Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Prototype of the first advanced version of Linux-XOS

D2.1.7

Due date of deliverable: May 31st, 2009
Actual submission date: June 12th, 2009

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP2.1
Task number: T2.1.7,T2.1.8,T2.1.9,T2.1.11

Responsible institution: INRIA
Editor & and editor's address: Christine Morin
IRISA/INRIA
Campus de Beaulieu
35042 REN NES Cedex
France

Version 0.9 / Last edited by Haiyan Yu / June 11th, 2009

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	23/04/09	Christine Morin	INRIA	first template
0.1	06/05/09	An Qin	ICT	initial draft of advanced version of xos-nss-pam
0.3	10/05/09	An Qin	ICT	added documentation
0.4	14/05/09	Haiyan Yu	ICT	polishing the document
0.5	20/05/09	Surbhi Chitre	INRIA	Checkpointing based on OpenVZ container
0.6	22/05/09	Christine Morin	INRIA	Revised introduction, conclusion and executive summary
0.7	05/06/09	John Mehnert-Spahn	UDUS	Applied reiewer comments on BLCR sections
0.8	08/06/09	An Qin	ICT	Unify the paper structure
0.9	11/06/09	Haiyan Yu	ICT	Incorporated comments from reviewers

Reviewers:
Joerg Domaschka (ULM) and Barry McLarnon (SAP)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.1.7	Specification and design of advanced node level VO support mechanisms	INRIA, ICT*, TID
T2.1.8	Specification, design and implementation of advanced application unit checkpoint/restart mechanisms	INRIA*, UDUS
T2.1.9	Software Integration and Support	INRIA*, STFC, CNR, NEC, ICT, TID
T2.1.11	Implementation of advanced node level VO support mechanisms	ICT*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

This document presents a prototype of the advanced version of Linux-XOS, the XtreemOS flavor for a single PC. This prototype aims at implementing system services described in deliverables D2.1.5 [1] and D2.1.6 [3]. This document describes new features and related procedures to get, install, configure, and experiment with each component of the prototype.

The current prototype consists of two separate parts: node-level VO support mechanisms and kernel checkpointing/restarting mechanisms. The node-level VO support part presented in Section 2 describes new features of previously developed modules: a PAM extension and an NSS extension. The enhanced support mainly relies on cgroups and namespaces in the latest kernel (+2.6.20), to achieve more fine-grained resource management and application isolation. The new feature is enabled with an optional configuration so it does not affect previous XtreemOS applications which have been deployed and run in XtreemOS platform. Concerning kernel checkpointers, we now support in XtreemOS the latest official release of BLCR checkpointer and we have implemented a new kernel checkpointer based on OpenVZ containers. The latest supported version of BLCR comes with a reduced number of necessary kernel modules and comprehensive debugging support. Furthermore, the XtreemOS job-unit to process group mapping has been improved so that diverse process group identifiers can be recreated at restart. The container-based kernel checkpointer allows to checkpoint a wide range of applications relying on the container checkpoint/restart functionalities. In the current prototype, the kind of kernel checkpointer to be used for a given application is specified in the application's JSDL. When the container-based kernel checkpointer is chosen, the AEM service is in charge of creating a container where the application is launched.

Contents

1 Introduction

This document presents a prototype of the advanced version of Linux-XOS, the XtreemOS flavor for a single PC. This prototype aims at implementing system services described in deliverables D2.1.5 [1] and D2.1.6 [3]. This document describes new features and related procedures to get, install, configure, and experiment with each component of the prototype. The current prototype consists of two separate parts: node-level VO support mechanisms and kernel checkpointing/restarting mechanisms.

The current prototype of node-level advanced VO support is a proof-of-concept of extending standard Linux to make use of lightweight virtualization support in kernel (i.e. cgroup and namespace) for resource usage enforcement in VOs. The software is in prototype state, so that industrial strength cannot be expected for all functionalities.

The XtreemOS D2.1.7 prototype is to be installed and work with latest XtreemOS packages, though it could be tested separately. The recommended kernel version is 2.6.28. The node-level VO support mechanisms are presented in Section 2.

The Linux-XOS prototype implements two kernel checkpointers: a new version of the BLCR based kernel checkpointer and a new kernel checkpointer exploiting the functionalities of OpenVZ containers.

BLCR is used to provide a job-unit checkpoint/restart functionality on a single PC. BLCR does not introduce any kernel modifications and will take existing lightweight virtualization techniques into account in the future. The latest BLCR version supported by XtreemOS is 0.8.0 which fits a 2.6.28 kernel. This checkpointer is described in Section 3.

