Checkpointing and Migration of Communication Channels in Heterogeneous Grid Environments

John Mehnert-Spahn[1] and Michael Schoettner[1]

Heinrich-Heine University, Duesseldorf, NRW 40225 , Germany,
John.Mehnert-Spahn@uni-duesseldorf.de
Michael.Schoettner@uni-duesseldorf.de

Abstract. A grid checkpointing service providing migration and transparent fault tolerance is important for distributed and parallel applications executed in heterogeneous grids. In this paper we address the challenges of checkpointing and migrating communication channels of grid applications executed on nodes equipped with different checkpointer packages. We present a solution that is transparent for the applications and the underlying checkpointers. It also allows using single node checkpointers for distributed applications. The measurement numbers show only a small overhead especially with respect to large grid-applications where checkpointing may consume many minutes.

1 Introduction

Fault tolerance can be achieved for many distributed and parallel applications, particularly for scientific ones, using a rollback-recovery strategy [1]. Here, programs are periodically halted to take a checkpoint that can be used to restart the application in the event of a failure. As nodes may fail permanently the restart implementation must support restarting checkpoints on new healthy nodes. The latter is also important for application migration, e.g. to realize load balancing. Checkpointing solutions have been available for many years, mainly used in high performance computing and batch processing systems. As grid applications are getting into mainstream there is an emerging need for a transparent checkpointing solution. Right now there is a great variety of different checkpointing packages such as single node checkpointers, e.g. BLCR (with MPI support) [8], OpenVZ [9]. Virtual machine technologies, e.g. VMware [4] and XEN [7] are also used as checkpointers. Furthermore, there are distributed checkpointers such as LinuxSSI [5], DCR [12] and DMTCP [2]. All these implementations come with different capabilities and there is no ultimate best checkpointer.
Therefore, we have designed XtreemGCP - a grid checkpointing service capable of checkpointing and restarting a grid job running on nodes equipped with different checkpointing packages [13]. Each grid job consists of one or multiple job units. Each job unit represents a set of processes of a job on one grid node. Checkpointing a distributed job does not only require to take a snapshot of all process states on all involved nodes but also to handle in-transit messages as well. Otherwise, orphan messages and lost messages can lead to inconsistent

checkpoints. Orphan messages occur if the reception of a message is part of the receiver side checkpoint, however the message send event is not part of the sender side checkpoint. In case of a restart the sender will send this message again which may cause faults.
Lost messages occur if reception of a message is not part of the receiver side checkpoint, however the message send event is part of the sender side checkpoint. During restart this message will not be sent again and the receiver may block and run into a failure situation. Obviously, in-transit messages need to be handled by all involved checkpointers. As the latter have not been designed to cooperate with each other, e.g. BLCR does not cooperate with DMTCP, we need a subordinated service transparently flushing all communication channels at checkpoint time avoiding in-transit messages. And this service must also support channel re-connections in case of a job migration. The contributions of this paper are the concepts and implementation of a transparent grid channel checkpointing (GCC) facility for a heterogeneous setup with various existing checkpointer packages. We address TCP sockets used by a great range of distributed applications. As UDP communication inherently tolerates lost messages we do not take care of in-transit UDP messages.
The outline of this paper is as follows. In Section 2 we present an overview of grid channel checkpointing followed by Section 3 describing the architecture in detail. The different phases to checkpoint, restart and migrate channels are discussed in Section 4 followed by an evaluation. Related work is discussed in Section 6 followed by conclusions and future work.
XtreemGCP is implemented within XtreemOS - a Linux-based distributed OS for next generation grids [3]. This work is funded by the European Commision under FP6 (FP6-033576).

