Independent Checkpointing in a Heterogeneous Grid Environment

Eugen Feller*, John Mehnert-Spahn†, Michael Schoettner†, Christine Morin*
*INRIA Centre Rennes - Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
{Eugen.Feller, Christine.Morin}@inria.fr
†Institut für Informatik
Heinrich-Heine-Universität Düsseldorf, Universitätsstr. 1, 40255 Düsseldorf, Germany
{John.Mehnert-Spahn, Michael.Schoettner}@uni-duesseldorf.de

***Abstract*—The EU-funded XtreemOS project implements an open-source grid operating system based on Linux. In order to provide fault tolerance and migration for grid applications, it integrates a distributed grid-checkpointing service called XtreemGCP. This service is designed to support different checkpointing protocols and to address the underlying grid-node checkpointers (e.g. BLCR, LinuxSSI, OpenVZ, etc.) in a transparent manner through a uniform interface. In this paper, we present the integration of an independent checkpointing and rollback-recovery protocol into the XtreemGCP. The solution we propose is not checkpointer bound and thus can be transparently used on top of any grid-node checkpointer.**

To evaluate the prototype we run it within a heterogeneous environment composed of single-PC nodes and a Single System Image (SSI) cluster. The experimental results demonstrate the capability of the XtreemGCP service to integrate different checkpointing protocols and independently checkpoint a distributed application within a heterogeneous grid environment. Moreover, the performance evaluation also shows that our solution outperforms the existing coordinated checkpointing protocol in terms of scalability.

***Keywords*-fault tolerance, independent checkpointing, rollback-recovery, heterogeneity, distributed systems, grid computing.**

I. Introduction

With the growing demand for computing power, grids have become very popular for managing large amounts of resources during the last decade. Grids provide a large-scale distributed platform for the execution of various kinds of computation and data intensive applications (e.g. weather prediction and climate simulation, particle system simulation, etc.) across many heterogeneous resources. Hence, they are vulnerable to all sorts of failures, either caused by hardware, communication or programming errors. Given a long-running scientific application, it is obvious that fault-tolerance becomes a major concern. Fault tolerance techniques based on checkpoints can be used to save the state of these applications on stable storage in order to recover them in the event of failure. Thus, upon failure the execution can be resumed from the last consistent checkpoint while preserving already done computations. Moreover, checkpointing and recovery are essential in the process of dynamic scheduling in order to migrate applications across multiple resources and thereby help optimizing global resource usage.

A lot of work has been done during the last three decades on developing fault-tolerance solutions. That work can be classified into two categories: non-distributed and distributed. The most prominent examples for non-distributed solutions include BLCR [1], Condor [2], libCkpt [3] and OpenVZ [4]. These approaches focus on single node and either save the local state of a process or encapsulate the processes within a container (e.g. OpenVZ) and save the latter entirely. While most of the complexity of those solutions relies on the proper extraction of local resources (e.g. process information, files, signals, message queues, shared memory, etc.) and their recovery, additional complexity is introduced when considering the distributed case.

In distributed systems where parts of the application communicate across multiple resources it is not sufficient to solely rely on the individual checkpoints of the participating processes in order to compute a consistent global state of the application. Here, in-transit messages need to be handled, otherwise orphan-messages and lost-messages can lead to inconsistent checkpoints. Orphan-messages are messages whose receive events are part of the destination process checkpoint but the corresponding send events are lost. In case of a recovery the destination process would receive those messages twice, which could result in unpredictable application behavior. On the other hand, lost-messages occur when the send events are part of the sender-side checkpoint, however the receiving events are lost. As a consequence, protocols and checkpointers are needed in order to address the problem of in-transit messages. In the two common checkpoint protocols, the processes either *coordinate* their checkpoints or take them *independently* [5]. Some examples for distributed cluster-level checkpointers include BLCR+MPI [6], which supports both protocols within MPI and DMTCP [7] which integrates the coordinated checkpointing protocol.

Regardless of the checkpointing solution it is obviously not realistic for each grid node to use one specific checkpointer. Therefore, in order to achieve fault tolerance for distributed applications in grids, XtreemOS [8] deploys a heterogeneous grid checkpointing service called XtreemGCP [9]. This service is designed to support different checkpointing protocols and address the underlying grid-node checkpointers

in a transparent manner through a uniform interface. Despite the generic design of this service, it currently only implements the coordinated checkpointing protocol [10]. Given a grid environment where data and computing resources of one application can be distributed across thousands of grid nodes, coordinated checkpointing can be costly in terms of scalability [10]. Furthermore, the entire application needs to be rolled back in the event of a single process failure. Thereby, additional overhead is added to the communication and storage infrastructure.

