The Architecture of the XtreemOS Grid Checkpointing Service

John Mehnert-Spahn[1], Thomas Ropars[2], Michael Schoettner[1], and Christine Morin[3]

[1] Department of Computer Science, Heinrich-Heine University, Duesseldorf, Germany
{John.Mehnert-Spahn,Michael.Schoettner}@uni-duesseldorf.de
[2] Université de Rennes 1, IRISA, Rennes, France
Thomas.Ropars@irisa.fr
[3] INRIA Centre Rennes - Bretagne Atlantique, Rennes, France
Christine.Morin@inria.fr

Abstract. The EU-funded XtreemOS project implements a grid operating system (OS) transparently exploiting distributed resources through the SAGA and POSIX interfaces. XtreemOS uses an integrated grid checkpointing service (XtreemGCP) for implementing migration and fault tolerance for grid applications. Checkpointing and restarting applications in a grid requires saving and restoring distributed/parallel applications in distributed heterogeneous environments. In this paper we present the architecture of the XtreemGCP service integrating existing system-specific checkpointer solutions. We propose to bridge the gap between grid semantics and system-specific checkpointers by introducing a common kernel checkpointer API that allows using different checkpointers in a uniform way. Our architecture is open to support different checkpointing strategies that can be adapted according to evolving failure situations or changing application requirements. Finally, we discuss measurements numbers showing that the XtreemGGP architecture introduces only minimal overhead.

1 Introduction

Grid technologies like Globus [5] can help to run businesses in dynamic distributed environments effectively. Grids enable/simplify the secure and efficient sharing and aggregation of resources across administrative domains spawning millions of nodes and thousands of users.

The key idea of the EU-funded XtreemOS project is to reduce the burden on grid administrators and application developers by providing a Linux-based open source Grid operating system (OS). XtreemOS provides a sound base for simplifying the implementation of higher-level Grid services because they can rely on important native distributed OS services, e.g. security, resource, and process management [1]. In XtreemOS a grid node may be a single PC, a LinuxSSI cluster, or a mobile device. LinuxSSI is an extension of the Linux-based Kerrighed single system image operating system [17]. Because of resource constraints mobile nodes act as clients rather than fully fledged grid nodes.

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 429–441, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Grid technologies offer a great variety of benefits but there are still challenges ahead including fault tolerance. With the increasing number of nodes more resources become available but at the same time the failure probability increases. Fault tolerance can be achieved for many applications, in particular scientific applications, using a rollback-recovery strategy. In the simpliest scenario an application is halted and a checkpoint is taken that is sufficient to restart the application in the event of a failure. Beyond using checkpointing for fault tolerance it is also a basic building block for application migration, e.g. used for load balancing.

There are numerous state of the art system-specific checkpointing solutions supporting checkpointing and restart on single Linux machines, e.g. BCLR [8], Condor [10], libCkpt [13]. They have different capabilities regarding what kind of resources can be checkpointed. Furthermore, there are also specific distributed checkpointing solutions for cluster systems dealing with communication channels and cluster-wide shared resources, like the Kerrighed checkpointer [17].

The contribution of this paper is to propose an architecture for the XtreemGCP service integrating different existing checkpointing solutions in order to checkpoint grid applications. This is realized by introducing a common kernel checkpointer API that is implemented by customized translation libraries. The architecture has been implemented currently supporting the BLCR and LinuxSSI checkpointer within the XtreemOS project which is available as open source [1].

The outline of this paper is as follows. In Section 2 we briefly present relevant background information on XtreemOS. Subsequently, we present the architecture of XtreemGCP. In Section 4 we discuss the common kernel checkpointer API and implementation aspects. Other grid related issues including resource conflicts and security are described in Section 5 followed by an evaluation. Related work is discussed in Section 7 followed by conclusions and future work.

2 Checkpointing an Application in XtreemOS

In this paper, we consider sequential, parallel, and distributed applications. We call a job an application that has been submitted to the grid. Each job has a grid-wide unique job id. A job is divided into job-units, a job-unit being the set of processes of a job running on one grid node.

