Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Containers checkpointing and live migration

Andrey Mirkin
OpenVZ
major@openvz.org

Alexey Kuznetsov
OpenVZ
alexey@openvz.org

Kir Kolyshkin
OpenVZ
kir@openvz.org

Abstract

Container-type virtualization is an ability to run multiple isolated sets of processes, known as containers, under a single kernel instance. Having such an isolation opens the possibility to save the complete state of (in other words, to checkpoint) a container and later to restart it. Checkpointing itself is used for live migration, in particular for implementing high-availability solutions.

In this paper, we present the checkpointing and restart feature for containers as implemented in OpenVZ. The feature allows one to checkpoint the state of a running container and restart it later on the same or a different host, in a way transparent for running applications and network connections. Checkpointing and restart are implemented as loadable kernel modules plus a set of user-space utilities. Its efficiency is proven on various real-world applications. The overall design, implementation details and complexities, and possible future improvements are explained.

1 Introduction

OpenVZ is container-based virtualization for Linux. OpenVZ partitions a single physical server into multiple isolated *containers*. As opposed to other virtualization solutions, all containers are running on top of a single kernel instance. Each container acts exactly like a stand-alone server; a container can be rebooted independently and have root access, users, IP address(es), memory, processes, files, etc. From the kernel point of view, a container is the separate set of processes completely isolated from the other containers and the host system.

Having a container not tied to a particular piece of hardware makes it possible to migrate such a container between different physical servers. The trivial form of migration is known as *cold* (or *offline*) *migration*, which is performed as follows: stop a container, copy its file system to another server, start it. Cold migration is of limited use since it involves a downtime which usually requires prior planning.

Since a container is an isolated entity (meaning that all the inter-process relations, such as parent-child relationships and inter-process communications, are within the container boundaries), its complete state can be saved into a disk file—the procedure is known as *checkpointing*. A container can then be *restarted* back from that file.

The ability to checkpoint and restart a container has many applications, such as:

- Hardware upgrade or maintenance.
- Kernel upgrade or server reboot.

Checkpoint and restart also makes it possible to move a running container from one server to another without a reboot. This feature is known as *live migration*. Simplistically, the process consists of the following steps:

1. Container's file system transfer to another server.
2. Complete state of container (all the processes and their resources) is saved to a file on disk.
3. The file is copied to another server.
4. The container is restarted on another server from the file.

Live migration is useful for:

- High availability and fault tolerance.
- Dynamic load balancing between servers in a cluster of servers.

This paper is organized as follows. Section 2 discusses related work. Section 3 provides prerequisites and requirements for checkpointing. Section 4 presents overall design of checkpoint and restart system. Section 5 describes the algorithm for live migration of containers. Section 6 provides possible ways for live migration optimization. Finally, the paper is ended with a brief conclusion.

2 Related Work

There are many another projects which proposed checkpoint and restart mechanisms:

- CHPOX (Checkpoint for Linux) [1]
- EPCKPT (Eduardo Pinheiro Checkpoint Project) [2]
- TCPCP (TCP Connection Passing) [4]
- BLCR (Berkeley Lab Checkpoint/Restart) [7]
- CRAK (Checkpoint/Restart As a Kernel Module) [5]
- ZAP [6]
- Sprite [10]
- Xen [8]
- VMware [9]

Not all the systems are available as open source software, and the information about some of them is pretty scarce. All the systems which are available under an open source license lack one feature or another. First, except for some written-from-scratch process migration operating systems (such as Sprite [10]), they can not preserve established network connections. Second, general-purpose operating systems such as UNIX were not designed to support process migration, so checkpoint and restart systems built on top of existing OSs usually only support a limited set of applications. Third, no system guarantees processes restoration on the other side because of resource conflicts (e.g., there can be a process on a destination server with the same PID).

Hardware virtualization approaches like Xen [8] and VMware [9] allow checkpointing and restarting only an entire operating system environment, and they can not provide checkpointing and restarting of small sets of processes. That leads to higher checkpointing and restart overhead.