A new kernel checkpointer based on OpenVZ containers has been implemented. OpenVZ provides checkpoint, restart and migration of containers. A job unit can be run in a container and then use the underlying facility of OpenVZ to checkpoint, restart and migrate. This new kernel checkpointer is presented in Section 4.

Section 6 concludes.

2 Advanced Node-level VO support Mechanisms

2.1 Introduction

The main objectives of advanced node level VO support mechanisms are i) strong isolation of resource usage and enforcement of fine-grained access control for VOs, in terms of separated security, performance and other QoS constraints among multiple VO users accessing the same physical node. ii) auditing, logging, ac-

counting mechanisms that are needed at node level to provide the information needed by the higher-level VO management and security services.

In current Linux there are two important mechanisms related to user management: Pluggable Authentication Module (PAM) and Name Service Switch (NSS). PAM and NSS are both extensible frameworks that allow new user authentication methods or name resolving schemes to be plugged into Linux easily. VO support functionalities are implemented as specific PAM and NSS extensions together with an auxiliary runtime service, the Account Mapping Service (AMS). These extensions enable applications to process VO-level user information via standard PAM and NSS APIs. VO users are dynamically mapped into local user accounts during PAM conversations, and the mapping information can be fetched via NSS APIs. AMS is designed to serve as the back-end for PAM and NSS extensions, which does the actual mapping operations based on local configured mapping rules. AMS also acts as the local policy engine for enforcing both VO-level and node-level policies.

As discussed in D2.1.5[3] and D2.1.6[4], the `cgroup` subsystem appearing in Linux kernel (version 2.6) has been seen as a new partition technology. It provides a mechanism for aggregating/partitioning sets of processes, and all their future children, into hierarchical groups with specialized behaviors. Linux allows a control group of process hierarchy to be associated with subsystems (scheduling, memory management, accounting, etc. A subsystem is a module, which makes use of the process grouping facilities provided by cgroups, to treat groups of processes in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies limits predefined for each cgroup.

In the advanced version of VO support mechanisms, we leverage the cgroup and namespace support present in latest kernels to enforce the resource usage by VO jobs or processes and provide better isolation among them. This enhancement is done internally in PAM module without the change of upper-layer interfaces (e.g. PAM APIs). The following sections give an introduction of installation and testing procedures for the modules.

2.2 Installation

The current release of the XtreemOS NSS/PAM modules is version 0.2.0, packaged in file `xtreemos-nss-pam-0.2.0.tar.gz` [1].

[1] This file can be downloaded from `https://gforge.inria.fr/frs/download.php/22142/xtreemos-nss-pam-0.2.0.tar.gz`

2.2.1 Prerequisites

The following Linux package must be present in order to install the XtreemOS NSS/PAM modules.

Package	*Minimal version*	*Current version*	*Comments*
automake	>= 1.9	1.10	
autoconf	>= 2.59	2.61	
libtool	>= 1.5.6	1.5.22	
doxygen		1.5.1-1	to generate documentation files
libc6-dev		2.3.6	development headers
libpam		0.79-4	development headers
libdb headers	>= 4.3	4.4.20	
check	>= 0.9.3		
XtreemOS libcredstore			to store credentials
libssl-dev		0.9.8c-4	SSL development libraries
XtreemFS-server	>= 0.9		XtreemFS client for auto-mount mechanism

Libcredstore is available in WP2.3 SVN. Checkout the project:
`svn+ssh://user@scm.gforge.inria.fr/svn/xtreemos/foundation/linux-xos-md/libcredstore`
and run "make install-credstore". Libcredstore needs `libz-dev` to build.

The library `check` has to be installed for unit testing of the package. And, the package `XtreemFS-server` has to be reinstalled in local machine if testing with the auto-mount mechanism of XtreemFS volume.

Different from the previous version, new features in the new release need to be supported from kernel cgroups and namespace facilities. To make use of these facilities, the kernel needs to be compiled with the following options:

```
* General
 * Control Group support
   -> namespace cgroup subsystem
   -> cpuset support
   -> Group CPU scheduler
   -> control group freeze subsystem
   -> Basis for grouping tasks (Control Groups)
   -> Simple CPU accounting
   -> Resource counters
```

```
   -> Memory resource controllers for Control Groups
   -> Namespace support
     -> UTS namespace
     -> IPC namespace
     -> User namespace
     -> Pid namespace
* Network support
 -> Networking options
  -> Network namespace support
```

2.2.2 Compilation and installation

The following procedure depicts the compilation and installation of NSS and PAM modules:

```
$ tar xzvf xtreemos-nss-pam-0.2.0.tar.gz
$ ls -F
xtreemos-nss-pam-0.2.0/       xtreemos-nss-pam-0.2.0.tar.gz
$ cd xtreemos-nss-pam-0.2.0/
$ ./configure --prefix=/usr
...
$ make
...
$ su
# make install
...
```

After the `install` step has completed successfully, two modules have been created: an NSS modules in `/lib/libnss_xos.so` and a PAM modules in `/usr/lib/`
`pam_xos.so` and some configuration files in `/etc/xos/nss_pam`. The binaries are installed in directory `/usr/bin`.

