic checkpointing tools (automatic ALC and SLC). This API must enable each of the various forms parallel checkpointing (data-parallel, uncoordinated, block and non-blocking coordinated).

An implementation of this API must provide certain QoS guarantees to the user and system provider. For instance, the system provider should be able to specify that checkpoint files are stored on machines that are not part of the computation resource. The user should be able to query the system to determine what guarantees are in place.

An implementation of this API must accommodate failures that occur during the checkpointing process. A common technique used in ALC-based solutions to the fault-tolerance problem is to cycle through two (rarely more) sets of checkpoint files so that if a failure occur while a checkpoint is being taken, recovery from the earlier checkpoint is still possible.

If an application is so designed, it must be able to resume on a machine that is different, both in terms of number of processors and architecture, than the machine on which the checkpoint files were created. This means that it must be possible to transport checkpoint files between these machines.

Since multiple checkpoints may be present at recovery, the user must have the option of specifying which checkpoint to recover from.

*Non-requirement:* The API does not have to provide for mechanisms for initiating checkpoints or for specifying how the checkpoints should be taken. The details of the checkpointing process (ALC vs. SLC, etc.) are left to the application or to other automatic tools.

**Checkpoint Meta-data** It must be possible to annotate checkpoint files with meta-data. For instance, it must be possible to examine stable storage and determine which set of files constitute a complete snapshot of the job. Also, certain non-blocking coordinated checkpoint protocols for parallel jobs require that a set of checkpoint files first be created and then separately be marked as "complete" once the protocol terminates.

Meta-data should provide general-purpose mechanism for annotating checkpoint files. It should be possible to use this meta-data to encode the causal relationships of checkpoint trees that were described in Section 6.

*Non-requirement:* While QoS is a requirement for checkpointing data, it is not a requirement for checkpoint meta-data.

**Failure Detection** The GridCPR API should provide mechanisms for receiving notifications when failures are detected, and for allowing the job to notify the service in the event that it detects a failure. The notification system should differentiate between system detected failures and application initiated failures.

*Non-requirement:* Although these APIs are intended to support job migration, they do not define APIs or services for job migration. In other words, a Grid job migration system might use the CPR APIs to capture job state, but would have to provide additional APIs and services for initiation, resource management, security, etc.

**Other requirements** Checkpoints and their associated metadata must be transportable. That is, it should be possible for an external agent (user, job monitor, etc.) to initiate authorized transfer of checkpoints and their associated metadata from one GridCPR system to another.

Since checkpoints and associated metadata must be transportable between GridCPR systems, this implied that there must be a common encoding that is used when data is transported between systems. It is not a requirement that a GridCPR system store checkpoints or metadata using this encoding, but this encoding must be used during transport.

**Other non-requirements** The details of checkpoint transfer protocols are beyond the scope of this working group.

Issues involving job scheduling are beyond the scope of this work. This includes issues involving how a checkpointable job is initiated or how a checkpointed job is resumed from a checkpoint file.

# 8 Security Considerations

Authentication, Authorization, and Accounting (AAA) must be provided for the job state that is saved to stable storage. This includes, but is not limited to, ensuring that only authorized users are able to restart an job from checkpoints.

The user must be prevented from initiating failures for their own advantage (e.g., in order to obtain refunds for failures).

# 9 Performance Considerations

Systems that implement the GridCPR API and services described above should impose as small an overhead on the job as possible. They should also endeavor to provide good scalability.

Performance during the Checkpointing phase is more important than performance during the Recovery phase. Systems that implement the GridCPR API and services described above may, for instance, defer processing time during Checkpointing until Recovery.

An implementation that uses the Checkpoint Storage API should only observe an overhead from the system when it calls the functions for manipulating checkpoint files. For instance, the Checkpoint Storage system should not significantly slow down a job that never takes a checkpoint.

# 10 Editor Information

Paul Stodghill, Department of Computer Science,
Upson Hall, Cornell University, Ithaca, NY, 14853, USA
Phone: 607-254-8838 Email: stodghil@cs.cornell.edu

# Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0085969. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

# References

[1] Ramamurthy Badrinath, Christine Morin, and Geoffroy Vallée. Checkpointing and recovery of shared memory parallel applications in a cluster. In *Proc. Intl. Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, pages 471–477, May 2003. Held in conjunction with CCGrid 2003. Available at http://www.inria.fr/rrrt/rr-4806.html.

[2] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[3] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[4] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, 2002.

[5] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. $C^3$: A system for automating application-level checkpointing of MPI programs. In *The 16th International Workshop on Languages and Compilers for Parallel Computers (LCPC'03)*, October 2003.

