Integrated Process Management in a Grid Checkpointing Environment

John Mehnert-Spahn*, Michael Schoettner†, Christine Morin‡
*†Department of Computer Science
Heinrich-Heine University
Duesseldorf, Germany
Email: John.Mehnert-Spahn, Michael.Schoettner@uni-duesseldorf.de
‡INRIA Rennes-Bretagne Atlantique
Email: Christine.Morin@irisa.fr

***Abstract*—For many businesses, the ability to manage dynamic distributed environments has become a key success factor. Joint industry and/or academic cooperations exploit resources spawning multiple administrative domains with millions of nodes and thousands of users. In order to run the overall business effectively Grid technologies can be applied. The EU-funded XtreemOS project implements a grid operating system transparently exploiting resources of virtual organizations through the standard POSIX interface. The application execution management uses checkpointing and restart for migration and fault tolerance. Grid checkpointing and restart requires to save and restore jobs executing in a distributed heterogenous grid environment. The latter consists of grid nodes (PCs, clusters, and mobile devices) using different system-specific checkpointers saving and restoring application and kernel data structures for all processes executing on a grid node. In this paper we shortly describe the XtreemOS grid checkpointing architecture and how we bridge the gap between the abstract grid and the system-specific checkpointers. Then we discuss how we keep track of processes and how different process grouping techniques are managed to ensure that all processes are checkpointed. Finally, we present how cgroups shortly introduced in Linux can be used to address resource isolation issues during the restart.**

I. Introduction

A grid encompasses a huge number of heterogeneous computer nodes. Computing capacity dramatically increases and is available throughout the world. However, the node failure probability increases as well. Consequently, distributed applications running on these nodes are affected and must be kept from failing.

Fault tolerance can be achieved using the Backward Error Recovery (BER) strategy. Generally an application is halted and a checkpoint is taken (covering application and system states) that is sufficient to restart the application after a failure. In coordinated checkpointing a coordinator causes all application processes to be stopped to form a consistent checkpoint. Messages in transit have to be taken into account as well. In indepedent checkpointing no coordination overhead exists, application processes can be checkpointed individually. However, at recovery a consistent checkpoint of the application has to be computed by analyzing taken checkpoints. If no such checkpoint can be determined the application will be restarted from the initial state. This is known as domino effect.

In grid checkpointing one has to pay attention to the grid topology, semantic differences on various topology levels and grid node characteristics. Saving a job that is spread over multiple grid nodes requires to save states on all involved grid nodes. The grid topology in connection with coordinated checkpointed requires to adapt the checkpoint/restart sequence into modular pieces for checkpointing: stop all job units, checkpoint all job units and resume all job units and for restart: rebuild all job units and resume all job units. The next phase of these sequences can begin just after all job units have finished the current one for consistency reasons.

Furthermore existing checkpointers that are bound to the native OS (executed on a cluster or single PC) are not aware of the grid topology and semantics of an upper logical (grid) level, the distributed grid OS. The most significant implication is that existing checkpointers can not save states that belong to a distributed grid OS.

Another major aspect is that existing checkpointers are not aware of one application part residing on another grid node then themselves. The methodology to assemble processes into one logical unit and distribute them over many grid nodes is not known to existing checkpointers. Consequently a checkpointer on one grid node is not aware of other checkpointers being involved. Thus, an upper grid checkpointer must address multiple checkpointers on multiple grid nodes. They have to be made working together, especially in terms of consistently saving states of network connections spawning multiple grid nodes, [1]. Depending on the availability of checkpointers on grid nodes, checkpointers of the same or of diverse types must be conducted. Thus, grid checkpointing must provide an infrastructure to take diverse checkpointers into account.

Snapshots of grid node states shall be taken in an application transparent manner. A grid's aim towards fault tolerance is to support unmodified and legacy applications to seamlessly switch from PCs or clusters to the grid.

Another requirement is that all process events must be known. Otherwise the grid checkpointer may fail to take a consistent checkpoint image. For example the SSi grid node type requires additional efforts such that all process events

can be tracked.

Various grid environments exist. The vast majority is middle ware related. Generally middle ware packages come up as bundle of services, distributed and non-distributed, offering functionality that merely can be realised in user space. Since the underlying operating system keeps structures that represent processes, network connections, memory etc. only access to the kernel allows saving processes at checkpoint time and rebuilding them after a failure. Thus, access to kernel checkpoint/restart functionality is requried to achieve comprehensive fault tolerance. A middle ware approach can not meet this requirement. Thus, once failed applications can only be restarted from an initial state.