Therefore, we decided to design and implement a solution for independent checkpointing within XtreemGCP. In this paper, we present all the necessary steps towards this goal. This involves transparent dependency tracking among processes, transparent selection of the underlying grid-node checkpointer, monitoring of process failures, computing a consistent global state out of the recorded dependency information and recovery of an application. Our solution is not bound to a specific checkpointer and can be transparently used on top of any existing grid-node checkpointer.

The result of this work is an extended XtreemGCP service with the support of two checkpointing protocols, which is able to checkpoint an application distributed across different grid-node checkpointers. This is the first work dealing with integration aspects of independent checkpointing in grids. We believe that our results can help future works to identify the issues related to implementing checkpointing protocols in large heterogeneous environments such as grids.

This paper is organized as follows. Section II outlines the architecture and the components of the XtreemGCP service. Section III details the theoretical foundations of our work. Section IV presents the integration of independent checkpointing within XtreemGCP. Section V describes the results from the performance evaluation of our prototype. Section VI discusses the related work. Finally, Section VII closes the paper with conclusions and future work.

II. XtreemOS Grid-Checkpointing Service

In the following section we briefly describe the architecture of the XtreemGCP service and its components. In particular, we first introduce the terminology used in the context of XtreemGCP and then detail the relevant components where we integrate independent checkpointing.

A. Terminology

We use the term *job* for any kind of applications (e.g. parallel, distributed, etc.) running in the grid. Each job is identified by a unique global job identifier and consists of one or multiple *job units*, whereas each job-unit may itself comprise one or multiple processes running on one grid-node. The processes of a job-unit are addressed by a unique process group identifier.

B. Architecture

The architecture of the XtreemOS grid checkpointing service is depicted in Figure 1.

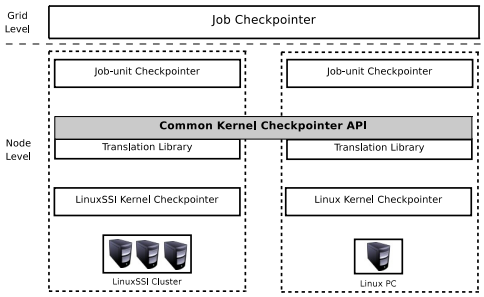

Fig. 1. XtreemOS Grid-Checkpointing Architecture [9]

At the grid level, a job checkpointer service is in charge of managing the checkpoint/restart of a job, possibly spanning multiple grid nodes. It is located on the same node as the job manager and uses virtual nodes [11] to achieve service replication and thus high-availability. The job checkpointer is designed to implement different checkpointing protocols and control the underlying node-level components (i.e. job-unit checkpointer) in order to issue checkpoints and restarts of the corresponding job-units. Moreover, with its global view of a job, it has the knowledge about all of its job-units and their locations. Hence, it is capable of taking checkpoint decisions either issued by grid users or the grid scheduler. When the coordinated checkpointing protocol is used, it coordinates the job-unit checkpointers in order to take a globally consistent checkpoint and restart the job-units in the event of failure. Furthermore, job meta information is created by the job-checkpointer. This information is required during the recovery and contains the resource requirements of the job, information about the job units (e.g. job-unit ID), location of the job-units checkpoint images and other job related details.

Each grid node has a job-unit checkpointer. It is addressed by the job checkpointer during the checkpoint coordination phase and is in charge of taking checkpoints of job-units running on a grid-node. This is done by selecting the appropriate translation library (e.g. BLCR, LinuxSSI, etc.) and addressing its routines by the use of a uniform access interface, referred as "Common Kernel Checkpointer API" [9]. This interface abstracts the different kernel checkpointer specific calling semantics and provides a uniform way to access the kernel checkpointers in a transparent manner to the job-unit checkpointer.

A dedicated translation library implements this API for each kernel checkpointer and provides a mapping of the grid semantics to the node-level semantics (i.e. grid job to local process group). It is dynamically loaded by the job-unit checkpointer during the checkpointing operation and implements all the necessary routines to address the underlying kernel checkpointer. In addition, some kernel checkpointers require extra parameters to be set, such as *permissions flags* before being able to initiate a checkpoint. Therefore, additional routines are available within the translation library. Currently, two translation libraries exist in order to support the kernel checkpointers of the XtreemOS PC and cluster flavor. These

checkpointers provide the low-level functionality necessary to take a snapshot of the entire process group, including process memory and other process related resources. Furthermore, they are used during the recovery to restart the process group from a given snapshot.

III. Theoretical foundations of independent checkpointing

In this section, we first give a brief introduction into the concepts of independent checkpointing. Afterwards, we detail our system model and then define how we record the dependencies among the job-units. Finally, we show how this information is used in order to compute a consistent global state of the application.