Applications are handled by the application execution management service (AEM). AEM job managers handle jobs life cycle, including submission, execution and termination. Each job is managed by one job manager. A job directory service stores the list of active jobs and the location of their associated job managers. On each grid node, an execution manager handles the job-units running on the node. It performs the actions requested by the job manager and is in charge of controlling and managing job-units.

XtreemOS includes a grid file system called XtreemFS [6], providing location-transparent file access (including replication and striping). XtreemGCP uses XtreemFS as persistent storage for grid checkpoint files.

Checkpoint/restart is of major importance in XtreemOS - it is the basic block for several functionalities: *scheduling*: including job migration and job

suspension, *fault tolerance*: implemented by saving a job snapshot, and *debugging*: allowing to run applications in the past.

3 Architecture of XtreemGCP

The XtreemGCP service architecture, its components and their interactions are explained here in the context of checkpointing and restarting grid applications.

3.1 Grid Checkpointing Services

The XtreemGCP is a layered architecure, see Figure 1. At the grid level, a job checkpointer (JCP) manages checkpoint/restart for one job possibly distributed over many grid nodes. Therefore, it uses the services provided by the job-unit checkpointer (JUCP) available on each grid node. A JUCP controls the kernel checkpointers available on the grid nodes to take snapshots of the job-units processes. The so-called *common kernel checkpointer API*, introduced by XtreemOS, enables the JUCP to address any underlying kernel checkpointer in a transparent way. A translation library implements this API for a specific kernel checkpointer and translates grid semantics into specific kernel checkpointer semantics. In the following we provide details for all these services.

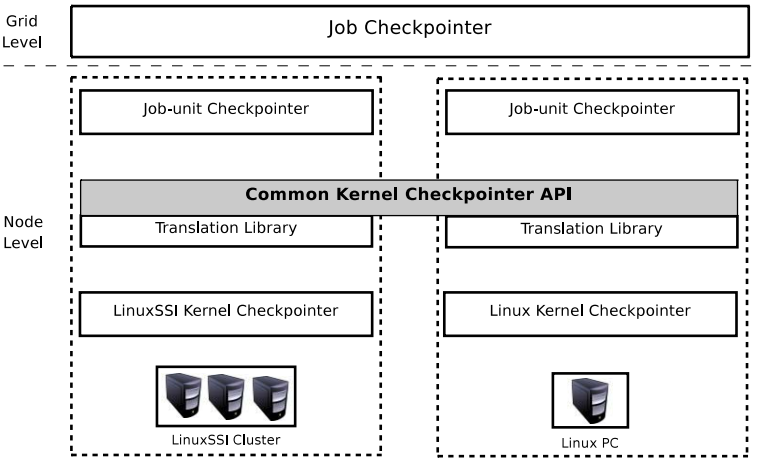

Fig. 1. XtreemOS Grid Checkpointing Service Architecture

There is one **job checkpointer** (JCP) service per job in the grid. It is located on the same node as the corresponding job manager. Furthermore, it distributes the load, induced by XtreemGCP over the whole grid. We use virtual nodes [14] to make JCPs and job managers highly available via service replication.

The JCP has a global view of the job. It knows the job-units composing the job and the job-units location. It controls the underlying components to apply a checkpoint decision. When checkpointing is used to migrate or suspend a job,

the decision is taken by the AEM. In case of checkpoints used for fault tolerance, the JCP is responsible for scheduling the checkpoints.

In coordinated checkpointing, the JCP takes the role of the coordinator. With the help of the JUCPs, it synchronizes the job-units at checkpoint time to guarantee a consistent checkpoint. At restart, it coordinates the job-units to ensure a consistent job state, see Section 3.2.

Because of its global view on the job, the JCP is in charge of creating job checkpoint meta-data. The latter describes the job checkpoint image required at restart to recreate a job and its constituting job-units, see Section 5.2.