The XtreemOS project (supported by the European Commision's) aims at buidling a Linux-based operating system to support virtual organizations (VOS) in next-generation grids. In XtreemOS heterogeneous grid nodes are single PC, SSI cluster and mobile device. Managing VOs on these node types requires appropriate mechanims for load balancing and fault tolerance. Both can be addressed by checkpoint and restart.
In XtreemOS each grid node provides a common set of services. These services interact with services on the same and on remote grid nodes - distributed job and resource management will be achieved. These services abstract from the underlying grid node specific characteristics. Regarding fault tolerance kernel checkpointer for SSI and singlePC grid node types will be taken into account. Interfacing kernel functionality from user space services bears challenges referring semantic differences that must be mapped.
Generally one can classify SSI specific and all grid node type specific challenges that influence consistent grid checkpointing. Both will be referred to this paper.
The paper is organised as follows: ...

II. Related Work

CoreGRID [2][3] aims at defining a high-level checkpoint/restart grid service to locate it among other grid services. In CoreGRIDs grid checkpointing architecture (GCA) this grid service indirectly addresses a so called Core Service. Bridging the high level grid service with the Core Service has been realised by a translation service that exists per Core Service. In former times existing and emerging kernel checkpointer packages and library checkpointers were used as the Core Service for saving and recovering jobs at kernel level. The kernel checkpointers checkpointability of an applications resources has been taken into account.
Resource conflicts at restart in connection with existing kernel checkpointers seem to be the reason for focussing on another approach - using Virtual Machine Managers (VMM). A VMM can provide and manage Virtual Machines (VM). A VMM dumps a VMs state that hosts a users jobs to persistent storage and is able to resume it. The GCA's Core service has been replaced by VMM. According to that per VMM there is a translation service. Within a separate Virtual Machnine per job resources that were valid before checkpointing can be assigned easily to a restarted job.
The Open Grid Forum (OGF) follows a different approach as CoreGRID. Basically, OGF's Grid Checkpoint and Recovery (GridCPR) [4],[5] defines new grid services for checkpoint and restart and a dedicated user level API. Future applications shall take this API into account to allow jobs being checkpointed and recovered in the grid. XtreemOS is member of the OGF and in deep contact with its people.
The HPC4U project [6] that aims at enabling applications to be migrated from a local to a remote cluster in case of a failure. This happens in a transparent manner by using virtualisation capabilities to be integrated in mainline Linux kernel. The so called Ressource Management System archestrates subsystems to provide requested level of fault tolerance. The checkpoint subsystem interacts with the MPI fault tolerance component (Scali MPI Connect of Scali AS) to support fault tolerance for MPI applications.
Since our approach is based on existing kernel checkpointers a survey of them will be given. Tools with diverse capabilites to checkpoint/restart software resources (process hierarchy, multiple threads, sockets, IPC, etc.) are available at user and kernel level as well as a combination of both levels (hybrid checkpointers). However, only kernel checkpointers comprehensively enable to save and rebuild kernel states and thus, transparently save and restore an application. An important requirement in the context of checkpoint/restart is to ensure all application resources such as PIDs, that were valid before checkpointing, are available at restart. Thus, process management, especially process isolation, reservation and virtualisation, become a significant part of (grid) checkpointing.
Zap [7] is a system that provides transparent migration for several types of applications. It groups processes into pods. A pods processes are decoupled from host operating dependencies by an underlying virtualization layer. Thus, pods can seamlessly be migrated, which includes checkpointing and restart, between nodes. Virtualization is achieved by using namespaces, virtual identifiers and intercepting system calls. The mapping of real to virtual identifiers and vice versa is realised via hashtables. Pods are not part of native Linux.
Cruz [8] is a distributed checkpoint-restart mechanism based on ZAP. It allows checkpointing and restarting of resources such as transient socket buffer states, socket options, TCP state and by adding migratable IP and MAC addresses.
As well as ZAP, Berkely Lab's Linux Checkpoint/Restart (BLCR) [9] is a checkpoint/restart implementation at kernel level based on loadable kernel modules. It can be used as a component to save and rebuild parallel jobs running on multiple nodes. LAM/MPI has been arranged to comply with BLCR [10]. Furthermore BLCR has been integrated into OpenMPI and SSS. BLCR does not take operating system resources virtualisation e.g. by using containers into account.
CHeckPOinting for linuX (CHPOX) [11] is implemented as a kernel module. It allows to dump linked libraries and

all process children into a checkpoit image. No process management as described above is realised.
In Ckpt [12] process checkpointing is performed in user space. It is implemented as a set of libraries and programs. Programs need not to be relinked, this tool can be injected inot running programs. No information is provided regarding process isolation and reservation.
Condors [13] checkpointing mechanisms are based on ckpt. Condor enables to checkpoint and restart on a different machine since the process state can be written into a socket, thus is independent of a distributed file system. Again, no details can be given how to deal in case of process isolation and upcoming resource conflicts when restarting.