Compilation and installation of test programs.

```
$ cd src/examples
$ pwd
/home/anqin/xtreemos-nss-pam-0.2.0/src/examples
$ make
...
```

Like in previous configuration (D2.1.4), the NSS and PAM modules need to be enabled in `/etc/nsswitch.conf` and `/etc/pam.d/`.

```
root: cat /etc/nsswitch.conf
# /etc/nsswitch.conf
passwd:         compat xos
group:          compat xos
shadow:         compat

hosts:          files dns
networks:       files

protocols:      db files
services:       db files
ethers:         db files
rpc:            db files

netgroup:       nis
root:
```

The PAM configuration file of each PAM-aware code should be updated in order to enable the XtreemOS authentication and session management. For instance, one new configuration file,`/etc/pam.d/aem`, is to be created to enable the PAM-aware interface in AEM's ExecMng component.

```
root: cat /etc/pam.d/aem
#%PAM-1.0
...
        auth       sufficient    /usr/lib/pam_xos.so
...
root:
```

Like `pam_app_conv` and `ssh-xos`, a new testing program is provided to validate new features in packages. The program named `pam_app_aem` is in src/examples/.

2.3 PAM configuration for new features

2.3.1 Opening Support of cgroups in PAM

Interact with kernel cgroup support is done in session control functions of a PAM module, namely (`pam_open_session()` and `pam_close_session()`). The subsystems are registered in kernel and are given specific names like "cpuacct", "memory",etc. Different version kernel release may support different type of subsystems. Also, custom subsystems could be added (or patched) into kernel, such as "disk" for disk quota limitation and "xconveyer" to control disk I/O bandwidth.

If the kernel is configured with cgroup related options (see 2.2.1), the PAM module is ready to manage the job processes with a control group together with resource control via subsystems. The related setting in configuration file [2] is listed as follows:

```
root: cat /etc/xos/nss_pam/pam_xos.conf
...
UseCgroups                [yes|no]
        Subsystems      cpuacct,memory[,ns,disk,net, ...]
...
root:
```

If `UseCgroups` is set as "yes", `Subsystems` also needs to be present to let cgroups facilities know which kind of subsystems are supported. Multiple subsystems are allowed in the same time, but their names have to be given explicitly (with a comma to separate them). For example,

```
UseCgroups                      yes
        Subsystems      cpuacct
```

to call "cpuacct" subsystem to control processes in PAM session. Or,

```
UseCgroups              yes
        Subsystems      cpuacct,memory,disk
```

to call three subsystems: "cpuacct", "memory", and "disk" to control processes in PAM session.

In each open PAM session, the subsystems will be loaded automatically and cleared after the session finish. When PAM-aware applications are run, the subsystems can be found in /mnt/xos_cgrp, and all pids of processes within a PAM session are also recorded in /mnt/xos_cgrp/<job_ID>/task. (A sample program pam_app_aem.c demonstrates this, which is to test the cgroups facilities for AEM job control). Currently the full cgroup mounting item "cgroup" in `subsystems` option is not supported.

2.3.2 Testing the PAM plugin using `pam_app_aem`

Since the PAM/NSS is mainly used to manage AEM jobs (WP3.4, see [5]), we need testing set to emulate the PAM-aware functionalities applied in AEM ExecMng.

The `pam_app_aem` program calls the `xpamexecvp()` to run a job. The `xpamexecvp()` is same as the one used in AEM (in ExecMng/JNI/XPamAPIs.c).

[2] /etc/xos/nss_pam/pam_xos.conf

```
user: cat src/examples/pam_app_aem.c
int main(int argc, char *argv[])
{
        unsigned char *cert_buffer = NULL;
        ...

        /* get cert from file, and to buffer */
        if (cert_path
            && xos_cert_buffromfile(cert_path, &cert_buffer,
                                    &cert_len) != XOS_SUCCESS) {
                DEBUG("Can not load certificate to buffer !");
                return -1;
        }
        ...

        /* get parameters of job structure */
        job.path = argv[2];
        job.params = argv+2;
        job.jobcert = cert_buffer;
        ...

        /* execute the PAM-aware API, which is the one
         * used in AEM job ExecMng.
         */
        if (xpamexecvp(&job) < 0)
                DEBUG("ERROR: Can not execute job !");
        ...
        return 0;

}
user:
```

The `xpamexecvp()` launches a session to encapsulate job processes at runtime. The session is created via PAM module (pam_session_open()). During the opening of the session, subsystems will be installed according to configuration with `pam_xos.so`. Next it creates a container named with job ID and puts the job's first process into that container. The `cgroups` facilities in kernel will take charge of its children processes in future. When PAM is called to close the session, the container will be destroyed and subsystems will be uninstalled.