2 Grid Channel Checkpointing Overview

The main idea of the GCC approach is to flush all TCP channels of a distributed application before a checkpoint operation in order to avoid lost and orphan messages. Flushing of TCP channels is achieved by preventing all application threads from sending and receiving messages as well as from creating new channels during the checkpointing operation.
Concurrent GCC protocol execution is achieved by using separate threads for channel control and flushing. Once, appropriate controller threads have been installed at both peer processes of an application TCP channel, a marker message is sent through the channel, signaling the marker receiver that the channel is empty. Potential in-transit messages received by a controller thread in the context of channel flushing are stored in a so-called channel draining buffer at the receiver side.
No messages can get lost since all in-transit messages will be assigned to the current checkpoint on the receiver side. No orphan message can occur since sending of application messages is blocked until the checkpoint operation has finished and all received messages have been recognized as being sent on the sender side.

To support checkpointers incapable of saving/restoring socket descriptors, sockets may need to be closed before a checkpoint operation and recreated after a taken checkpoint. A recent version of LinuxSSI failed during restart, since sockets that were open during checkpointing time could not be reconstructed. But of course, whenever possible we leave sockets up and running for performance reasons.
Before an application resumes with its normal execution, potentially closed sockets will be recreated and reconnected. Furthermore, any blocked channel creation system calls and formerly blocked send and recv calls will be released. Messages formerly stored in the channel draining buffer get consumed by application threads before new messages can be received.
Finally, GCC also needs to handle changed IP-addresses caused by job-unit migration. To handle changing IP-addresses we introduce a GCC manager component that can be contacted by all involved GCC controller instances.

3 GCC Architecture

In the following text we describe the Grid Channel Checkpointing (GCC) architecture and its components in detail. Fig. 1 presents all GCC components.
GCC marks TCP sockets because UDP sockets are not relevant. It also determines the current socket usage mode, indicating whether being in receive or in send mode. The socket usage mode may change dynamically as TCP sockets can be used in a bidirectional fashion. If a process sends and receives messages over one TCP socket it is called to be in duplex mode.
For application-transparent channel checkpointing network system calls such as send, recv, connect, accept, listen, etc. must be controlled. Application threads must neither send or receive messages nor create new channels while checkpointing is in progress. However, GCC threads must be able to exchange GCC-messages on application channels (to be flushed) and on GCC control channels. The library interposition technique is used, to achieve these features in an application-transparent way. Therefore, we initialize the environment variable LD_PRELOAD with the path to our interposition library. Thus, all network calls of the application end up in the interposition library. The contained network function wrappers pass the calls to the original library. Therewith we are able, e.g. to block a send call within the associated function wrapper.
As previously mentioned, sockets may have to be re-connected to new IP addresses in case of process migration. Sockets must also be closed and recreated on nodes whose checkpointers are incapable of handling open socket descriptors. Both tasks can be handled using the callback mechanism provided by XtreemGCP [13]. The latter explicitly executes registered callbacks before and after a checkpoint or after a restart.
In addition these callbacks are used to integrate the channel flushing protocol. GCC callbacks are registered in an application-transparent way, realized by a fork-wrapper included in the above mentioned interposition library.
Two control threads are created per process, the send-controller thread and the

recv-controller thread, see Fig. 1. If a socket is shared by multiple processes on one node, the send- or recv-controller thread of one process becomes channel leader of it. The channel leader exclusively supervises channel flushing and reconnection, but relies on cooperation with the remaining controller threads. More