A. Overview

Unlike coordinated checkpointing where processes coordinate their checkpoints to take a consistent global checkpoint, independent checkpointing allows the processes to initiate checkpoints independently. Therefore, it requires that all the non-deterministic events (i.e. reception of messages) are logged with checkpoints on stable storage in order to be able to detect and resolve inter-process dependencies during the recovery.

There exist two independent checkpointing approaches: with full and with partial message-logging. The latter solution is solely based on checkpoints and the recorded event information. Therefore, it is vulnerable to the so called *domino effect* [5]. Depending on the application communication pattern this effect can enforce the entire application to roll back to its initial state, losing the entire computation. On the other hand, independent checkpointing with full message-logging is based on the piecewise deterministic (PWD) [12] assumption which requires to record all the data necessary in order to replay the events in case of failure. It is therefore not affected by the domino effect but introduces additional complexity to record and replay the data in the same order as it was initially received.

In this paper we focus on independent checkpointing with partial message-logging.

B. System model

Our system model considers exclusively jobs whose job-units communicate by exchanging messages. Further, we assume a reliable communication protocol (i.e. TCP), which delivers the messages in the First-In-First-Out (FIFO) order. The job-units can interact either using a client-server or a peer-to-peer networking model. At the beginning of the execution each job-unit halts, takes an initial checkpoint and continues to run. In the event of failure, job-units stop according to the fail-stop model [13]. Failures which occur during the checkpoint operation are not tolerated as they may introduce a non-deterministic behavior in the kernel checkpointer. Each job-unit has access to a fault-tolerant distributed storage (i.e. XtreemFS [14]) in order to store the dependency information and checkpoints. This information can be transparently accessed by XtreemGCP during the job-unit recovery.

C. Recording the dependency information

Independent checkpointing requires the knowledge of all inter job-unit dependencies in order to compute a consistent global state in the event of failure. Therefore, it is necessary to identify which information needs to be recorded and how to record it. In our system the dependency recording is inspired by the work done in [15] and works as depicted in Figure 2. Thereby, we use arrows to represent messages and circles to illustrate checkpoints.

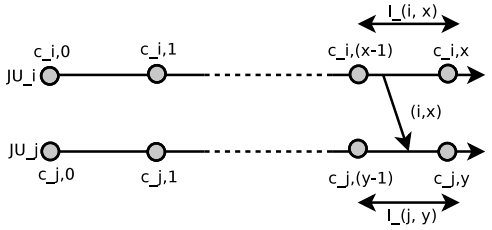

Fig. 2. Dependency information recording

We define $c_{i,x}$ as the $x-th$ checkpoint $(x \geq 0)$ of the job-unit JU_i $(0 \leq i \leq N-1)$ and denote x as the checkpoint-index and N as the number of job-units. Two checkpoints $c_{i,x}$ and $c_{j,y}$ are considered as inconsistent if either $x < y$ and a message was sent after $c_{i,x}$ and received before $c_{j,y}$ or $x = y$ and a message was sent before $c_{j,y}$ and received after $c_{i,x}$. Further, we define $I_{i,x}$ as the checkpoint-interval between two consecutive checkpoints $c_{i,x-1}$ and $c_{i,x}$. When a job-unit JU_i sends a message to JU_j within the interval $I_{i,x}$, the pair (i, x) is attached. JU_j receives the message in the interval $I_{j,y}$ and records a dependency between $c_{i,x-1}$ and $c_{i,y}$. Thereby, one message received by job-unit j from a job-unit i within the same interval $I_{j,y}$ is enough in order to establish a dependency.

D. Recovery line computation

In order to compute a consistent global state we first need to acquire all the recorded dependency information and then generate a set of consistent checkpoints, also referred to as the recovery line. In this section we use a concrete example to illustrate the process. Figure 3 shows a job composed of three job-units which communicate and record the dependency information according to the previously introduced definitions.

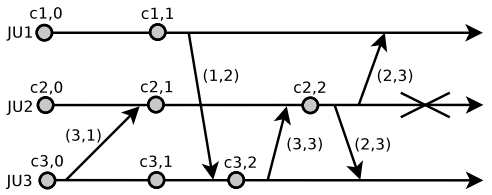

Fig. 3. Example job with three job-units

Shortly after issuing the third checkpoint $c_{2,2}$, job-unit 2 fails and the recovery line computation is started by generating a checkpoint graph and applying the rollback-propagation algorithm [15] on it.

Figure 4 depicts the checkpoint graph and the final recovery line for our example job. Here we use dashed circles to represent the current state of the remaining job-units.