Before testing, it would be helpful to know which cgroup subsystem is supported in current kernel. By default, there are "cpuacct", "memory" and "ns" subsystems in mainstream kernel. However, the default subsystems only support resource usage accounting rather than enforcement required by WP2.1. In the prototype, we provide a modified cgroup subsystem to do experiments of resource usage control. This cgroup subsystem is adopted to control allocation of

disk space and file inodes, which can limit the storage usage by VO users within a given quota.

The patch can be downloaded from

`svn+ssh://user@scm.gforge.inria.fr/svn/xtreemos/foundation/linux-nss_pam/branches/patches`

It adds a "disk" subsystem in current kernel. Hence, administrators can configure a new parameter named "disk" in subsystems parameter in `pam_xos.conf`.

The instructions to patch kernel can be found in subsection 2.4. The configuration is modified as follows:

```
root: cat /etc/xos/nss_pam/pam_xos.conf
...
UseCgroups              yes
        Subsystems      disk
...
root:
```

The configuration tells PAM to adopt the cgroup facilities to monitor processes and resources during job running. In this case, it asks for "disk" subsystems to take care of the control.

Before running `pam_app_aem`, we still need to make sure other setting and configuration are correct, which are similar to those in running `pam_app_conv` (see [2]), including

1. XOS-certificate used to run AEM job is present;
2. Path of CA self-signed certificates is correctly set in `pam_xos.conf`;
3. `xos_amsd` is running;
4. PAM configuration in /etc/pam.d/ is correctly set.

Item 4 would have some difference in file name. Here, we let program adopts the configuration file of `pam_app_conv`, but AEM Execmgr uses another file name (e.g. "aem").

2.3.3 Running `pam_app_aem`

The usage of `pam_app_aem` is:

```
root: ./pam_app_aem --help
Usage: ./pam_app_aem  <path/of/cert> <command> <parameters>
Example:
  ./pam_app_aem  /tmp/testbed1_usercert.pem  /bin/echo  hello
root:
```

Once everything is configured correctly, running the pam_app_aem test program as root should produce the similar result as pam_app_conv:

```
root: ./pam_app_aem  /tmp/testbed1_usercert.pem  /bin/bash
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
...
root:
```

Differently from pam_app_conv, the kernel-level cgroup subsystem ,"disk", has taken charge of the resource management. We can find the subsystem is installed via checking its corresponding pseudo-filesystem.

```
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$ ls
/mnt/xos_cgrp/
aem_jobid_123            disk.usage_in_block
disk.limit_in_block      disk.usage_in_inode
disk.limit_in_inode      notify_on_release
disk.max_usage_in_block  release_agent
disk.max_usage_in_inode  tasks
disk.stat
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
...
```

The directory, aem_jobid_123, is the name of job container, in which "disk" subsystem is inherited and process PID is recorded.

```
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$ ls
/mnt/xos_cgrp/aem_jobid_123/
disk.limit_in_block      disk.usage_in_block
disk.limit_in_inode      disk.usage_in_inode
disk.max_usage_in_block  notify_on_release
disk.max_usage_in_inode  tasks
disk.stat
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$ cat
/mnt/xos_cgrp/aem_jobid_123/tasks
16047
16149
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$ ps
  PID TTY          TIME CMD
16047 pts/0    00:00:00 bash
16153 pts/0    00:00:00 ps
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
...
```

Current process (PID: 16047) has been monitored by cgroups.

Now, we can do quota enforcement with "disk" subsystem. By interacting with cgroupfs, we can set the maximal quota for specific container.

```
root: echo 4>/mnt/xos_cgrp/aem_jobid_123/disk.max_usage_in_inode
root:
```

The above command is to tell "disk" that the container (`aem_jobid_123`) is allowed to create four files in local filesystem maximally. If total files created by processes in a container (`aem_jobid_123`) exceeds four, the requests will be rejected.

```
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
echo 1 > /tmp/test1
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
echo 2 > /tmp/test2
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
echo 3 > /tmp/test3
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
echo 4 > /tmp/test4
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
echo 5 > /tmp/test5
bash: /tmp/test10: Disk quota exceeded
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$
...
```

Finally, the "disk" subsystem will be uninstalled after logouting from `pam_app_aem`.

```
[/CN=62a57c32-9c7f-4da2-a08d-1093f5e6bee8@testbed0 tmp]$ exit
exit
root: ls /mnt/xos_cgrp/
root:
...
```

2.4 Extending the kernel to support custom cgroup subsystem

Many cgroup-based patches have been developed in kernel community. Any one can patch the kernel to get specific functionalities by standard patch instructions.