Adaptive grid checkpointing targets to efficiently checkpoint a grid application by taking several aspects into account, e.g. a grids dynamicity, checkpointing overhead, application behaviour. The central challenge is to determine when to take what kind of checkpoint. Beside strategies that figure out a appropriate checkpointing interval such as Last Failure Dependent Checkpointing (LFDC), Mean Failure Dependent Checkpointing (MFDC) there are strategies that decide when to replicate applications and those that combine replication with checkpointing [14]. In the context of adaptive grid checkpointing one of our future research activities deals with investigating when it is best to use sequential and incremental checkpointing.

III. Grid checkpointing architecture

As described before checkpointing and restart of a job is a very important facility for grid job-unit migration and fault tolerance. The major challenges in the grid for checkpointing and restart are heterogeneity and scalability. The latter affects checkpointing strategies, e.g. coordinated or independent and how they scale with the number of nodes. For example, the coordination of thousand of distributed grid nodes may cost considerable time. Scalability is addressed by the XtreemOS operating system but is beyond the scope of this paper. We focus here on checkpointing and restart in a heterogeneous grid environment. This includes different hardware, software, and different checkpointing implementations. The latter are always customized for a specific platform as they need to save and restore kernel data structures. Furthermore, there are applications/libraries (e.g. MPI) linked against a specific checkpointing implementation with specific features. BCLR for example allows applications to register for being informed about checkpoint/restart events. Thus, applications can save/restore additional states under their own control. Obviously, a grid checkpointing architecture cannot ignore all these existing implementations and therefore we have designed an architecture bringing together existing code.
The grid checkpointing architecture is integrated within the XtreemOS Application Execution Management (AEM) [15], [16] and [17]. AEM is responsible for executing, controlling, and monitoring jobs and resources. In the grid a job, e.g. a business, scientific or interactive application, may be spread

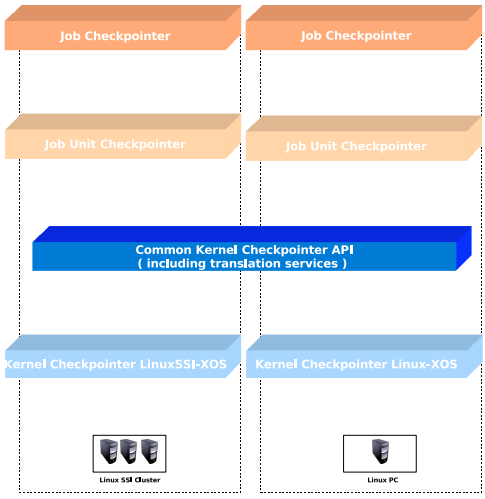

Fig. 1. Grid checkpointing architecture

over a huge number of grid nodes (PCs, clusters, and mobile devices). Checkpointing and restarting such a job requires saving/restoring the state of all processes running on all grid nodes. The basic building blocks of our grid checkpointing architecture are shown in 1.
The top-level AEM service is the distributed job checkpointer running on all grid nodes. It keeps track of all grid nodes running job units of a job. The job checkpointer extends the job manager by providing checkpointing and restart functionality for jobs. Using the job manager the job checkpointer can localize and address all job units to checkpoint and restart them. Below the job checkpointer resides the job unit checkpointer. A job unit checkpointer is a local service running within AEM. It is an extension of the execution manager which controls the underlying processes on a grid node associated with a job unit. The job unit checkpointer checkpoints and restarts processes based on the execution manager's process knowledge. The job unit checkpointer addresses one out of potentially several underlying kernel checkpointers in order to checkpoint or restart processes belonging to a job unit. A kernel checkpointer saves/restores kernel data structures and is customized for a certain operating system version and platform. To address multiple kernel checkpointers a common kernel checkpointer API together with translation services (e.g. for process grouping) is used by the job unit checkpointer. The XtreemOS implementation uses a BCLR- (for PCs) and a Kerrighed (for SSI clusters) checkpointing implementation. Below we discuss selected aspects of our components. When talking about checkpointer we always have in mind the restart functionality, too.

A. Job Checkpointer

As the job checkpointer and the job unit checkpointer work on different levels - job unit and process, scalability is improved when synchronizing processes. Each job unit checkpointer synchronizes processes of one job unit on one grid node (may be a cluster), while a job checkpointer synchro-

nizes job units of one job. This hierarchical approach reduces synchronization overhead during coordinated checkpointing and restart.

The job checkpointer is also involved during job submission and identifies suitable kernel checkpointers with specific properties if specified by the user. This is done through an own XML-based checkpointing properties file that is referenced from the job description language (JSDL) file used by AEM. All existing kernel checkpointers vary in their capabilities to save and rebuild resources such as multiple processes, multiple threads, sockets, shared segments etc. A suitable kernel checkpointer has to be identified for a given job. Otherwise a given job may not be saved or rebuilded correctly and the restart could fail. A similar approach was proposed by the CoreGRID checkpointing architecture [2], [3].