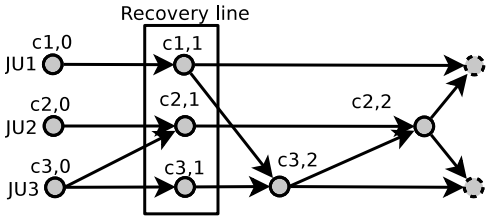

Fig. 4. Checkpoint graph

The generation of the checkpoint graph works as follows. We draw a directed edge from $c_{i,x-1}$ to $c_{j,y}$ if and only if $i \neq j$ and a message was send during $I_{i,x}$ and received in $I_{j,y}$, or $i = j$ and $y = x+1$. Afterwards, we use the rollback-propagation algorithm to compute the recovery line. Therefore, we first create a set of checkpoints and include the *dependency information* associated with the last checkpoint from the failed job-units as an element into the set. In addition, we include the *dependency information* associated with the current state of the other still running job-units as an element into the set. We then sequentially examine the *dependency information* of the elements in the set and *mark* all elements which are reachable by following the edges of all element in the set. Afterwards, we iterate over the *marked* elements and *replace each marked element* with the previous checkpoint of the same job-unit. In the next step the *dependency information* of the elements in the set is examined again and all the elements are marked which are reachable by following the edges of all element in the set. The two latter steps are repeated until there are no marked elements left in the set. The result is a set of checkpoints which represents the consistent global state (i.e. recovery line).

IV. Integration of independent checkpointing in XtreemGCP

The following section presents the design decisions and integration aspects of independent checkpointing in XtreemGCP. Therefore, we first detail how the dependency information can be recorded transparently to the application in a heterogeneous environment. Afterwards, we describe the process of independent checkpointing, which involves the interaction between the job-units and the corresponding job-unit checkpointer. Finally, we detail the recovery process and introduce the supported networking models and recovery scenarios of the proposed solution.

A. Transparent dependency tracking

During job execution, all the non-deterministic events (i.e. reception of messages) need to be recorded in the distributed storage (i.e. XtreemFS [14]) for a potential recovery line calculation (see Section III). Therefore, we have developed a library which is dynamically loaded by each job-unit using the library interposition [16] mechanism available on Linux systems during job submission. This mechanism works by initializing the *LD_PRELOAD* environment variable with the desired shared library. Once submitted, all the job-units are dynamically linked against this library, and all communication-related system calls are redirected. In the used library we currently redirect all the *send()* and *receive()* requests in order to attach/extract the dependency information transparently and save it on distributed storage (i.e. XtreemFS [14]). Furthermore, we use this library to transparently increment the version number of the checkpoint and assign a tag to the dependency information during the checkpoint. In particular, this is done by intercepting the *fork()* call and transparently registering a callback using the callback mechanism provided by XtreemGCP. This callback is then executed during the checkpoint. Figure 5 shows the interaction between the job-units, our library and the distributed storage.

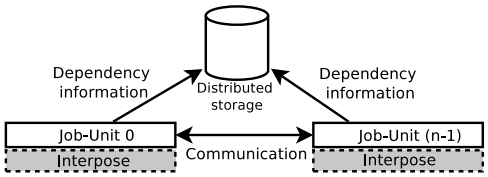

Fig. 5. Transparent dependency tracking

B. Application checkpoint

The ultimate goal of independent checkpointing is to provide support for the job-units to take checkpoints independently. Therefore, we have developed another shared library which is statically linked to each job-unit providing an interface to the programmer in order to trigger checkpoints out of the application at any point in time. Thereby, the main task of this library is to communicate with the job-unit checkpointer and request it to take a checkpoint. Moreover, it communicates with the interpose library introduced before and instructs it to increase the version number of the checkpoint and save the dependency information on distributed storage during a checkpoint.

We have modified the job-unit checkpointer accordingly to handle the communication, initiate the checkpoints on demand and update the job meta information in case when the checkpoint was successful. The procedure of independent checkpointing is depicted in Figure 6. When a checkpoint is triggered by the job-unit, a blocking checkpoint request is sent to the job-unit checkpointer. The job-unit checkpointer processes the request, detects the underlying kernel checkpointer of the grid-node and calls the translation library to initiate a checkpoint. Thereby, potentially already existing checkpoints need to be removed before a new one can be created. This happens always when some of the job-units fail and need to be recovered from one of the previous checkpoints. Thus, they will start triggering checkpoints whose images already exist on the distributed storage. Therefore, we have modified the translation libraries to integrate the cleanup functionality before initiating the checkpoint.

In case of a successful cleanup and checkpoint, the job-unit checkpointer updates the job meta information on the distributed storage and sends back a reply to the job-unit. After the reception of the reply the job-unit continues its execution.

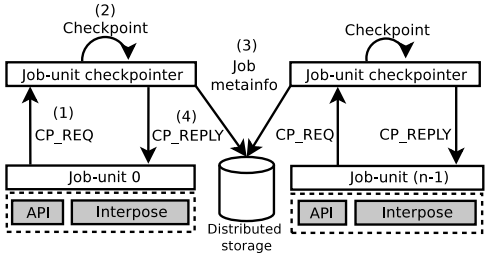

Fig. 6. Independent checkpointing in XtreemGCP

In addition to independent checkpointing, job-checkpointer can be configured to initiate checkpoints periodically. Last but not least, checkpointing is triggered when a migration request is issued either by the user or the XtreemOS grid scheduler itself.