There is one **job-unit checkpointer** (JUCP) per grid node. It interacts with the JCP to apply a checkpoint decision. The JUCP is in charge of checkpointing job-units. Therefore, it relies on the common kernel checkpointing API to transparently address a specific kernel checkpointer.

Each grid node comes with heterogenous kernel checkpointers having different calling interfaces, checkpoint capabilities, process grouping techniques, etc. One grid node may also offer several kernel checkpointers. Therefore, we introduce the **common kernel checkpointer API**, see Section 4, that allows the JUCP to uniformly access different kernel checkpointers. A translation library implements the common kernel checkpointer API for one kernel checkpointer. Such a kernel checkpointer-bound translation library is dynamically loaded by the JUCP when the kernel checkpointer is required to checkpoint/restart a job-unit. In general the translation library converts grid semantics into a kernel checkpointer semantics, e.g a grid user ID is translated into the local user id.

A **kernel checkpointer** is able to take a snapshot of a process group, including process memory and diverse process resources, and is able to restart the process group from this snapshot. Unlike checkpointers exclusively implemented in user space, a kernel checkpointer is also capable of restoring kernel structures representing process' resources like process ids or session ids. XtreemGCP also supports virtual machines to be used as kernel checkpointer.

3.2 Checkpointing Strategies

In this section we present how our architecture supports different checkpointing strategies including coordinated and uncoordinated ones. Here we only focus on the interactions between the services. Other issues related to checkpointing, like security or checkpoint file management, are described in Section 5.

Coordinated checkpointing is the protocol used to realise job migration and suspension. In coordinated checkpointing, the job takes the role of the coordinator. It addresses those JUCPs residing on the same grid nodes as the job-units of the job to be checkpointed. Upon receiving the job *checkpoint signal* the JCP requests all involved JUCPs to load the appropriate translation library whose associated kernel checkpointer is capable of checkpointing the job. Afterwards the JCP enters the *prepare phase*. A job is notified of an upcoming system-initiated checkpoint action to prepare itself, e.g. an interactive application notifies the user to be temporarily unavailable. Then, the JUCPs extract the restart-relevant meta-data, described in Section 5.2, with the help of

the kernel checkpointers storing this meta-data in XtreemFS. Afterwards the JCP instructs the JUCPs to enter the *stop phase*. Processes of all involved job-units are requested to execute their checkpoint-related callbacks, see Section 4.3. Then, each job-unit process is stopped from further execution and an acknowledgement is sent by the JUCPs to the JCP. Meta-data are updated in case processes have been created or destroyed between initial saving and process synchronization. The JCP waits for all JUCP acknowledgments to enter the *checkpoint phase*. The kernel checkpointers are requested through the JUCPs to take snapshots of the processes. These snapshots are saved onto XtreemFS. Afterwards the *resume phase* is entered. At *restart* the *rebuild phase* is entered. The JCP parses the checkpoint meta-data of the job to be restarted. This data is used by the job manager, to identify grid nodes with spare resources (PIDs, shmIDs, etc.) being valid before checkpointing in order to reallocate them, and the JUCPs, to select the same kernel checkpointers as the ones used at checkpoint time. Afterwards, the JCP informs all involved JUCPs of the job-units belonging to the job that is to be restarted. Each JUCP gets the JUCP image it needs from XtreemFS and rebuilds the job-unit processes with the help of the kernel checkpointer. Subsequently, the JUCPs enter the *resume phase*. Here all restart callbacks are processed and afterwards the job-units resume with normal execution.

In **uncoordinated checkpointing**, checkpoints of job-units are taken independently thus avoiding synchronization overhead at checkpoint time. The decision to trigger checkpoints is shifted from the JCP to the JUCPs. They can take node and job-unit state-information into account to decide when it is opportune to checkpoint. The common kernel checkpointer API also allows an application to checkpoint a job-unit.