Additionally the job checkpointer performs checkpoint file management. It manages directories and files used for checkpointing. A garbage collection mechanisms controls disk space needed for checkpoint files (may be a huge amount of memory). The user may influence the garbage collector in order to keep checkpoints, e.g. for debugging purposes. Disk space required for checkpoint files may be limited for the user and may be specific before job submission. If however, AEM uses checkpointing and restart for migration the system itself has to provide the needed disk space; this will not be accounted to the user.

Since checkpoints allow to rebuild an other users job state (including all process states and kernel data structures) checkpoints contain sensitive data that need to be protected. Therefore, the job checkpointer incorporates authentication and authorisation (optionally en/decryption) mechanisms provided by XtreemOS [18].

To minimize the overhead during checkpointing or restarting an appropriate checkpointing strategy has to be selected. Coordinated checkpointing causes significant coordination overhead especially, when a lot of processes need to be coordinated [19]. On the other hand the last set of checkpoints always form a consistent checkpoint and restart. Thus, it is very fast. Independent checkpointing strategies avoid this coordination overhead but a consistent set of checkpoints need to be calculated during restart time which can be time consuming. Due to the domino effect the job can fall back to the initial state. Message logging is known to solve the domino effect allowing to recover single nodes but cause considerable overhead during fault-free execution. We plan to implement an adaptive checkpointing approach using monitoring information, e.g. fault frequency, memory write behaviours to dynamically adapt to the best checkpointing strategy. Similar approaches have been described in the literature [20]. . Currently, we are implementing the adaptive control of the checkpointing frequency that is defined either statically by the user or dynamically depending on the monitored fault frequency. Furthermore, the job/user can explicitly force checkpoints at critical points of the job. Another parameter we are currently studying is when to use incremental checkpointing. The latter comes with some cost, e.g. monitoring memory writes, book-keeping of memory page versions, and its overhead costs depend on how many pages have been modified since the last checkpoint. But often a huge file is only modified at some small points and it is much cheaper to save only these modifications. As written above adaptive grid checkpointing is on our roadmap but not fully implemented yet.

B. Job Unit Checkpointer

The job unit checkpointer provides an abstraction layer, the common kernel checkpointer API, for the job checkpointer to address several underlying, node bound, kernel checkpointers. This API features an interface for checkpoint and restart sequences for coordinated and independent checkpointing and callback management.

C. Common Kernel Checkpointer API

For each kernel checkpointer there is a specific translation library CRTransLib that implements the common kernel checkpointer API according to the semantic of one kernel checkpointer. This library has to be implemented for each kernel checkpointer that should be used within the XtreemOS grid checkpointing facility. The CRTransLib bridges AEM and kernel checkpointer specific semantics such as different process control groups techniques described in section IV and V. Furthermore, it detects job dependencies and together with our cascading synchronization checkpoints and restarts dependend jobs V.

IV. Process event tracking

Single System Image (SSI) clusters give the illusion of one single powerful grid node. Thus, it is natural to execute AEM only on one master node of such a SSI cluster in order to keep the SSI illusion. Of course, for fault tolerance reasons AEM is executed on some more cluster nodes that take over the masters' role in case the master fails. For grid checkpointing and restart it is essential to know anytime all processes belonging to a job unit executing on a SSI cluster. As a consequence AEM must be able to detect relevant process events such as process creation and process termination within the cluster.

Process event detection is realised within XtreemOS by using Linux native process event connectors (PEC) [21], a kernel-user-kernel space communication system based on netlink sockets. A user-space application can register itself for being notified whenever certain kernel events occur. For standalone PCs this is not difficult but in a SSI cluster two challenges must be addressed: process creation on slave nodes and process migration within the cluster. As AEM runs on the master node only it is not informed about process events occuring on slave nodes.

Furthermore, process events occuring during process migration must be handled carefully, depending on the migration implementation. LinuxSSI uses *do_fork* and *exit* system functions within its process migration implementation. A migration scenario is presented in figure 2. AEM is executing on the master node. Process P_a is initially forked by AEM on the