C. Application recovery

The following section deals with the recovery of the job. We have integrated all the necessary components such as failure monitoring support and recovery line computation (see Section III) into the XtreemGCP service. Now, we detail the networking models and recovery use cases we have identified. Furthermore, we describe the integration aspects.

1) Supported networking models: The job-checkpointer is in charge of managing the recovery of a job and its job-units. Therefore, it detects the job-units based on the recorded job meta information and executes the appropriate checkpointing protocol to perform the recovery. However, no matter which protocol is used, the recovery process is asynchronous. Thus, job-units are restarted in parallel by the corresponding job-unit checkpointers. Thereby, some job-units are restarted faster than others due to different resource requirements (i.e. cpu, memory, etc.) and try to deliver messages to job-units which potentially were not restarted yet.

We distinguish between two networking models: client-server and peer-to-peer. In the former model each job-unit operates exclusively either as client or as server. Therefore, the recovery process is straight forward. The job-checkpointer first restores the server job-units and then the client job-units, avoiding the possible conflicts introduced through parallelization. In the latter model each job-unit operates as client and server simultaneously. Thus, the job-checkpointer first computes a consistent global state and then initiates the recovery of the job-units without taking into account the recovery order. Afterwards, we use the dependency tracking library to interpose the *connect()* call and send a connection request message consisting of the IP address and the port of the target job-unit to the *job-unit checkpointer*, which then forwards the message to the assigned job-checkpointer. The job-checkpointer verifies if the target job-unit has already been restored and sends back a reply message to the source job-unit. During this period the source job-unit remains blocked and continues its execution in case of a positive feedback. This way we can make sure that the target job-unit has always been restored before others can send messages to it.

2) Permanent application failure: In the previous section we have introduced the two networking models supported by our architecture. Now, we detail the first recovery scenario which assumes a permanent application failure. In such a case, the job checkpointer computes a consistent global state based on the last taken checkpoint of each job-unit and restores the job-units afterwards. The recovery itself can be either initiated by the user or done automatically by the system upon failure detection. Figure 7 illustrates this workflow.

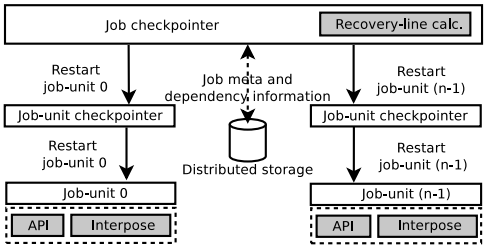

Fig. 7. Application recovery - Permanent failure

First, the job checkpointer identifies the job-units belonging to the job based on the recorded job meta information. Then, the dependency information is loaded and the consistent global state of the application is generated using the rollback-propagation algorithm introduced in Section III. Finally, the job-checkpointer initiates the recovery of the job-units using one of the two previously introduced networking models. Therefore, it addresses the job-unit checkpointers in order to select the appropriate kernel checkpointer and restart the job-units. Thereby, sockets need to be recreated first as most of the available kernel checkpointers (e.g. BLCR, LinuxSSI, etc.) nowadays do not support the saving and restoring of sockets. In fact, they are ignored during the checkpoint. Hence, sockets are not part of the checkpoint and need to be recreated during the restart, otherwise job-units will fail to resume their execution. We use the XtreemGCP callback mechanism to register a restart callback and recreate them within this callback before starting the job-units execution.

3) Partial application failure: The second recovery use case supported by our work deals with the most common grid environment scenario where some job-units fail while others keep running. In order to support the recovery of the failed job-units we have extended the job-checkpointer by a failure monitoring module. This module keeps track of the failed job-units and is required during the recovery. Figure 8 depicts our modifications and the workflow of the failure monitoring. First, the job-unit checkpointer detects a failure based on the *exit-code* of the job-unit and informs the corresponding job-checkpointer. This information is then stored within the job-checkpointer failure monitoring module for later usage.

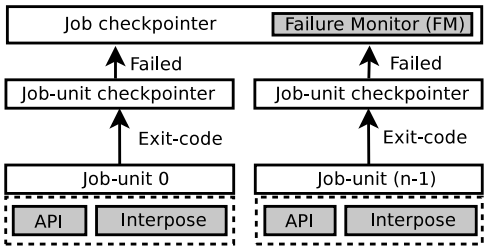

Fig. 8. Application recovery - Failure monitor

Once the failure has been detected, recovery can be initiated. Therefore, failure monitoring information is extracted, a checkpoint graph is generated and the recovery line computation algorithm is run. Depending on the resulting recovery line, one or multiple still alive job-units need to be rolled back to some previously taken checkpoint due to existing dependencies. In this particular case the job-checkpointer requests the status of the currently running job-units from the failure monitoring module. In case the job-units are still running they are stopped and the recovery from the set of checkpoints given by the recovery line is started analogous to the first case. After a successful recovery the job-unit status needs to be updated inside the job-checkpointer and its failure monitoring module. Thus, the job-unit checkpointers send an update message to the job-checkpointer upon successful recovery.