At restart, a consistent global state of the application has to be computed. Therefore dependencies between checkpoints have to be logged during failure-free execution. As described in Section 4.4, the AEM service monitors the processes to provide information about their communications. The involved JUCPs are in charge of analyzing this data in order to compute the dependencies between checkpoints. At restart, the JCP uses the dependency data saved with the checkpoints to compute a consistent global state. Then, it informs the appropriate JUCPs to perform the restart procedure. JUCPs could log messages to avoid the domino effect.

In **application-level checkpointing**, the application itself executes the checkpoint and recovery functions. Here the checkpointing code is embedded in the application, written by the programmer. In case of application-level checkpointing, the application interacts with the JUCP(s). MPI has become a de-facto standard for high performance computing applications. Several MPI libraries provide fault-tolerance mechanisms. Open MPI [7] and LAM/MPI [15] have chosen to implement generic checkpoint/restart mechanisms that can support multiple existing kernel checkpointers. Since this approach fits well with ours, these libraries can be supported by our service, too. When an MPI library is used, the responsibility to coordinate the processes or to detect dependencies is

shifted to this library. On the grid node where the *mpirun* has been launched the JUCP merely initiates checkpointing and manages the checkpoint images created.

3.3 XtreemOS Adaptive Checkpoint Policy Management

Since the grid is a dynamic environment and applications consume resources in an unpredictable manner, XtreemGCP must be able to dynamically adapt its checkpointing policy for efficiency reasons.

As described before, XtreemGCP supports several checkpointing strategies. Both, coordinated and uncoordinated checkpointing, have pros and cons. Adapting checkpointing parameters or switching from one strategy to another depends on various factors, e.g. failure frequency, job behaviour, etc. strategy changes can improve the performance, see [4].

4 Common Kernel Checkpointer API

Obviously, we cannot assume that each grid application or grid node uses the same kernel checkpointer. The *common kernel checkpointer API* bridges differences allowing the JUCPs to uniformly access these different kernel checkpointers. Currently, we have implemented translation libraries for the BLCR and LinuxSSI checkpointers. The translation libraries are able to address user space components and kernel components, too. The following sections give an overview of the API and discuss important aspects of the translation libraries.

4.1 API Overview

The *common kernel checkpointer API* is implemented by the dynamically loadable translation library per kernel checkpointer. The current version of the implementation of the API provides following primitives:

- *xos_prepare_environment:* A process group is searched, containing all processes of job-unit, that can be addressed by the underlying kernel checkpointer. If successful, the process group reference id and type are saved by the JCP with the checkpoint meta-data.
- *xos_stop_job_unit* The execution of checkpoint callbacks is enforced and afterwards, all processes belonging to the job-unit are synchronized.
- *xos_checkpoint_job_unit* A process group checkpoint is taken and saved in a XtreemFS volume.
- *xos_resume_job_unit_cp* All stopped processes are waked up and resume their execution.
- *xos_rebuild_job_unit* The previously stored checkpoint meta-data indicate the appropriate XtreemFS volumes that stores the JCP image of a given checkpoint version number. The checkpointer is instructed to rebuild all structures representing the process group. Processes are still stopped to avoid any interferences.
- *xos_resume_job_unit_rst* The rebuilded processes are waked up and proceed with normal execution. Execution of restart callbacks is enforced.

4.2 Process Groups

Different process grouping semantics are used at AEM and kernel checkpointer level. Instead of a process list, checkpointers like BLCR use an identifier to reference a UNIX process group (addressed via PGID), a UNIX session (addressed via SID), a root-process of a process tree. LinuxSSI uses a LinuxSSI application identifier to checkpoint/restart processes. However, AEM uses only the concept of job-unit (addressed by a jobID) to group processes. Since the AEM JCP initiates checkpoint/restart, it merely has a jobID. In general, if process groups used by AEM and kernel checkpointers mismatch, too many or just a subset of the processes are checkpointed/restarted which results in inconsistencies. Obviously, AEM cannot be hardcoded against one process group type, since various kernel checkpointers use different process groups semantics.