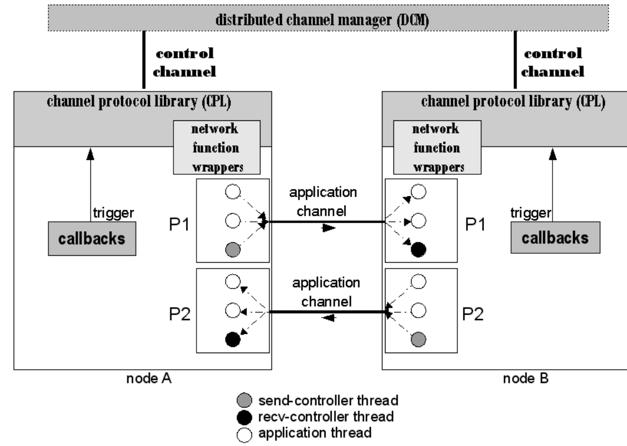

Fig. 1. GCC components

precisely, the send-controller takes control over sockets of TCP channels being in send mode. At checkpoint and restart time it is in active mode which means it initiates flushing and reconnection with the remote recv-controller. The recv-controller takes control over TCP channels being in receive mode. It is in passive mode and reacts on flushing and reconnection request from a send-controller.
Distinguishing send- and recv-controllers allows handling situations where two communicating processes are server and client for different channels at the same time. No deadlock can occur during restart e.g. if one server socket on each side, recreated by the relevant send-controller, waits for the opposite node to reconnect, since the relevant recv-controller is ready to handle a reconnection request.
Migrated sockets must be detected before a controller thread tries to initiate a socket reconnection. Thus, each socket creation, recreation and any other socket state changes are registered at the distributed channel manager (DCM). Therefore, all controller threads can use a TCP control channel to the channel manager. Currently, we have a central channel manager implementation (one manager for each grid job) that will be replaced by a distributed version for scalability reasons.
GCC execution is triggered by callbacks triggered before and after checkpointing and immediately after a restart.

3.1 Shared sockets

As mentioned before a socket formerly shared by multiple processes must be recreated with special care, just one process must recreate it. Thus, in a multi-process application the channel leader exclusively recreates the socket in the kernel within the post-checkpoint or restart callback. For the remaining application processes to see the newly recreated socket we use the UNIX descriptor passing mechanism [10]. Using the latter allows the channel leader to pass recreated socket descriptors to controller threads of other processes using UNIX domain socket connections, see Fig. 3.1.
The channel manager assigns a unique key per channel to involved controller threads for a UNIX domain socket connection setup and descriptor exchange. The key remains the same also after a potential migration. While the descriptor is sent, a corresponding entry is made in the process-owned descriptor table. Process socket descriptors assigned to a logical channel must be matched with those being valid before checkpointing. Socket descriptors are assigned in an in-

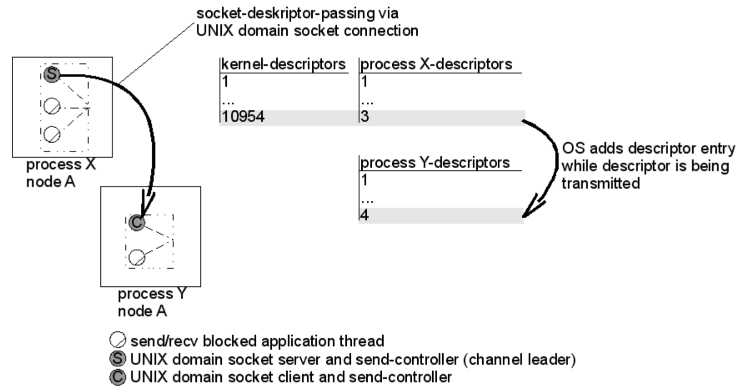

Fig. 2. Process-wide shared soclets and descriptor passing

creasing order during normal runtime. If an intermediate descriptor gets closed, a gap exists, and the highest descriptor number is bigger than the total number of sockets currently used. During socket rebuild, descriptors will be assigned in an increasing number, without gaps by taking the association of channel and descriptor number into account saved during pre-checkpoint time and descriptors will be rearranged to the correct order using the dup2 system call.
Thus, this approach avoids false multiple recreations of a shared socket and the latter do not need to be reestablished exclusively via process forking and inheritance in user space. Additionally kernel checkpointer based process recreation, which excludes the calling of fork at user space, is supported as well.