In XtreemOS, the patch of "disk" subsystem is just a simple example to show XtreemOS job get support from kernel. The patch is developed with kernel 2.6.28.5,

```
root: diffstat linux-2.6.28.5-cgroup-disk-quota.patch
 fs/ext2/balloc.c              |   20 +-
 fs/ext2/ialloc.c              |    7
 fs/ext2/xattr.c               |   17 +
 fs/ext3/balloc.c              |   23 +-
 fs/ext3/ialloc.c              |    7
```

```
 fs/ext3/xattr.c                |   15 +
 include/linux/cgroup_disk.h    |   28 ++
 include/linux/cgroup_subsys.h  |    4
 init/Kconfig                   |   10 +
 kernel/Makefile                |    1
 kernel/cgroup_disk.c           |  397 ++++++++++++++++++++++++++
 kernel/cgroup_disk.h           |   26 ++
 12 files changed, 539 insertions(+), 16 deletions(-)
root:
```

and patching the kernel can be done simply as:

```
root: cd /path/to/linux-2.6.28.5
root: patch -p1 </path/to/linux-2.6.28.5-cgroup-disk-quota.patch
root:
```

More powerful patches still need to be developed to control internal resources, such as I/O bandwidth and physical memory.

3 Kernel Checkpointer based on BLCR

BLCR 0.8.0 is latest version supported by XtreemOS as it supports 2.6.28 kernels and thus fits the Linux kernel version used by the XtreemOS release. One innovation targets the reduction of required kernel modules. Now, just two kernel modules need to be loaded at system start.
There are some new functionalities, that are visible to application-developers, too. Restoration of diverse identifiers, e.g. for a single process, a UNIX process group and a UNIX session, has been realised. This feature allows to put processes of a XtreemOS job-unit into a separate UNIX process group or UNIX session group, the job-unit-process-mapping can be realised in a safer way, especially in case a root process of a process tree terminates.
Another new functionality targets integration of debugging support. Almost all errors caused by BLCR can be reported. It helps to identify bugs related to integration issues much faster.
Additionally, a new library call capable of initiating an application restart removes the necessity to fork a new process executing the BLCR cr_restart binary. In general BLCR is capable of checkpointing/restarting single/multi process applications (single/multi-threaded). Checkpoint/restart can be refined using callbacks. Although most process resources can be saved and be restored, the following limitations remain. Support for checkpoint/restart of SYSV IPC objects as memory segments, semaphores and message queues as well as sockets is not yet provided.

3.1 How to install the BLCR based kernel checkpointer

In order to make use of the checkpointer, BLCR v0.8.0 and the XtreemOS-BLCR patch must be installed. Both are provided in the svn repository[3]. The following installation steps must be taken. The patch needs to be copied into the BLCR source directory. Then, the following sequence of calls needs to be executed:

```
# patch -p1 < patch
# ./autogen.sh
# ./configure
# make
# make install
# make insmod
# ldconfig
```

3.2 How to configure the BLCR based kernel checkpointer

Using BLCR requires the BLCR kernel modules to be loaded. Thus, insmod blcr must be applied to the BLCR source directory.
Each application to be checkpointable and restartable in XtreemOS context must be statically linked against the BLCR library (attach '-lcr' to the gcc call). The XtreemOS version of BLCR relies on a dedicated directory to host images and checkpoint-sequence related files. Therefore, the directory */xtreemfs/blcr* must be created and read/write permission be given for each XtreemOS user.

3.3 How to use the BLCR based kernel checkpointer

The XtreemOS version of BLCR cannot be used stand-alone. It is a low-level service which is called by the high-level grid checkpointer of XtreemOS. As an additional prerequisite the directory */xtreemfs/job_checkpoint_meta_data* must exist since grid checkpoint/restart data are stored there. A job checkpoint and restart on a LinuxXOS grid node can be done by issuing:

```
# xcheckpoint jobID
# xrestart jobId Version
```

[3] xtreemos/foundation/linux-xos/ckpt-support/xos-blcr

4 Checkpointer based on OpenVZ Containers

OpenVZ is a container-based virtualization mechanism which provides a multiple job execution environment for Linux. A container is an isolated entity which can be rebooted independently. It has its own root access, users/groups, IP address(es), memory, processes, files, applications, system libraries and configuration files etc. OpenVZ provides the functionality to checkpoint, restart and migrate containers. In XtreemOS, we provide the functionality to submit the grid jobs in such OpenVZ containers. From XtreemOS, these grid jobs can be checkpointed, restarted and migrated using the underlying functionality of OpenVZ.
All the job units of a same job, submitted to a same node, shall be submitted to a same container. A job unit belonging to a job new to a node, shall be submitted to a new container. A container shall be created when a job unit of a job new to a node, has to be executed. A container shall be cleaned up when all the job units inside a container have completed their work. All the job units executing inside a container shall be checkpointed and restarted together. All interprocess communication, file accesses and process state shall be checkpointed and successfully restarted when requested for the same.