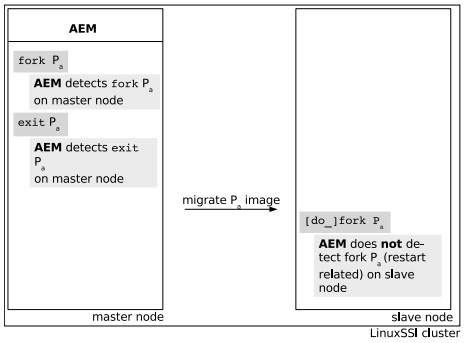

Fig. 2. SSI process migration

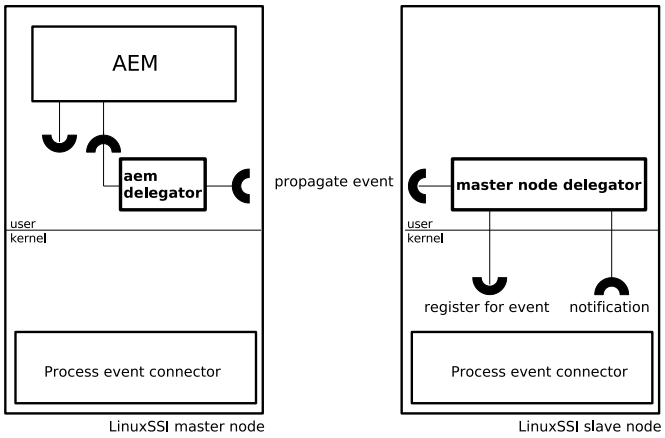

Fig. 3. User-level approach: collector and delegators

master node and is thus visible for AEM. Afterwards, the system decides to migrate P_a to the slave node within the SSI cluster. The migration module of LinuxSSI calls *exit* for P_a on the master node which will be detected by AEM. Now P_a does not longer exist for AEM. Shortly later the slave node will call *do_fork* in the Linux kernel to recreate P_a using the transferred process image [22]. Unfortunately, the process creation event on the slave node will not be detected by AEM because PEC netlink socket messages are not forwarded to the master node. Even if such events would be forwarded there would be a short time interval where P_a vanishes before appearing again. If a grid checkpointer would fire a checkpoint within this time interval this process could be missed. Obviously, for AEM only the initial *fork* and the real termination (not related to migration) is of interest with respect to checkpointing and restart. Of course for global job and resource management other events are of interest, too. Subsequently, we discuss different solutions including the one we have implemented.

LinuxSSI shares resources and meta data through the Kerrighed Distributed Data Management (KDDM). Process information is available on each cluster node by the pid KDDM set (a set hosts shared objects of the same type). Pid KDDM set objects are created on the node creating a new process. Thus, the master node could periodically poll for new objects that would be retrieved automatically over the network if new ones appear. Of course polling is a bad approach wasting CPU time and the event notification is delayed depending on the polling freqeuency. Therefore, it would be better to enforce that all pid KDDM set objects are allocated on the master node based on the centralized manager algorithm [23]. Unfortunately, this would slow-down process creation on all slave nodes and would require complex kernel modifications.

An alternative would be to modify the standard Linux connectors that inform user applications that have registered before about kernel events. Several applications may register for the same event but events cannot be distributed over the network. In order to allow remote applications to register for kernel events on other machines another communication protocol, other than netlink (PF_NETLINK), would be needed. Although this would be possible we refrain from the necessary kernel-code modifications because it is difficult to get kernel changes accepted and integrated in the Linux kernel mainstream.

Therefore, we have decided to develop a user-level approach using two client applications: *collector* (running on the master node, only) and *delegator* (running on the master and on all slave nodes), see figure 3. The delegators register at the kernel connector for process creation and termination events (and some more events not relevant for checkpointing). All these events are delegated to the collector (known by each delegator) which notifies AEM. In order to use PEC each slave node must be properly configured in advance. The delegator on each slave node sets up a netlink connection to the local kernel connector. Furthermore, it sets up a TCP connection to the collector. The IP address and port number of the collector application will be passed as arguments to each delegator. Every time the delegator receives a local connector event it propagates it via the TCP connection to the collector on the master node. The collector in turn receives the delegator message and passes it to the AEM ExecMng.

As decribed before special care is needed during migration as AEM is only interested in process creation and termination and not in intermmediate forks and exits used for migration. This is achieved by masking the migration related fork and exit events on each node.

Our implementation is able to keep track of all processes belonging to a job unit with just minimal LinuxSSI kernel modifications to mask migration related fork and exit events.

V. Process grouping

During a coordinated checkpoint or a restart (independent of the checkpointing strategy used) all processes belonging to a job need to be checkpointed or restarted. AEM references processes contained in a job unit (having parent-child relationship) by the root job-unit process. In section IV we have described how we keep track of child process creation and termination within a SSI cluster. Therewith, the root process and all child processes of the job will be checkpointed or restarted.

The root process of a job is created by AEM using the *execvp* system call. According to the execvp semantic open

file descriptors (socket relation) will not be closed. Thus, the root process of the job sees part of the AEM process context. Checkpointing or restarting the job would result in saving these descriptors and restoring them respectively. Especially, the restart could cause AEM failures. Thus, we close all file descriptors of the AEM process directly after the root process creation of the job.

A. Hetereogenous kernel checkpointers

Passing a job unit root process to the underlying kernel checkpointer to reference a job units processes however may cause undefined behaviour. Since the kernel checkpointer is unaware of the job unit semantic it may checkpoint too many or not all processes. In the worst case the root process exits and its children are reparented to init, in that case the kernel checkpointer would not save any process. One has to take into account that different kernel checkpointer may use a different process grouping concept to define the set of processes belonging together.