4 GCC Phases

4.1 Pre-Checkpoint Phase

Blocking channel creation: Since there is a short time gap between pre-checkpoint-callback execution and process synchronization in the checkpoint phase, the creation of new TCP channels must be blocked. This is realized by blocking the socket calls accept and connect until the checkpoint operation is finished.
Determining channel leaders: Sending on TCP channels must be prevented to drain channels in finite time. Two challenges need to be addressed in this context. First, if multiple processes use a shared socket, each of them may send a marker, many could receive one or multiple markers. Obviously, it is unclear when the last message has been received. Second, application threads could be blocking on send and recv calls which would prevent the protocol to start.
This first challenge is addressed by determining a so-called channel leader whose task is to exclusively treat sockets shared by multiple processes. More precisely, one send controller will be installed per process and out-going channel, one recv controller thread will be installed per process and in-coming channel. Each controller sends a channel-leader-request to the channel manager. Thus, e.g. a two-process application, whose child inherited socket descriptors of its parent, sends two requests to the channel manager. The channel manager selects one of the requesting controller threads as the channel leader and informs all about this decision.
The second challenge is solved by the controller threads sending a SIGALRM signal to applications threads blocking on send or recv calls. Application and controller threads can be distinguished such that just application threads will be put asleep in the signal handler routine while controller threads can start with channel draining. Finally, no application thread is able to send or receive messages along TCP channels anymore until the end of the post-checkpoint phase, see Section 4.2.
Before channel flushing can be initiated by a send controller, a recv controller must contact the DCM to learn on which socket it is supposed to listen for the marker. Both use a handshake to agree on which channel is to be flushed.
Channel flushing: The send controller being the channel leader at the same time sends a marker. The marker is the last message received by the remote recv controller which is the channel leader on the peer node. Messages received before the marker are in-transit messages. Marker and in-transit content will be separated. The latter will be put in a channel draining buffer. The buffer data is assigned to the appropriate receiving application thread. Usually, just one thread waits for messages on a peer. However, multiple threads can do so as well. In the latter case the OS scheduler decides non-deterministically which thread receives the message. Thus, the channel checkpointing protocol copies received data into the receive buffer of the application thread that has been listening to this channel recently. This equals a possible state during fault-free execution.
Marker recognition can be achieved at different levels. Currently, a special marker message is sent along the application channel, see Fig. 4.1. Since the underlying

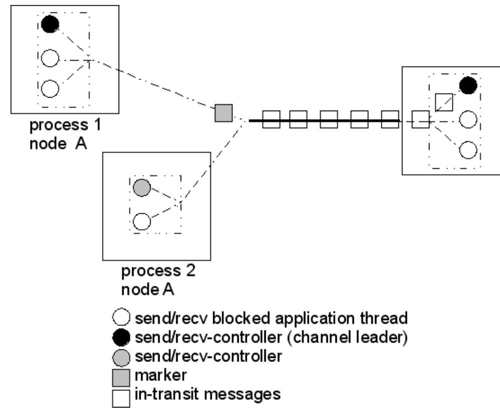

Fig. 3. Channel flushing with marker message

TCP layer fragments application data independent of application semantics a marker message can overlap with two or multiple TCP packages. Thus, at each byte reception the recently received and buffered data must be matched backwards against the marker. In the future we plan to replace this first approach by another alternative, e.g. extending each application network packet by a GCC header which is removed at receiver side or using a socket shutdown to enforce the sending of a TCP FIN message to be detected in the kernel.
Furthermore, we optionally need to close open sockets for checkpointer packages that cannot handle them. This is no problem for the application threads as they are blocked and sockets will be re-opened in the post-checkpoint phase, see Section 4.2. The final step of the pre-checkpoint phase is saving the association of socket descriptor and channel key, needed during post-checkpoint and restart time.

4.2 Post-Checkpoint Phase

Unblocking channel creation: This GCC phase aims at unblocking the network communication just after a checkpoint has been taken. At first, channel creation blocking is released, unblocking system calls such as connect and accept.
Recreating (shared) sockets: Sockets need to be recreated only if they had been closed before checkpointing or in case of a restart. There is no need to adapt server socket addresses, since no migration took place.
Socket recreation becomes more complex if sockets are shared by multiple processes, see Section 4.1.
Release send/recv barriers: The last step of this GCC phase is to unblock formerly blocked send and recv calls and to wake up application threads. Furthermore, any buffered in-transit messages need to be consumed before any new messages.