4.1 Installing and configuring OpenVZ containers

XtreemOS comes with a separate kernel for OpenVZ. You need to choose this kernel while installing XtreemOS. OpenVZ user space utilities have to be installed as a part of the XtreemOS installation. The user space utilities are *vzctl, vzquota and vzpkg*. The containers in OpenVZ are created using the template files kept in */var/lib/vz/template/cache*. The default template has to be installed in this directory. There is no separate installation or configuration expected from a user.
A template is a populated root filesystem which is tar.gzipped. If a owner of a node/machine wishes that the containers created on that node/machine should have a rootfs of his choice, then he/she can put the corresponding tar.gzipped rootfs in this directory and make this the default template in */etc/vz/vz.conf*.

4.2 Using OpenVZ

The user that wishes to make use of OpenVZ for job submission, checkpoint and restart should just add the following in the jsdl file of the job:
<JobDescription>
<Checkpointer> OpenVZ </Checkpointer>
</JobDescription>

Adding the *"Checkpointer-tag"* in the JobDescription field, instructs the Application Execution Manager to submit a job in an OpenVZ based container. Later, whenever the job has to be checkpointed/restarted, the container is checkpointed/restarted appropriately.

5 Implementation Details of OpenVZ integration

5.1 Requirements

We first look at the requirements in brief to understand what code in XtreemOS is added/modified for OpenVZ integration. They are as follows:

- Identify that a job has to be submitted to a container and not a native Linux environment.
- Create a container for job submission.
- Submit the job to a container.
- At the grid level, link the job and the container to which it is submitted.
- Identify when the job executing within a container has finished.
- When the job has completed its execution remove the container and remove the job.
- When there is a checkpoint/restart request for a job get the corresponding container id and checkpoint/restart the container.

5.2 Implementation

Now we shall look at how each of the above goals are achieved.

- *Identify that a job has to be submitted to a container and not a native Linux environment*
  OpenVZ provides container-based multiple execution environment, which makes checkpointing of jobs attractive. For example, there is never the classical process id clash problem when you restart a job in an OpenVZ container. OpenVZ provides functionality to checkpoint all the common resources like open files, sockets, pipes, message queues etc. of the processes running within a container. A job which wishes to take advantage of these advanced features needs to be submitted not to a native Linux environment but to an OpenVZ container environment. So the user of a grid, should

specify that he/she needs to submit the job in an OpenVZ environment by specifying this in the JSDL file.
At job submission time, the JobManager requests for the checkpointer field in the JSDL. If this is not mentioned, then the job manager instructs the Application ExecutionManager(AEM) to submit the job in a native environment. However if this field is specified and populated with "OpenVZ" then the JobManager instructs the AEM to submit the job to an OpenVZ environment. For future retreival, the JobManager also stores the checkpointer type in the corresponding jobUnit which has to be executed.

- *Create a container for job submission:* When the job has to be submitted to an OpenVZ environment, the following steps are followed:

  1. A new container is created.
  2. A job is submitted to this container.

  When a new container id has to be created, the important points to ponder upon are :

  - The container id assignment.
  - The ip-address assignment to the container.

  The choice of the container id should be done randomly. If an ordered approach is followed, then you can have the same container ids on different nodes. For example: If you always start from id 100 then all nodes/machines having OpenVZ support shall have containers starting from 100, 101, 102.. etc. When you now want to migrate a job running in container with id 100, you have to essentially migrate the container with id 100. In this case, we need to choose a node/machine which does not have the container id 100 for migration. This will clearly lead to a container id clash. Hence we should be careful while choosing the container id. Using random() for choosing the container id solves this clash. Similarly we must take care that the ip-addresses assigned to the containers are not ordered so as to avoid the ip address clash. Clearly we cannot have two containers with the same ip address on the same node/machine. Hence ip-addresses should be globally unique (or atleast unique for a subset of machines).