In XtreemGCP this issue is solved by the translation library which ensures, that a kernel checkpointer can reference all processes that AEM captures in a job-unit. In terms of LinuxSSI the translation library manages to put all job-unit processes into an SSI application. The SSI application ID is then determined and can be used for checkpoint/restart. In terms of BLCR the root process is determined based upon the job-unit process list. Since no separate UNIX session or UNIX process group is initiated at job submission, BLCR cannot use these two process group semantics. More information can be found under [11].

Using the root-process of a process tree is not safe. If an intermediate process fails, the subtree cannot be referenced anymore by the root-process. Linux provides so-called *cgroups* [3] - a mechanism to group tasks and to apply special behaviours onto them. *Cgroups* can be managed from user space via a dedicated file system. Since the upcoming Linux-native checkpointer will be based on *cgroups* and LinuxSSI will be ported towards using *cgroups* - a commmonly used process group semantic on user and kernel checkpointer level has been found which avoids the occurence of mismatchings. *Cgroups* allow all processes to be referenced by one identifier even in case of the application internally establishing multiple UNIX process groups or sessions and in case of intermediate process failures.

Integration of *cgroups* is planned by XtreemOS as soon as the Linux-native checkpointer is available. Management of cgroups will also be done within a kernel checkpointer-bound translation library.

4.3 Callbacks

As mentioned before different kernel checkpointers vary in their capability to record and re-establish application states. On the one hand limitations of kernel checkpointers must be handled, e.g. resources that cannot be checkpointed. On the other hand they must be prevented from saving too much to be efficient, e.g. it might not be necessary to save very large files with each checkpoint.

Our solution for both aspects is to provide callback functions for application developers, that are executed before and after checkpointing (pre/post-checkpoint callbacks) and after restart. Resources can be saved in an application-customized

way and unnecessary overhead can be reduced, e.g. large files can be closed during checkpointing and re-opened at restart. Integration of callbacks into the execution of checkpoint/restart has been proposed and implemented by BLCR, [15]. We have extended LinuxSSI towards this functionality, too.

To hide kernel checkpointer related semantics of callback management, the common kernel checkpointer API also provides an interface to applications for de/registering callbacks.

4.4 Communication Channels

In order to take consistent distributed snapshots, job-unit dependencies must be addressed, too. The latter occur, if processes of disjunctive job-units exchange messages. Saving communication channel content is required only in terms of coordinated checkpointing for reliable channels. In-transit messages of unreliable channels can be lost anyway.

Most kernel checkpointers do not support checkpointing communication channels or use different approaches. Therefore, XtreemGCP executes a common channel snapshot protocol that is integrated into the individual checkpointing sequence of each kernel checkpointer without modifying it.

XtreemGCP drains reliable communication-channels before taking a job-unit snapshot similiar to [15]. Therefore, we intercept *send* and *receive* socket functions. When the checkpointing callback function is executed the draining protocol is executed by creating a new draining thread after other functions optionally defined by the application programmer have finished. Of course we cannot just stop application threads as messages in transit need to be received first. In order to calculate how many bytes are in transit we count the number of sent and received bytes for each application thread on each grid node. This data can be accessed using the distributed AEM communication infrastructure. The draining thread sets a warning variable causing the interceptor to buffer all outgoing messages of involved threads when calling *send*. All these messages will not be passed to the network. Threads that need to receive in-transit data are allowed to run until they have received all in-transit messages. In the latter case a thread is stopped. Depending on the communication pattern it may take some time to drain all communication channels but by buffering outgoing data the protocol will eventually stop.

During restart all buffered messages need to be injected again. Flushing these buffers is done when restart callback functions are exectuted.