4.3 Migration and Restart Phase

The GCC restart is similar to the post-checkpoint phase but both differ from the location of execution and migration-specific requirements. The first step here includes the release of the channel creation blockade (unblocking connect and accept calls). Furthermore, we need to address different checkpointer (CP) capabilities:

1. CPs capable of saving, restoring and reconnecting sockets (for changed IP addresses),
2. CPs capable of saving and restoring sockets, but not supporting socket migration,
3. CPs being unable to handle sockets at all.

Obviously, we need to address cases 2 and 3, only. Here sockets are recreated, their descriptors are rearranged as described under Section 3.1. Before a client socket can reconnect to a server socket, we must check if a migration took place recently and if the server is already listening for reconnections. The latter is the task of the channel manager which receives any changed server addresses from the control threads, see Section 3. Thus, a client just queries the DCM to learn the reconnection state.
The last step in this phase includes releasing any formerly blocked send and recv calls and waking up application threads.

5 Evaluation

The GCC pre- and post-checkpoint phases have been measured using a synthetic distributed client server application running on nodes with heterogeneous and homogeneous checkpointer packages installed. The test application sends periodically 100 Byte packets in five second intervals. At client and server side each channel is handled by a separate thread.
In the first test case the server part is executed and checkpointed on one node part of a LinuxSSI cluster (v2.1), the client part on grid node with BLCR (v0.8.2) installed. The channel manager is executed on a separate node inside the LinuxSSI cluster. In the second test case client and server have been executed and checkpointed on nodes with BLCR installed.
The testbed consists of nodes with Intel Core 2 DUO E6850 processors (3 GHz) with 2 GB RAM interconnected with a Gigabit Ethernet network.

5.1 Test Case 1: Heterogeneous checkpointers and GCC

Fig. 4 indicates the times taken at a client and a server for flushing, closing and reestablishing channels on top of LinuxSSI and BLCR checkpointers (no shared sockets have been used). The pre-checkpoint phase takes up to 4.25 seconds to

handle 50 channels. The duration is mainly caused by the serial synchronization of the send and recv controller threads via the channel manager. An improved channel manager is on the way handling requests concurrently. Furthermore, the duration includes memory buffering and consumption of potential in-transit messages.
Fig. 4 also shows the time needed for the post-checkpoint phase taking about half of the time as the pre-checkpoint phase. This is due to less interaction with the channel manager and of course no channels need to be flushed. As expected if necessary, rebuilding and reconnecting of sockets is costly.
Fig. 5 indicates the times for the same scenario as shown in Fig. 4 but without closing and reestablishing channels. Here the pre-checkpoint phase takes less time (about 3.25 seconds to handle 50 channels). Furthermore, without socket rebuilding and reconnecting this post-checkpoint phase is also significantly shorter than the one from above just taking about 120 milliseconds for 50 channels.
Another aspect is that GCC is working on top of heterogeneous callback implementations without major performance drawbacks. While BLCR comes with its own callback implementation implicitly blocking applications threads, LinuxSSI does not. For the latter we have to use the generic callback implementation provided by XtreemGCP.