V. Performance Evaluation

A. Experimental setup

To evaluate the implementation, we have configured a heterogeneous environment composed of six AMD Opteron 244 1.8 GHz nodes, running Debian GNU/Linux 5.0 ("lenny"). Each node is equipped with 1GB RAM, 1GB swap and a gigabit network card. We have chosen the first four nodes to operate in a single-PC mode and installed a modified version of BLCR (v0.8.2) for XtreemOS on them. Moreover, 2.6.20.20 vanilla-kernel is used. The remaining two nodes are running the latest LinuxSSI kernel (v1.0-rc2) and thus form an SSI cluster. LinuxSSI is a Single System Image (SSI) cluster operating system based on Kerrighed [17] which serves as the basis for the XtreemOS cluster flavor. In all experiments we use the Networking File System (NFS) to provide storage for the checkpoints, dependency and the job meta information to all the nodes.

Currently, our prototype implementation supports independent checkpointing and recovery for distributed applications using the first networking model, discussed in Section IV-C1. We have installed our extended version of the XtreemGCP service on all nodes and measured the time to record the dependencies, to checkpoint the application and to perform the recovery.

B. Synthetic distributed client-server application

We have developed a synthetic distributed client-server application and executed it on top of our heterogeneous environment consisting of single-PC nodes and a SSI cluster. Thereby, we start our server job-unit on one of the single-PC nodes and distribute the remaining five client job-units to the other nodes. Each time a job-unit starts it triggers an independent checkpoint and continues its execution. Furthermore, in order to simulate random independent checkpoints each client job-unit defines two numbers $x \geq 0$ and $y > x$, randomly selects a number in the interval $[0, y]$ and continues its execution. Afterwards, the random number is compared with x. In case of a match independent checkpoint is triggered, else a message (2 Bytes) is sent to the server. In our experiments we have preassigned x to 4 and y to 6. Similarly, the server job-unit simulates whether it will take a checkpoint or not.

In addition to random checkpointing, failures need to be simulated. This can be either simulated by an external *kill* event or by the job-unit itself using the *exit* call. We use the latter approach and terminate the job-unit in case of a match of the two numbers.

In the following we present our experimental results using this application. The results are based on 10 measurements for each test series and the computation of the arithmetic mean.

C. Dependency tracking

The dependency tracking mechanism intercepts the *send* and *receive* calls of the job-units in order to attach and extract the dependency information, used during the recovery. Hence, we have measured the overhead introduced by this procedure. The result from our evaluation is that this overhead is under 1μs and thus negligible. We explain this with the way how we manage the interception process. In order to attach a dependency information, we do not need to copy the message into a buffer. In fact, we split the *send and receive* sequences and first send the dependency information, followed by the unmodified message. As the dependency information is small, there is only a minimal additional overhead.

D. Application checkpoint

Figure 9 shows the times of independent checkpointing for a single job-unit running on top of a single-PC node and the LinuxSSI cluster.

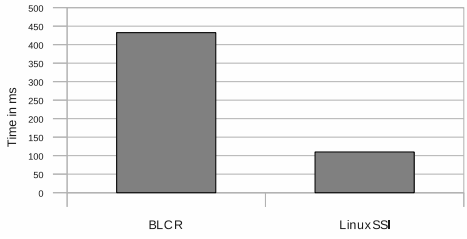

Fig. 9. Time to checkpoint a single job-unit

We can note that independent checkpointing on top of BLCR is slower approximately by a factor of 4 than with LinuxSSI. Particularly, independent checkpointing on BLCR takes *430 ms* and LinuxSSI *110 ms*. We explain this with the integration of BLCR within the XtreemGCP service. In LinuxSSI the kernel-checkpointer routines can be accessed

directly by the use of *ioctl()* calls, whereas checkpointing with BLCR needs to be synchronized with semaphores, message queues and local files. Therefore, the checkpointing process is slower. However, the time to checkpoint the job-units is still significantly lower than when using the coordinated checkpointing protocol (see Section V-F).