In native Unix there are concepts to group processes into *process groups* (identified by a process group ID) and *sessions* (identified by a session ID). A session consists of at least one process group. One process is the session leader and the leader of one process group at the same time. BLCR can use a Unix session (identified by a session identifier, SID) and a Unix process group (identified by a process group identifier, PGID) to reference one or multiple processes [24], [25], [26]. BLCR can also be applied to a process tree consisting of a process and all its non-orphaned descendants.

The LinuxSSI checkpointer [22] uses a specific frontier calculation algorithm [27] to determine a process group (referenced by application id, APPID) to be checkpointed/restarted on demand taking into account indirect process dependencies. The recursive transitive closure calculation stops each time it reaches a process that has not set the CHECKPOINTABLE cabability flag. The latter indicates if a process can be checkpointed or not.

Of course there are further state-of-the art kernel checkpointers like Cruz [8], Zap [], Condor [13] etc. that use similar or different grouping techniques. Of course future emergin kernel checkpointers may come with different process grouping concepts that do not correspond to the one used so far.

Thus, AEM cannot be hardcoded against one specific process grouping mechanism and within a grid environment we do not want to restrict our architecture to a specific kernel checkpointer. Furthermore, independent of the used process grouping scheme, AEM itself should be kept out of process groups that constitute a job unit in order to avoid interferences (e.g. like described above).

To avoid checkpointing/restarting to much or to less processes we need to bridge the gap between grid jobs and specific process grouping solutions. This translation service is located within the CRTranslib tailor-made for each specific kernel checkpointer, see section 3. In the following subsection we discuss our solution for resource isolation that solves also the problem of how to reference process groups in a abstract way.

B. Lightweight virtualization for job-unit isolation

There may be multiple job units running on a single grid node (PC, cluster, mobile device) that need to be isolated to avoid resource conflicts, especially during process restarts. If job A is about to be restarted requiring process P_a (PID=54) to be restarted it may happen that a process of job B on the same node forks a new process getting the just freed PID=54 assigned. As a consequence the restart of job A will fail because P_a cannot associated with its previous PID.

A straigtforward solution is to use virtual machine technology running each job unit on a grid node in its own virtual machine. Although there are applications where this approach makes sense it comes with considerable resource costs during fault-free execution (here we would use it for resource ID isolation, only).

Therefore, we have decided to use the new lightweight virtualization (LWV) [28] coming with Linux (2.6.24). LWV introduces *control groups (cgroups)* (formerly known as task containers). Cgroups are a generic framework that allows to define hierarchical groups of processes. Each cgroup can be associated with one or more control subsystems, e.g. namespaces, resource management. Namespaces tied to cgroups can be used to isolate processes of one cgroup from those in a different cgroup. Tasks can be inserted into cgroups from user space by using the cgroup file system. Isolation is realized by virtualizing the underlying kernel resource instead of providing a full-size virtual-machine. So far we have shown how cgroups help to isolate processes (ipc, network etc. to be supported soon) in order to allow a correct restart.

Another important feature of cgroups is that they can help bridging the gap between AEM's and specific kernel checkpointers process grouping mechanisms in a kernel checkpointer transparent way - without the need to hard-code AEM against one specific kernel checkpointer, see section V-A.

AEM sets up a cgroup per job unit and takes care that all job unit related processes are contained in the same cgroup accordingly. When AEM initiates checkpointing it passes the cgroup name to the underlying kernel checkpointer. Before this call reaches the real kernel checkpointer the intermediate CRTransLib checks if the provided cgroup name can be translated into a specific process grouping ID the underlying kernel checkpointer uses for referencing processes. If a process grouping ID has been found that exactly references *all*, not more and not less processes, in the cgroup contained processes, the job unit can be checkpointed without any further restrictions.

However, it is the CRTransLib's responsibility to intercept relevant system calls (e.g. setsid and setgrp) used by the job unit to be checkpointed with the library associated kernel checkpointer. Due to a job unit's semantic the session (SID) or process group (PGID) of a subset of processes may be

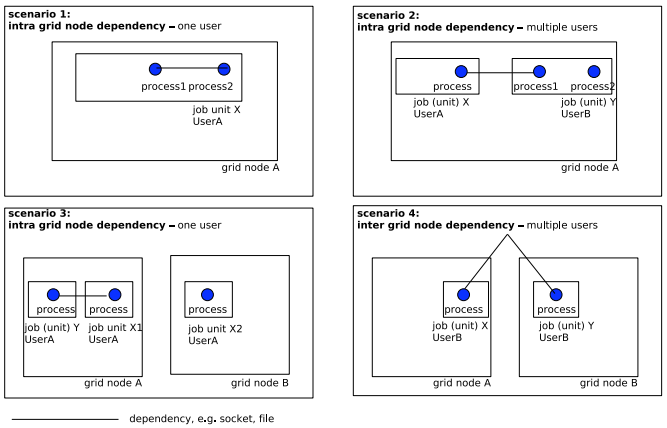

Fig. 4. Job dependency scenarios

changed by calling setsid or setpgrp. This can lead to having multiple sessions (SID) or process groups (PGID). If thenall processes in the cgroup cannot be referenced using a single ID the associated kernel checkpointer cannot checkpoint/restart a job unit.