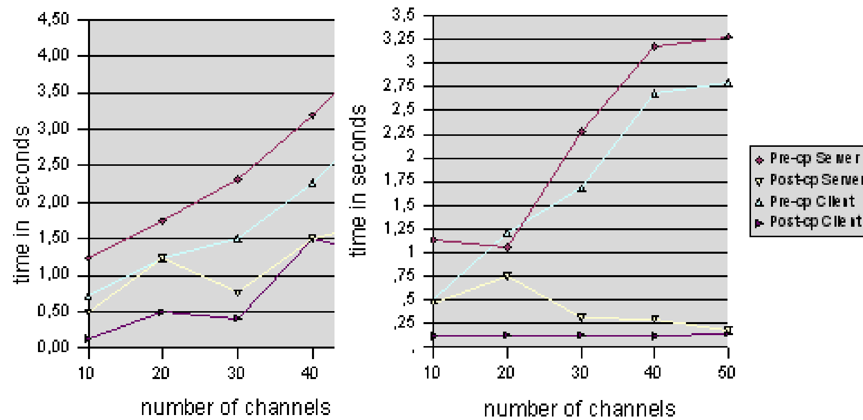

Fig. 4. GCC behavior on top of LinuxSSI and BLCR with closing and reestablishing channels

Fig. 5. Same scenario as in Fig. 5 but without closing and reestablishing channels

5.2 Test Case 2: Homogeneous checkpointers and GCC

Fig. 6 indicates the times taken for the client and server to flush, close and reestablish channels based on the mere usage of BLCR checkpointers. The pre-checkpoint phase takes up to 2.75 seconds to handle 50 channels. It is faster than the pre-checkpoint phase of test case 1 because both grid nodes run native Linux there is no SSI-related overhead caused by LinuxSSI. The post-checkpoint phase is significantly shorter than the one of test case 1 because no native SSI structures must be updated when sockets are being recreated and reconnected.
Fig. 7 indicates the times taken for a client and server to flush open channels. It takes less time during the pre- and post-checkpoint phase compared to the previous setup of test case 2.
Overall we see that the current implementation of GCC consumes more time when checkpointing more channel connections per grid node. Although we do not expect thousands of connections per grid node as the typical case we plan to optimize the GCC solution to be more scalable.
Another aspect is that the amount of messages in-transit to be drained during checkpointing operation will also influence GCC times. But as the bandwidth of grid networks is typical several Mbit/s or even more, we do not expect an extensive time overhead here. Furthermore, though checkpointing communication channels may consume several seconds we think this is acceptable because checkpointing large grid applications may take many minutes to save the application state to disk.

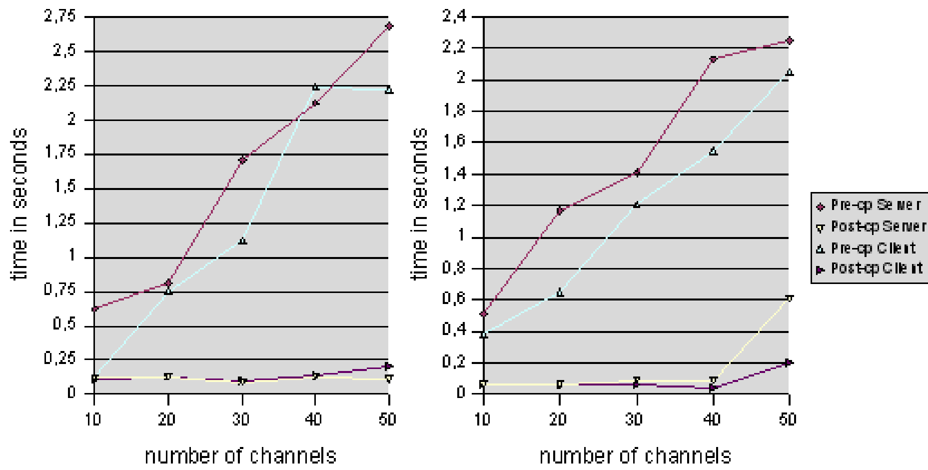

Fig. 6. GCC with BLCR only, with closing and reestablishing channels

Fig. 7. Same scenario as in Fig. 6 but without closing and reestablished channels