E. Application recovery

Figure 10 shows the time needed to compute the recovery line for a job composed of one, three and six job-units respectively. The time taken to compute a recovery line for one job-unit is approximately *26.4 ms*. We explain this time with the need to load the dependency information from the storage, as no recovery line computation is done in this case. When increasing the job-unit count to three and six, the time increases to *71 ms* and *131 ms* respectively. Hence, the time to compute the recovery line is proportional to the number of job-units. We think that the reason for that is twofold. With tripling the number of job-units, the amount of dependency information to load is tripled too. Moreover, our application did not show the strong domino effect which could potentially increase the time to compute the recovery line for applications of higher complexity (i.e. dependencies).

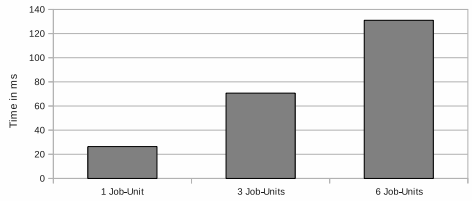

Fig. 10. Time to compute the recovery line

Figure 11 indicates the time to restart the application. This time includes the time to compute the recovery line and the time to restore the job-units. Here, the time to restart one, three and six job-unit takes approximately *7.3*, *7.7* and *8.1* seconds. We explain this good scalability with the parallel restart of job-units in XtreemGCP. Obviously, bigger job-unit data sizes in conjunction with the previously mentioned domino effect could result in larger restart times.

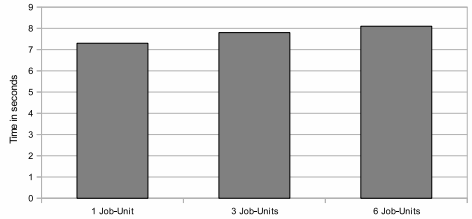

Fig. 11. Time to restart the application

F. Performance of coordinated and independent checkpointing

In [10] we have evaluated the coordinated checkpointing protocol using a similar client-server application within a heterogeneous environment consisting of single PC-nodes and an SSI cluster. Therefore, we have measured the time it takes to checkpoint this application when it opens up 50 communications channels and sends 100 Byte packets periodically every five seconds. The result from this evaluation was that it takes approximately *4.25* seconds to synchronize the processes, buffer the messages and close the sockets. In addition, *2.12* seconds more are spent in order to reestablish the connections, unblock the communication and deliver the possible in-transit messages after the checkpoint has been taken. Consequently, in total approximately *6.36* seconds were needed to checkpoint this application. Furthermore, when it was not necessary to close and reestablish the connections, improved checkpointing duration of *3.37* seconds was measured.

Nevertheless, the results show that even if it is sometimes possible to keep the connections open during the checkpoint, additional global synchronization overhead still increases the checkpoint duration approximately by the factor of 8 compared to the usage of the independent checkpointing protocol. Our previous measurements (see Section V-D) confirm this with checkpointing times of just *430 ms* and *110 ms* for LinuxSSI and BLCR respectively. Hence, we believe that especially in the context of grids, independent checkpointing is a more scalable solution.

VI. Related Work

A lot of research has been done in order to provide fault tolerance for distributed applications during the last two decades. This has led to the development of many checkpointing protocols (e.g. [18], [12], [19], etc.) and a few cluster level software frameworks with fault tolerance support. CoCheck [20] supports the coordinated checkpoint/restart protocol using the Condor checkpointer. Further, LAM/MPI [21] provides coordinated checkpointing support for MPI applications using either the BLCR or a "self" checkpointer. In the latter case it is the responsibility of the application to perform the checkpoint/restart functionality.

The most related work on supporting independent checkpointing can be found in the implementation of Starfish [22] and MPICH-V2 [23]. Starfish, implements coordinated and independent checkpointing for MPI applications. MPICH-V2 integrates independent checkpointing with full message-logging using the Condor checkpointer. Moreover, in [24] the authors have implemented independent checkpointing with full message-logging and evaluated their implementation within a homogeneous environment consisting of traditional workstations. However, our solution differs from all these approaches as it is not limited to the MPI context and implements independent checkpointing in a transparent manner within a *heterogeneous grid environment* across different checkpointers (e.g. BLCR, Condor, LinuxSSI, OpenVZ, etc.).

In terms of XtreemGCP architecture similar work can be found in the CoreGRID grid checkpointer [25]. However, it is currently limited to the use of Virtual Machines (VMs) and does not provide any support for checkpointing distributed applications. Even though virtualization helps preventing resource conflicts during recovery, coordination is still necessary

for jobs spanning multiple grid nodes. In contrast, XtreemGCP supports either coordinated or independent transparent checkpoint/restart of jobs those job-units are running on multiple heterogeneous grid nodes.