3 Prerequisites and Requirements for System Checkpointing and Restart

Checkpointing and restarting a system has some prerequisites which must be supplied by the OS which we use to implement it. First of all, a container infrastructure is required which gives:

1. *PID virtualization* – to make sure that during restart the same PID can be assigned to a process as it had before checkpointing.
2. *Process group isolation* – to make sure that parent-child process relationships will not lead to outside a container.
3. *Network isolation and virtualization* – to make sure that all the networking connections will be isolated from all the other containers and the host OS.
4. *Resources virtualization* – to be independent from hardware and be able to restart the container on a different server.

OpenVZ [11] container-type virtualization meets all these requirements. Other requirements which must be taken into account during the design phase are:

1. The system should be able to checkpoint and restart a container with the full set of each process' resources including register set, address space, allocated resources, network connections, and other per-process private data.
2. Dump file size should be minimized, and all actions happening between a freeze and a resume should be optimized to have the shortest possible *delay in service*.

4 Checkpointing and Restart

The checkpointing and restart procedure is initiated from the user-level, but it is mostly implemented at the kernel-level, thus providing full transparency of the checkpointing process. Also, a kernel-level implementation does not require any special interfaces for resources re-creation.

The checkpointing procedure consists of the following three stages:

1. *Freeze processes* – move processes to previously known state and disable network.

2. *Dump the container* – collect and save the complete state of all the container's processes and the container itself to a *dump file*.

3. *Stop the container* – kill all the processes and unmount container's file system.

The restart procedure is checkpointing, *vice versa*:

1. *Restart the container* – create a container with the same state as previously saved in a dump file.

2. *Restart processes* – create all the processes inside the container in the frozen state, and restore all of their resources from the dump file.

3. *Resume the container* – resume processes' execution and enable the network. After that, the container continues its normal execution.

The first step of the checkpointing procedure and also the last step of restart procedure before processes can resume their execution is process-freeze. The freeze is required to make sure that processes will not change their state and saved processes' data will be consistent. It is also easier to reconstruct frozen processes.

Process freeze is performed by setting the special flag `TIF_FREEZE` on all the processes' threads. In this case, the `PF_FREEZE` task flag can not be used, as atomic change is required. After `TIF_FREEZE` flag is set on all the threads, each process receives a fake signal. Sending the fake signal is for moving all the threads to a beforehand known state—in this case, it is `refrigerator()`. Using just a fake signal for freezing processes has the benefit that all the signals which are on the way to a process will be saved and delivered after the process restart.

Using such a mechanism for processes freeze has benefits for processes which are in the kernel context at the moment of freezing—they will handle the fake signal before returning to user mode, and will be frozen as all the other processes are. If a process is in an uninterruptible state (system call or interrupt handling), it will be frozen right after the kernel event is completed. If a process is in an interruptible system call, it will be interrupted and handle the fake signal. In most cases,

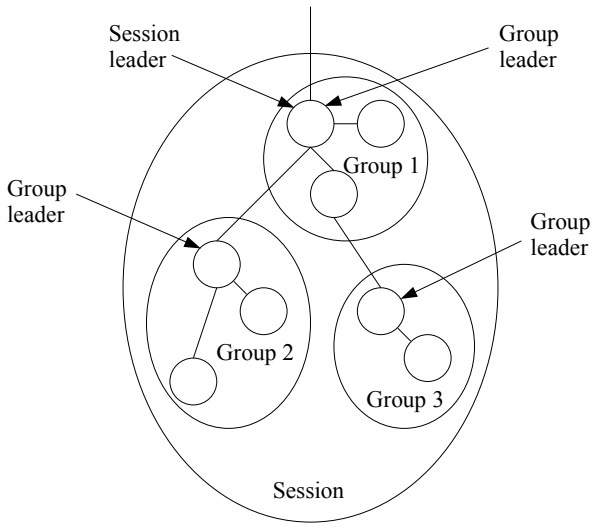


Figure 1: Process hierarchy

such system calls will be automatically restarted; otherwise, the caller should be prepared for the appropriate error handling. Such a mechanism is simple, as it uses the already implemented "software suspend" kernel feature, and so does not require much change in the kernel source code.