- Submit the job to a container: OpenVZ provides a user space utility called as *vzctl* for job submission. We make use of this utility for submitting a job from the AEM. vzctl submits a job to the container and waits for the job to finish execution. vzctl runs outside the container where we submit the grid job for execution. As explained in the figure 1 this renders the

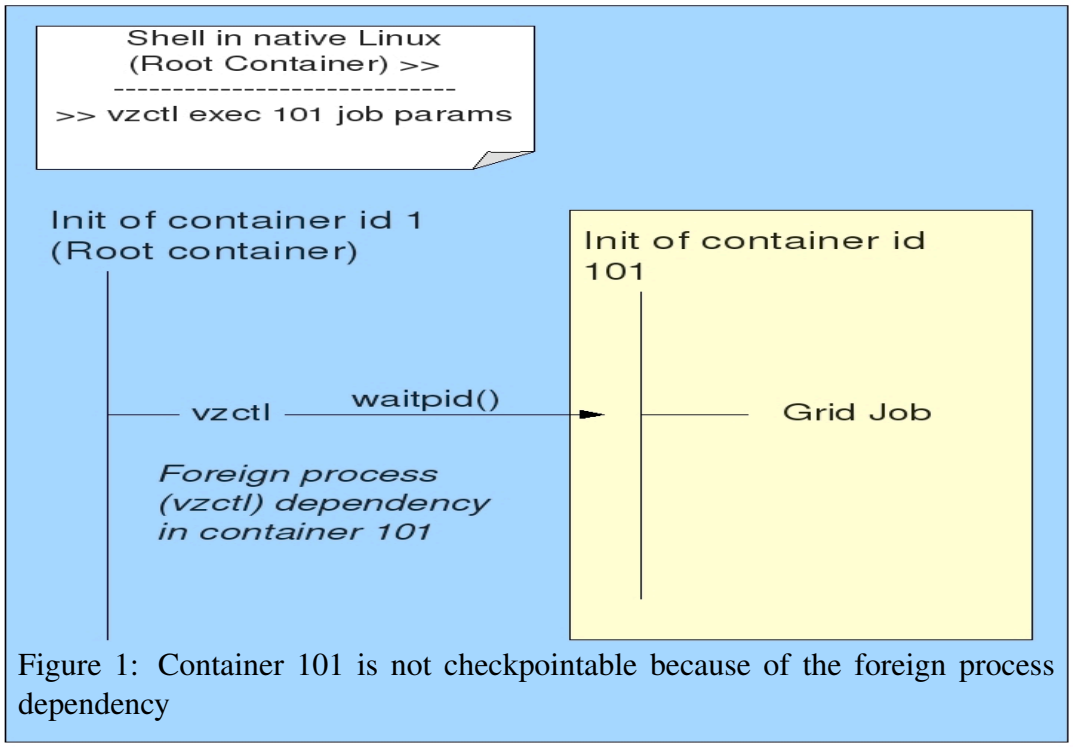

Figure 1: Container 101 is not checkpointable because of the foreign process dependency

container running the grid job non checkpointable. *OpenVZ does not let you checkpoint a container which has a process outside the container waiting for a job/process running inside the container.* So as long as the grid job executes, vzctl will wait for it and the container cannot be checkpointed till then. To overcome this we execute a loader application which shall spawn the grid job and return back to vzctl. However we are then left with a problem of finding out the grid job completion. We shall see a simple solution to this ahead.
In OpenVZ, the job submission done using vzctl can be done only by the super user. The only other method for job submission is using ssh. A user can ssh into a container and then execute processes. However this implies that we set up password less ssh logins for AEM to login to the container. This also means that we create a local grid user in the container. We chose to use the approach where we use vzctl to submit a loader application in the container. This loader application then sets the appropriate user id and group id before job submission.

- *At the grid level, link the job and the container to which it is submitted*
  We might want to stop/checkpoint/restart a job executing in XtreemOS. To achieve the same effect on a grid job submitted to an OpenVZ container, we

need to stop/checkpoint/restart that container. To do this we need to link the job id and the container id where the job is submitted. Once the AEM successfully submits a job to the container, the AEM informs the JobManager the container id for the corresponding job. To achieve this, the AEM should be aware of the job id. So we pass not only the checkpointer (i.e "OpenVZ") but also the jobid to the AEM for job execution. When the JobManager gets a message from the AEM about the container id corresponding to the jobid, the JobManager stores this information in a hashtable local to itself. For future retrieval, it also stores the container id in the jobUnit which is executing in the container.