5 Grid Related Issues

5.1 Job Submission

In XtreemOS, the resource requirements of a job are described using the job submission description language (JSDL). We have extended the JSDL file of a job by checkpointing requirements which are defined in a separate XML file referenced. This extension allows the user/administrator to control checkpointing

for specific applications, e.g. the best suited checkpointing strategy can be listed including strategy parameters such as checkpoint frequency or number of checkpoints to be kept. Furthermore, it includes the selection of an appropriate kernel checkpointer that is capable of consistently checkpointing application resources e.g. multi-threaded processes, that are listed in the XML file. The discovery of a suitable kernel checkpointer is integrated in the resource discovery of AEM.

5.2 Checkpoint File Management

A job checkpoint is made up of job-unit checkpoint images and grid meta-data describing the checkpoint. Storing *checkpoint image files* can be costly due to their size and thus resource consumption. *Garbage collection* (GC) is essential to use disk space efficiently. The JCP implements the GC deleting checkpoint files not needed anymore. When checkpointing is used to migrate or suspend a job, job-unit checkpoint images are removed immediately after the job has been restarted. In these cases, the disk space needed to store the checkpoints is provided by the system. The system allows the user to define how many checkpoints to keep. However, the user has to provide the necessary disk space. If his quota is exhausted old checkpoints will be deleted. Candidates need to be calculated by the GC respecting constraints by incremental images and/or uncoordinated checkpointing (here a set of consistent checkpoints needs to be computed by the GC in order to detect unnecessary ones). *Grid-checkpointing meta-data* describe job and job-unit related data and include: the kernel checkpointers used, the checkpoint version number, the process list, the process group reference id, the process group reference type etc. The JCP relies on this meta-data during restart, e.g. for re-allocating resources valid before checkpointing, using the appropriate kernel checkpointers and for correctly addressing checkpoint images. In addition this meta-data is also useful as a statistical input for the adaptive checkpoint policy management.

5.3 Resource Conflicts

Identifiers of resources, e.g. process IDs, being valid before checkpointing must be available after restart to seamlessly proceed with application execution. Therefore, identifiers are saved in the meta-data and their availability is checked during restart by the responsible translation library. If an ID is not available a job unit must be migrated to another node.

To avoid these resource conflicts during restart we are currently integrating a light-weight virtualization (namespaces, e.g. applied to process IDs) and cgroups in order to isolate resources belonging to different applications. Now the kernel is able to distinguish equally named identifiers used by different jobs.

5.4 Security

Only authorized users should be able to checkpoint and restart a job. XtreemGCP relies on the XtreemOS security services to authenticate and authorize users [1].

Checkpoint/restart can introduce some issues regarding security because things may have changed between checkpoint and restart. For example, a library that was used by the job, and that was possibly checkpointed with the job, may have been updated to correct a security hole. The job owner may have lost some capabilities and is not allowed anymore to access some resources the job was using before restart, e.g. files. We cannot detail all possible cases here, but the general rule XtreemGCP follows is that transparency comes after security. Thus a job restart can fail because of security reasons.

6 Evaluation

In this section we discuss a preliminary evaluation of our architecture including the duration of a grid checkpoint/restart action and characteristic sequence calls across all services and layers of the XtreemGCP architecture. Here we do not evaluate scalability regarding number of nodes; this will of course depend on the application and the checkpointing strategy used.

The testbed nodes have Intel Core 2 Duo E6850 CPUs (3 GHz) with 2 GB DDR2-RAM and can be booted with either LinuxSSI 0.9.3 or a Linux kernel 2.6.20. For testing grid checkpoint/restart in connection with BLCR v0.8.0, one node loads the Linux image and executes AEM. For the LinuxSSI tests we use two nodes loading the LinuxSSI 0.9.3 image, one of them executing AEM.

Each checkpoint/restart sequence is splitted into several phases. The phases are executed within a new process owned by the appropriate XtreemOS user instead of root, thus, executing with correct users access rights. Each phase is controlled in Java at the grid-level using the Java Native Interface (JNI) to access the translation library and the kernel checkpointers.