The basic challenge is to consistently save and rebuild shared structs used by multiple process groups. Shared structs refer to C structures that represent shared software resources in the kernel such as file descriptors, memory segments, semaphores, etc. We plan to establish a book keeping for identifying process groups that need to be checkpointed and rebuilded serially. The method to save is then: synchronize all processes, checkpoint serially all process groups via multiple calls to a kernel checkpointer passing a different process group identifier each time, resume all process groups.

Shared resources need special handling to avoid unncessary overhead, e.g. for huge shared files, and to avoid deadlocks, e.g. when saving/restoring shared structures multiple times.

C. Cascading synchronization supporting job dependencies

Job dependencies occur in workflow systems if processes of a *jobA* and processes of another job *job B* share a common resource, e.g. shared file, SYSV IPC segment segment, or are connected by sockets. Obviously, it is dangerous to checkpoint/restart only one of these dependent jobs. To avoid inconsistencies and failures job dependencies need to be detected and taken into account by AEM during checkpoint/restart operations. Fortunately, the kernel checkpointers have an explicit view on kernel data-structures representing shared resources and know what processes are using which resources. Any dependency detection must be sent to AEM.

Figure 4 shows several possible process dependencies scenarios. Scenario 1 shows inner job unit dependencies. In scenario 2 *job unit X* of *jobA* and *job unit Y* of *jobB* reside on the same grid node both sharing a file. In case of a restart the file will be reset what requires to restart both affected jobs: *jobA* and *jobB*. Scenario 3 and 4 demonstrate that inner and intra job dependencies will affect more than one grid nodes when synchronizing the units.

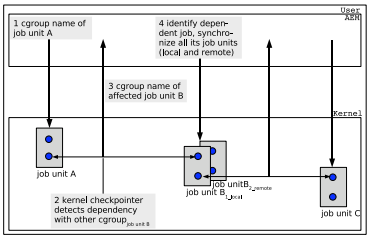

Fig. 5. Cascade synchronisation

Cascading synchronization is a generic mechanism to identify dependent jobs (units) based on dependent processes detected by the kernel checkpointers in order to consistently checkpoint/restart them.

Figure 5 shows the steps of cascade synchronization. A kernel checkpointer receives a cgroup name, that encapsulate job (unit) As processes, from AEM in order to checkpoint them. If the kernel checkpointer detects dependent processes not sharing the same cgroup (and thus the job unit / job), it informs AEM. After receiving the cgroup name of the local dependent job unit B_1 AEM identifies the dependent job and initiates the synchronization of all (local and remote) job units B_1 and B_2 belonging to job B (see scenarios 3 and 4) of this dependent job. This procedure is finished until no dependency needs to be resolved anymore. Afterwards a snapshot can be taken.

At restart all dependent job units must be identified. Thus, it requires to save a dependency graph of the involved jobs at checkpoint time.

VI. Conclusion

Grid checkpointing and restart is a mandatory facility for a grid operating system like XtreemOS. It is required for migration and fault tolerance. The first is required by the grid-wide resource scheduler which decides when a job or a job unit needs to be migrated to another grid node. The latter is essential to cope with the increasing node/process failure propability in a dynamic grid environment.

In this paper we have briefly described the XtreemOS grid checkpointing architecture, especially how different kernel checkpointers for different platforms or even on the same grid node can be integrated. A common kernel checkpointer API is necessary to manage all these different existing checkpointer implementations in an uniform fashion. Customized translation services must be provided for each specific kernel checkpointer. The XtreemOS project provides two: BCLR for PCs and Kerrighed for SSI clusters.

Beyond many challenges we have described in this paper our approach to keep track of all processes belonging to a job / job unit. Especially, for SSI clusters this is not a trivial task as these grid nodes appear as one powerful machine coming with their own cluster operating system transparently managing the cluster nodes. In order to keep track of all processes belonging to a job unit independent on which cluster node they are created we have implemented a process event

propagation system based on the process event connectors. The latter have been extended to forward events over the network to the master node where AEM is running. The proposed user-level approach avoids complex kernel modifications.

Furthermore, we have discussed process grouping techniques and how they can be integrated to enable the grid checkpointer to checkpoint all processes belonging to a job and not more or less. Our implementation is based on Linux cgroups, a recently in the Linux kernel introduced lightweight virtualization technology. Linux cgroups solve restart issues related to resource conflicts, e.g. PIDs that are no longer available and at the same time they allow a mapping of cgroup names to indivudual process grouping identifiers, depending on the underlying kernel checkpointer interface.