[6] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in an application-level fault tolerant MPI system. In *International Conference on Supercomputing (ICS) 2003*, San Francisco, CA, June 23–26 2003.

[7] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. Computational steering in realitygrid. In *Proceedings of the UK e-Science All Hands Meeting*, September 2-4 2003. Available at http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/179.pdf.

[8] J. Chin, J. Harting, S. Jha, P. V. Coveney, A. R. Porter, and S. M. Pickles. Steering in computational science: mesoscale modelling and simulation. *Contemporary Physics*, 44:417–434, 2003. Available at http://taylorandfrancis.metapress.com/openurl.asp?genre=article&eissn=1366-5812&volume=44&issue=5&spage=417 .

[9] The DataGrid Project. Gdmp heartbeat monitor. http://project-gdmp.web.cern.ch/project-gdmp/gdmp_hb/.

[10] G. Deconinck, J. Vounckx, R. Lauwereins, and J. Peperstraete. A user-triggered checkpointing library for computation-intensive applications, 1995.

[11] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[12] Alessio Gianelle, Rosario Peluso, and Massimo Sgaravatto. Datagrid: Job partitioning and checkpointing. https://edms.cern.ch/document/347730, June 3 2002.

[13] The Globus Project. The globus heartbeat monitor specification v1.0. http://www-fp.globus.org/hbm/heartbeat_spec.html.

[14] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing - VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin, 2003. Springer.

[15] I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing*, pages 170–179, 2001.

[16] Hdf5 - a new generation of hdf. http://hdf.ncsa.uiuc.edu/HDF5/.

[17] Robert Hood. P2d2: A portable distributed debugger. http://www.nas.nasa.gov/Groups/Tools/Projects/P2D2/.

[18] Chao Huang, Orion Lawlor, and L. V. Kale. Adaptive MPI. In *Languages and Compilers for Parallel Computers (LCPC)*, 2003.

[19] Sangbum Kim, Namyoon Woo, Heon Y. Yeom, Taesoon Park, and Hyoungwoo Park. Design and implementation of dynamic process management for grid-enabled MPICH. In *Proceedings of the 10th European PVM/MPI Users' Group Conference*, Venice, Italy, September 2003.

[20] Xavier Leroy. Re: [Caml-list] replay debugger. http://caml.inria.fr/archives/200110/msg00033.html, October 4 2001.

[21] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

[22] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, California, first edition, 1996.

[23] Stephen Pickles, Robin Pinning, Andrew Porter, Graham Riley, Rupert Ford, Ken Mayes, David Snelling, Jim Stanton, Steven Kenny, and Shantenu Jha. The realitygrid computational steering api - version 1.0. unpublished, July 9 2003.

[24] Stephen M. Pickles, Peter V Coveney, and Bruce M Boghosian. Transcontinental realitygrids for interactive collaborative exploration of parameter space (triceps). Available at http://www.scconference.org/sc2003/inter_cal/inter_cal_detail.php?eventid=10701#5. Winner of SC'03 HPC Challenge competition in the category "Most Innovative Data-Intensive Application".

[25] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under UNIX. Technical Report UT-CS-94-242, Dept. of Computer Science, University of Tennessee, 1994.

[26] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogenous architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[27] The RealityGrid Project. Realitygrid. http://www.realitygrid.org.

[28] C. Wang S. Flynn Hummel, I. Banicescu and J. Wein. Load balancing and data locality via fractiling: an experimental study. In Boleslaw K. Szymanski and Balaram Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, chapter 7, pages 85–98. Kluwer Academic Publishers, Boston, MA, 1996.

[29] Derek Simmel, Thilo Kielmann, and Nathan Stone. Draft charter v.1.2, Grid Checkpoint Recovery Working Group (GridCPR). Technical report, Global Grid Forum, January 1 2004. Available at http://gridcpr.psc.edu/GGF/charter/GridCPR-WG-charter.1.2.txt.

[30] Nathan Stone, John Kochmar, Raghurama Reddy, J. Ray Scott, Jason Sommerfield, and Chad Vizino. A checkpoint and recovery system for the pittsburgh supercomputing center terascale computing system. http://www.psc.edu/publications/tech_reports/chkpt_rcvry/checkpoint-recovery-1.0.html, December 3 2003.

[31] Sun Microsystems, Inc. RFC 1014 - XDR: External data representation standard. Published by The Internet Engineering Task Force. Available at http://www.ietf.org/rfc/rfc1014.txt.

[32] Yuval Tamir and Carlo H. Sequin. Error recovery in multicomputers using global checkpoints. In *13th International Conference on Parallel Processing*, pages 32–41, Bellaire, MI, August 1984.

[33] Andrew Tolmach. *Debugging Standard ML*. PhD thesis, Princeton University, October 1992.