6 Related work

Overall there is only one other project working on checkpointing in heterogeneous grid environments while there are different projects implementing checkpointing for MPI applications [11]. However, there are many publications proposing sophisticated checkpointing protocols but that are not related to heterogeneity challenges addressed by this paper.
The CoreGRID grid checkpointing architecture (GCA) [14] proposes a similar architecture like XtreemGCP aiming to integrate low-level checkpointers. However, the current GCA implementation supports Virtual Machines (VMs), only, and does not support checkpointing communication channels of distributed applications.
The Open Grid Forum (OGF) GridCPR Working Group has published a design document for application-level checkpointing [6] that is not addressing transparent channel checkpointing.
DMTCP [2] is most close to the approach proposed in this paper. It is a distributed library checkpointer able to checkpoint and migrate communication channels. They also use a marker message to flush in-transit messages but the latter will be sent back to the original sender at checkpoint time and forth to the receiver at resume/restart time. In contrast we store in-transmit messages at the receiver side. Furthermore, DMTCP supports only one specific checkpointer whereas our approach is designed for heterogeneous grid environments. Finally, shared sockets are recreated during restart by a root process in user space which inherits them to children processes created later. In contrast to our approach processes with disturbed original parent-child relations cannot be recreated.
In [12] communication states are saved by modifying the kernel TCP protocol stack of the OS. The approach is MPI specific, does not support shared sockets, and is designed for one checkpointer (BLCR) and not for a heterogeneous setup.

7 Conclusions and future work

Transparent checkpointing and restarting distributed applications requires handling in-transit messages of communication channels. The approach we propose flushes communication channels at checkpoint time avoiding orphan and lost messages. The contribution of this paper is not a new checkpointing protocol but concepts and implementation aspects how to achieve channel flushing in a heterogeneous grid where nodes have different checkpointer packages installed.
The proposed solution is transparent for applications and existing checkpointer packages. It also allows to use single node checkpointers for distributed applications without modifications because GCC takes care of checkpointing communication channels.
GCC is a user mode implementation not requiring kernel modifications. It also offers transparent migration of communication channels and supports recreation of sockets shared by multiple threads of one or more processes. Our measurements show that the current implementation can handle dozens of connections

in reasonable time, especially with respect to checkpointing times of huge applications which can be many minutes.
Future goals include implementation optimizations to improve scalability and channel flushing support for asynchronous sockets.

References

1. E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
2. 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.
3. http://www.xtreemos.eu
4. J. Sugarman and G. Venkitachalam and B.H. Lim. Virtualizing I/O devices on VMWare Workstations Hosted Virtual machine Monitor 2001
5. Matthieu Fertre and Christine Morin. Extending a cluster ssi os for transparently checkpointing message-passing parallel application. In *ISPAN*, pages 364–369, 2005.
6. N. Stone, D. Simmel, T. Kilemann and A. Merzky. An architecture for grid checkpoint and recovery services. Technical report, 2007.
7. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, J. Pratt and A. Warfiled XEN and the art of virtualization 2003
8. J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. 2003.
9. http://download.openvz.org/doc/openvz-intro.pdf
10. W.R. Stevens, B. Fenner, and A.M. Rudoff. *UNIX Network programming The Sockets Networking API*, volume I. Addison-Wesley, 2004.
11. S. Sankaran, J.M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove and E. Roman The LAM/MPI checkpoint/restart framework: System-intiated checkpointing International Journal of High Performance Computing Applications, 19(4), 2005
12. C. Ma, Z. Huo, J. Cai, and D. Meng. Dcr: A fully transparent checkpoint/restart framework for distributed systems. 2009.
13. John Mehnert-Spahn, Thomas Ropars, Michael Schoettner and Christine Morin The Architecture of the XtreemOS Grid Checkpointing Service Euro-Par, Delft, The Netherlands, 2009
14. G. Jankowski, R. Januszewski, R. Mikolajczak, M. Stroinski, J. Kovacs, and A. Kertesz. Grid checkpointing architecture - integration of low-level checkpointing capabilities with grid. Technical Report TR-0036, CoreGRID, May 22, 2007.