The source code of XtreemOS has just been released. The grid checkpointing and restart facility will be released for the public soon.

Beyond process tracking and grouping discussed in this paper there are a lot of further challenges to be solved, e.g. further kernel checkpointser need to be studied, communication channels between nodes running different kernel checkpointers, adaptive checkpointing strategies, etc. We are aware that the Linux kernel community is currently discussing about a potential native checkpoint and restart functionality to be integrated into the Linux kernel. If such a facility will be available it will simplify things and we will immediately integrate it into our architecture. Nevertheless, there are a lot of applications and libraries linked against specific kernel checkpointers with specific capabilities requiring a flexible grid checkpointing architecture like the one presented in this paper.

ACKNOWLEDGMENT

The authors would like to thank Marko Novak (XLABS, Slowenia) for working together in the context of LinuxSSI process tracking.

REFERENCES

[1] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[2] G. Jankowski, R. Januszewski, R. Mikolajczak, and J. Kovacs, "Grid checkpointing architecture - a revised proposal," Institute on Grid Information, Resource and Workflow Monitoring Systems, CoreGRID - Network of Excellence, Tech. Rep. TR-0036, May 2006. [Online]. Available: http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0036.pdf

[3] G. Jankowski, R. Januszewski, R. Mikolajczak, M. Stroinski, J. Kovacs, and A. Kertesz, "Grid checkpointing architecture - integration of low-level checkpointing capabilites with grid," Institute on Grid Information, Resource and Workflow Monitoring Services, CoreGRID - Network of Excellence, Tech. Rep. TR-0075, May 2007. [Online]. Available: http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0075.pdf

[4] N. Stone, D. Simmel, and T. Kilemann, "An architecture for grid checkpointing and recovery (gridcpr) services and a gridcpr application programming interface," Sept 2005.

[5] P. Stodgehill, "Use cases for grid checkpoint recovery," 2004.

[6] G. Schneider, H. Kohmann, and H. Bugge, "Fault tolerant checkpointing solution for clusters and grid systems," 2007-08.

[7] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: a system for migrating computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 361–376, 2002.

[8] G. Janakiraman, J. Subhraveti, and D. Turner, "Cruz: Application-transparent distributed checkpoint-restart on standard operating systems," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005, pp. 260–269.

[9] J. Duell, "The design and implementation of berkeley labs linux checkpoint/restart," Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), Tech. Rep. LBNL–54941, 2005.

[10] S. Sankaran, J. M. Squyres, B. Barrett, and A. L. adn J. Duell et al, "The lam/mpi checkpoint/restart framework: System initiated checkpointing," 2003.

[11] Checkpointing for linux. [Online]. Available: http://www.cluster.kiev.ua/eng/?chpx

[12] ckpt. [Online]. Available: http://pages.cs.wisc.edu/ zandy/ckpt/

[13] Condor. [Online]. Available: http://www.cs.wisc.edu/condor/

[14] M. Chtepen, F. Claeys, B. Dhoedt, F. DeTurck, P. Vanrolleghem, and P. Demeester, "Providing fault-tolerance in unreliable grid systems through adaptive checkpointing and replication," 2007.

[15] J. Corbalan, G. Pipan, and T. Cortes, "Requirements and specification of xtreemos services for job execution management d3.3.1," 2006.

[16] ——, "Design of the architecure for application execution management in xtreemos d3.3.2," 2007.

[17] J. Corbalan, G. Pipan, T. Cortes, M. Artac, A. Cernivec, E. Milosev, and U. Jovanovic, "Basic services for application submission, control and checkpointing d3.3.3 - basic service for resource selection, allocation and monitoring d3.3.4," 2007.

[18] A. Quin, H. Yu, Y. Jegou, and L. P. Prieto, "Design and implementation of node-level vo support d2.1.2," 2007.

[19] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello.

[20]

[21] Process events connector. [Online]. Available: http://lwn.net/Articles/157150/

[22] J. Mehnert-Spahn and M. Schoettner, "Design and implementation of basic checkpoint/restart mechanisms in linuxssi d2.2.3," 2007.

[23] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.

[24] P. L. Métayer, "Design and implementation of basic checkpoint/restart mechanisms in linux d2.1.3," 2007.

[25] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *In Proceedings of SciDAC 2006*, June 2006.

[26] J. Duell, P. Hargrove, and E. Roman, "The design and implementation of berkeley lab's linux checkpoint/restart," Berkeley Lab Technical Report, Tech. Rep. LBNL-54941, 2003. [Online]. Available: http://ftg.lbl.gov/CheckpointRestart/CheckpointPapers.shtml

[27] T. M.-P. P. A. C. in Kerrighed, "M. fertre and c. morin," November 2005.

[28] Notes from container. [Online]. Available: http://lwn.net/Articles/256389/