The time values (in milliseconds) of each checkpoint/restart phase are shown in Table 1. These are average values taken from ten checkpoints/restarts measurements. The test application is a single-process application allocating five mega byte of memory space and sequentially writes a random value at each byte address. Such a simple application is sufficient to evaluate the overhead introduced by our architecture.

The native BLCR checkpointing and restart functionality is split into the appropriate XtreemGCP phases by controlling the kernel part of BLCR from the user-mode translation library using message queues.

Table 1. Duration of grid checkpointing and restart in milliseconds

Checkpoint					
	prepare	stop	checkpoint	resume	total
LinuxSSI (v0.9.3)	495,8	13,9	69,6	11,0	590,3
BLCR (v0.8.0)	381,2	40,1	250,5	5,3	677,1
Restart					
			rebuild	resume	total
LinuxSSI (v0.9.3)			2597,7	12,6	2610,3
BLCR (v0.8.0)			1659,3	5,7	1665,0

The **prepare** phase includes user authentication, determination of the process group reference id and type, saving the extracted meta-data to disk and the setup of phase control structures.

Within the **stop** phase ids passed by AEM will be filtered to remove non-job processes, e.g. the XtreemOS security process. The process group reference id is updated in case of intermediate changes.

The **checkpoint** call includes saving the process image to disk. This is the work done by native system-speficic checkpointers. The time consumed here depends on the application behaviour, e.g. memory and file usage.

The **checkpoint resume** and **restart resume** phase resumes each process to proceed with execution. The first includes again process filtering for BLCR and LinuxSSI whereas the latter does not. Therefore, checkpoint resume is slower for both cases. The restart resume for LinuxSSI is faster than the one for BLCR because it does not require message-queue-based communication. All cases also include the release of the phase control structures.

The **rebuild** phase includes reading in meta-data, the setup of phase control structures, authentication and rebuilding of system-dependent kernel structures.

The observed overhead caused by XtreemGCP architecture is in the range of 0.5 - 2.6 seconds but independent of the number of nodes (it is a local overhead, only). Due to limited space we cannot discuss all implementation details causing minor time differences. Anyway, the times measured are rather small compared with the times needed to checkpoint large grid applications. For the latter the checkpoint phase dominates and may take minutes or even hours. Future work optimizations include a more fine-grained control of contents that really need to be saved and those which can be skipped (e.g. certain library files).

7 Related Work

XtreemGCP is the first implementation being able to checkpoint and restart distributed and parallel grid applications using hetereogenous kernel checkpointers.

The CoreGRID grid checkpointing architecture (GCA) [9] proposes a centralized solution to enable the use of low-level checkpointers. The *Grid Checkpoint Service* is the central component of the service that manages checkpoints in the grid and interacts with other grid services. The CGA implementation prefers the use of Virtual Machines (VMs) instead of kernel checkpointers. Overall it remains open to which extent CGA supports different kernel checkpointers and it is also currently not supporting checkpointing distributed applications.

The Open Grid Forum GridCPR Working Group targets application-level checkpointing [16][2]. For the latter they have described use-cases and requirements regarding APIs and related services. A service architecture comparable to the XtreemGCP architecture was deduced with a generic API partially included in SAGA. This work is hard to compare with our common kernel checkpointer API since it is limited to application-level checkpointing. But we plan to synchronize our common kernel checkpointer API with the one designed by OGF.

Adaptive checkpointing is important within dynamic grids and has been evaluted for parallel processing systems over peer-to-peer networks [12].

XtreemGCP callbacks and channel flushing mechanisms are inspired by previous work done by LAM/MPI [15]. The Open MPI Checkpoint/Restart framework [7] is very similar to XtreemGCP but limited to the MPI context. Moreover, since they do not especially target grids, they do not deal with grid specific issues.

8 Conclusion and Future Work

In this paper we have described the design and implementation of the XtreemGCP service used for job migration and fault tolerance. By introducing a common kernel checkpointer API, we are able to integrate and access different existing checkpointers in a uniform way. The proposed architecture is open to support different checkpointing strategies that may be adapted according to evolving failure situations or changing application behaviour.