VII. Conclusions and Future Work

To the best of our knowledge, this work is the first demonstration of the design and implementation of independent checkpointing within a heterogeneous grid environment. Although we did not propose any new checkpointing protocol, we have detailed the design and implementation aspects of independent checkpointing in grids. In particular, we have detailed how to transparently record the dependency information, detect failures within a distributed application, compute a consistent global state and finally how to recover a distributed application using the independent checkpointing protocol within a heterogeneous environment. Our work has resulted in an extended version of the XtreemOS grid checkpointer which now supports the independent checkpointing protocol. Moreover, we have evaluated our implementation within a heterogeneous environment consisting of traditional PC-like nodes running BLCR and a Single System Image (SSI) cluster.

The results show that independent checkpointing is more suited for grid environments than coordinated checkpointing. In fact, coordinating a checkpoint in a grid environment where jobs can span across hundreds of nodes and be riddled with communication links is often inefficient and can not scale. Nevertheless, our implementation has some limitations which we plan to address in the future. The first limitation regards the networking model we currently support. This model supports job-units which either act as a client or a server. We have proposed a solution for supporting the alternative peer-to-peer networking model (see Section IV-C1) which is scheduled for future implementation. Further, independent checkpointing with partial message-logging is vulnerable to the domino-effect. Therefore, we plan to extend our implementation with full message-logging support. Another limitation regards the recovery process. Currently, the user is in charge of initiating the recovery. We think about enabling the XtreemGCP service to automatically initiate the recovery in the event of failure.

Finally, besides improving the current implementation we plan to evaluate independent checkpointing in conjunction with the existing support for incremental checkpointing [26].

References

[1] J. Duell, "The design and implementation of berkeley labs linux checkpoint/restart," Berkeley Lab, Berkeley, CA, USA, Tech. Rep., December 2002.

[2] M. Litzkow and M. Solomon, "The evolution of condor checkpointing." New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 163–164.

[3] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," Knoxville, TN, USA, Tech. Rep., 1994.

[4] K. K. Andrey Mirkin, Alexey Kuznetsov, "Containers checkpointing and live migration," in *Linux Symposium 2008*, Jul. 2008, p. 101.

[5] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[6] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "Mpich-v: toward a scalable fault tolerant mpi for volatile nodes," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–18.

[7] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[8] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. P. Prieto, and A. Reinefeld, "XtreemOS: a Vision for a Grid Operating System," May 2008, white Paper Available at http://www.xtreemos.eu/publications/research-papers/xtreemos-cacm.pdf.

[9] J. Mehnert-Spahn, T. Ropars, M. Schoettner, and C. Morin, "The architecture of the xtreemos grid checkpointing service," in *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 429–441.

[10] J. Mehnert-Spahn and M. Schoettner, "Checkpointing and migration of communication channels in heterogeneous grid environments," in *The 10th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, Springer, Ed., Busan, Korea, May 2010.

[11] H. P. Reiser, R. Kapitza, J. Domaschka, and F. J. Hauck, "Fault-tolerant replication based on fragmented objects," in *In Proceedings of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, 2006, pp. 14–16.

[12] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 3, pp. 204–226, 1985.

[13] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," 1983.

[14] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "The xtreemfs architecture—a case for object-based file systems in grids," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 17, pp. 2049–2060, 2008.

[15] Y. min Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *in Proc. IEEE Symp. Reliable Distributed Syst*, 1993, pp. 78–85.

[16] B. A. Kuperman and E. Spafford, "Generation of application level audit data via library interposition," Tech. Rep., 1999.

[17] D. Margery, G. Vallee, R. Lottiaux, C. Morin, and J. yves Berthou, "Kerrighed: A ssi cluster os running openmp," in *In Proc. 5th European Workshop on OpenMP*, 2003.

[18] T. Ropars and C. Morin, "Fault tolerance in cluster federations with o2p-cf," in *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 807–812.

[19] Q. Jiang, Y. Luo, and D. Manivannan, "An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 12, pp. 1575–1589, 2008.

[20] G. Stellner, "Cocheck: Checkpointing and process migration for mpi," in *in Proceedings of the 10th International Parallel Processing Symposium*, 1996, pp. 526–531.

[21] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," in *in Proceedings, LACSI Symposium, Sante Fe*, 2003, pp. 479–493.

[22] A. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations," *High-Performance Distributed Computing, International Symposium on*, vol. 0, p. 31, 1999.

[23] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 25.

[24] P. Sens and B. Folliot, "The star fault manager for distributed operating environments. design, implementation and performance," *Softw. Pract. Exper.*, vol. 28, no. 10, pp. 1079–1099, 1998.

[25] A. Ciuffoletti, A. Congiusta, G. Jankowski, M. Jankowski, N. Meyer, and O. Krajicek, "Grid infrastructure architecture: a modular approach from coregrid," CoreGRID Project, Tech. Rep. TR-0089, August 2007.

[26] J. Mehnert-Spahn, E. Feller, and M. Schoettner, "Incremental checkpointing for grids," in *Linux Symposium 2009*, Jul. 2009, p. 201.