- *Identify when the job executing within a container has finished.*
  The grid job executes within a container whereas the AEM executes outside the container. The AEM cannot directly call wait() on any job executing inside the container to avoid foreign process dependencies. So we need some mechanism to identify the grid job termination. The existing mechanism used for identifying the end of grid job in a native environment is *kernel-connectors for processes*. However this cannot be used because of the way OpenVZ is designed. To put it in brief, you cannot have a kernel-connector listener in the user space of root container(i.e the default container) listening for any event which occurs in some other container. This means that the kernel sends messages to the listeners in the corresponding containers where the events occur. So we cannot get information in the AEM using the process kernel-connector mechanism.
  The other simple mechanism, which we follow is to use a socket communication based client-server approach. The server is started in the AEM, just before submitting the job. As we saw before, a loader application is used for spawning the grid job in the container. This loader application, also spawns a client when the grid job finishes execution. This client then notifies the server in the AEM through socket communication, about the status of the job. The server executing in the AEM context informs the JobManager about the status of the job when the job finishes its execution.

- *When the job has completed its execution remove the container and remove the job* The JobManager can remove the job when the job has finished execution. The server in the AEM, also deletes the container in case the job has finished execution.

- *When there is a checkpoint/restart request for a job, get the corresponding container id and checkpoint/restart the container* The CRExecMng (checkpoint-restart manager) gets the the container id corresponding to the jobUnitId from the ExecMng. The containerId is stored in the jobUnitId. The

ExecManager retrieves this from the JobUnit and sends it to the CRExecMng. The CRExecMng sends this containerId to the translation library which shall perform the checkpoint/restart of the corresponding container. OpenVZ requires that the checkpoint/restart of the container be done only by the root user. Hence we do not call setuid() or setgid() to set the current user as the grid user/group. However we need to ensure that the checkpoint is saved on the correct partition and accessible to the grid user. This is an open issue since we have not integrated XtreemFS with OpenVZ. We will be working on this. As of now, we allow only the root user to checkpoint/restart the corresponding container.

To put the changes in a nutshell, we can say that we made some changes to the following classes JobManager, Job, JobUnit, ExecMng, CRExecMng, CRJobMng, ExecMngHandler, SExecMng, SCRExecMng, SCRExecHandler, SCRJobMng, CRJobHandler and jsdl.xsd. We also added an OpenVZ translation library for providing the checkpoint/restart/stop functionality.

5.3 OpenIssues

The following are the known open issues in this implementation.

- Mounting the partition corresponding to a grid user in the container where the job of the grid user shall be executed. The checkpoint should be saved on this partition and should be accessed by the grid user.
- Reserving a node which has OpenVZ functionality. Currently we are showing the submission/checkpoint/restart on an OpenVZ kernel. Hence we have a guarantee of the node. However we need to ensure that in a grid environment a node/machine which has OpenVZ support is reserved for a job that requires OpenVZ support.
- Assigning ip addresses to the containers through AEM.
- Setting up networking such that the containers can access the outside world. Need to write scripts for this and make them a part of the installation. As of now, the communication between the host machine and the container is set up. We are working on this currently.

6 Conclusion and Future Work

The Linux-XOS version described in this document is based on Linux 2.6.28 kernel and constitutes the foundation layer for the XtreemOS PC flavour in the second

major release of XtreemOS Grid operating system. The corresponding software packages are included in the XtreemOS installation CD produced by Mandriva in May 2009 and the source code is also available on XtreemOS repository on INRIA Gforge in anonymous read access.

The current prototype of node-level VO support is a proof-of-concept of extending standard Linux to make use of kernel-level virtualization and isolation techniques (cgroup and namespace) for resource usage enforcement. The new features provided are configured optionally which does not affect current deployed XtreemOS applications. Future work is mainly providing full-fledged subsystems to do resource control of CPU utilization, memory usage, disk and network I/O bandwidth.

The BLCR-based kernel checkpointer allows to checkpoint/restart job-units on a single PC. In the current Linux-XOS prototype the most recent version of BLCR (0.8.0) is used. Future versions of BLCR will be made compatible towards the cgroup framework and will be extended by realisation of incremental checkpointing. Thanks to the new kernel checkpointer based on OpenVZ containers a wider range of applications can be checkpointed in XtreemOS. While a first basic version of the new *container-based kernel checkpointer* is functional, its development is still in progress. Future versions of the OpenVZ container based kernel checkpointer shall support job migration and shall be compatible with XtreemFS.

References

[1] XtreemOS Deliverables - D2.1.2: Design and Implementation of Node-level VO Support.

[2] XtreemOS Deliverables - D2.1.4: Prototype of the basic version of Linux-XOS.

[3] XtreemOS Deliverables - D2.1.5: Design and Implementation of Advanced Node-level VO Support Mechanisms.

[4] XtreemOS Deliverables - D2.1.6: Evaluation of Linux native isolation mechanisms for XtreemOS flavours.

[5] XtreemOS Deliverables - D3.3.3: Basic services for application submission, control and checkpointing.