We have also discussed challenges related to the implementation of the common kernel checkpointer API. We have described how to bridge different process group management techniques and how to save communication channels in a heteregenous setup. Furthermore, we have introduced a callback mechanism for generic system-related checkpointing tasks and also for programmer-assisted optimizations. Other grid-related issues, e.g. checkpoint file management, security, and resource conflicts have been discussed, too.

The current prototype supports checkpointing and restarting jobs using BCLR and LinuxSSI checkpointers. Preliminary measurements show only minimal overhead introduced by the XtreemGCP architecture. XtreemOS source code is available at [1].

Future work includes optimizations, adaptive checkpointing, more measurements, and integration of the announced Linux kernel checkpointer.

References

1. `http://www.xtreemos.eu`
2. Badia, R., Hood, R., Kielmann, T., Merzky, A., Morin, C., Pickles, S., Sgaravatto, M., Stodghill, P., Stone, N., Yeom, H.: Use-Cases and Requirements for Grid Checkpoint and Recovery. Technical Report GFD-I.92, Open Grid Forum (2004)
3. Bhattiprolu, S., Biederman, E.W., Hallyn, S., Lezcano, D.: Virtual servers and checkpoint/restart in mainstream linux. SIGOPS Operting Systems Review 42(5), 104–113 (2008)
4. Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F.: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, USA (2006)
5. Foster, I., Kesselman, C.: The globus project: a status report. Future Generation Computer Systems 15(5-6), 607–621 (1999)

6. Hupfeld, F., Cortes, T., Kolbeck, B., Focht, E., Hess, M., Malo, J., Marti, J., Stender, J., Cesario, E.: Xtreemfs - a case for object-based file systems in grids. Concurrency and Computation: Practice and Experience 20(8) (2008)
7. Hursey, J., Squyres, J.M., Mattox, T.I., Lumsdaine, A.: The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In: Int'l Parallel and Distributed Processing Symposium, Long Beach, CA, USA (2007)
8. Duell, J., Hargrove, P., Roman, E.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941 (2003)
9. Jankowski, G., Januszewski, R., Mikolajczak, R., Stroinski, M., Kovacs, J., Kertesz, A.: Grid checkpointing architecture - integration of low-level checkpointing capabilites with grid. Technical Report TR-0075, CoreGRID, May 22 (2007)
10. Litzkow, M., Solomon, M.: The evolution of condor checkpointing. In: Mobility: processes, computers, and agents, pp. 163–164 (1999)
11. Mehnert-Spahn, J., Schoettner, M., Morin, C.: Checkpoint process groups in a grid environment. In: Int'l Conference on Parallel and Distributed Computing, Applications and Technologies, Dunedin, New Zealand (December 2008)
12. Ni, L., Harwood, A.: An adaptive checkpointing scheme for peer-to-peer based volunteer computing work flows. ArXiv e-prints (November 2007)
13. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent Checkpointing under Unix. In: Proceedings of USENIX Winter 1995 Technical Conference, New Orleans, Louisiana, USA, January 1995, pp. 213–224 (1995)
14. Reiser, H.P., Kapitza, R., Domaschka, J., Hauck, F.J.: Fault-tolerant replication based on fragmented objects. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 256–271. Springer, Heidelberg (2006)
15. Sankaran, S., Squyres, J.M., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. International Journal of High Performance Computing Applications 19(4), 479–493 (2005)
16. Stone, N., Simmel, D., Kielmann, T., Merzky, A.: An Architecture for Grid Checkpoint and Recovery Services. Technical Report GFD-I.93, OGF (2007)
17. Vallée, G., Lottiaux, R., Rilling, L., Berthou, J.-Y., Dutka-Malhen, I., Morin, C.: A Case for Single System Image Cluster Operating Systems: the Kerrighed Approach. Parallel Processing Letters 13(2) (June 2003)