It is very important to save a consistent state of all the container's processes. All process dependencies should be saved and reconstructed during restart. Dependencies include the *process hierarchy* (see Figure 1), *identifiers* (PGID, SID, TGID, and other identifiers), and *shared resources* (open files, SystemV IPC objects, etc.). During the restart, all such resources and identifiers should be set correctly. Any incorrectly restored parameter can lead to a process termination, or even to a kernel oops.

Another big area of checkpointing and restart is networking. During checkpointing and restart, the network should be *disabled*—it is needed to preserve network connections. The simplest way to disable the network is to drop all incoming packets, as processes are frozen and can not process incoming packets. From the point of view of an outside, user it looks like a temporary link network problem, not something like "host unreachable" message. Such a behavior is acceptable since the TCP protocol has a mechanism to resend packets if no acknowledgment is received, and for the UDP protocol, packet loss is expected.

As most of the resources must be restored from the process context, a special function (called *"hook"*) is added

on top of the stack for each process during the restart procedure. Thus, the first function which will be executed by a process will be that "hook," and the process itself will restore its resources. For the container's `init` process, this "hook" also restores the container state including mount points, networking (interfaces, route tables, iptables rules, and conntracks), and SystemV IPC objects; and it initiates process tree reconstruction.

5 Live Migration

Using the checkpointing and restart feature, it is easy to implement live migration. A simple algorithm is implemented which does not require any special hardware like SAN or iSCSI storage:

1. *Container's file system synchronization.* Transfer the container's file system to the destination server. This can be done using the `rsync` utility.
2. *Freeze the container.* Freeze all the processes and disable networking.
3. *Dump the container.* Collect all the resources and save them to a file on disk.
4. *Second container's file system synchronization.* During the first synchronization, a container is still running, so some files on the destination server can become outdated. That is why, after a container is frozen and its files are not being changed, the second synchronization is performed.
5. *Copy the dump file.* Transfer the dump file to the destination server.
6. *Restart the container on the destination server.* At this stage, we are creating a container on the destination server and creating processes inside it in the same state as saved in dump file. After this stage, the processes will be in the frozen state.
7. *Resume the container.* Resume the container's execution on the destination server.
8. *Stop the container on the source server.* Kill the container's processes and unmount its file system.
9. *Destroy the container on source server.* Remove the container's file system and config files on the source server.

If, during the restart, something goes wrong, the migration process can be rolled back to the source server, and the container will resume execution on the source server as if nothing happened.

In live migration for external clients which connected to the container via the network, the migration process will look like a temporary network problem (as live migration is not instantaneous). But after a delay, the container continues its execution normally, with the only difference being that it will already be on the destination server.

In the above migration scheme, Stages 3–6 are responsible for the most delay in service. Let us take a look at them again and dig in a little bit deeper:

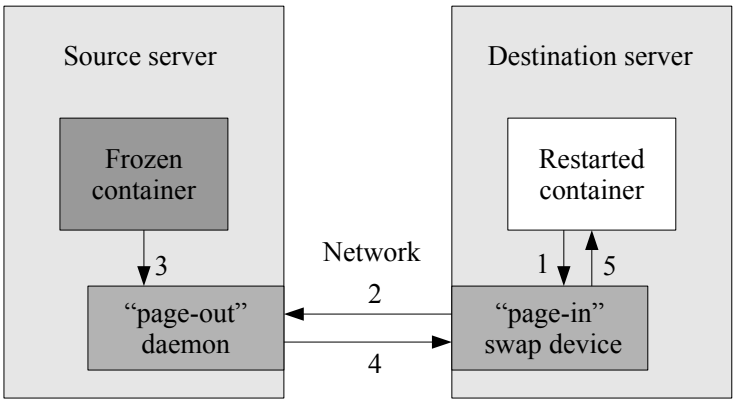
1. *Dump time* – the time needed to traverse over all the processes and their resources and save this data to a file on disk.
2. *Second file system sync time* – time needed to perform the second file system synchronization.
3. *Dump file copying time* – time needed to copy the dump file over the network from the source server to the destination server.
4. *Undump time* – time needed to create a container and all its processes from a dump file.

6 Migration Optimizations

Experiments show that second file system sync time and dump file copying time are responsible for about 95% of all the delay in service. That is why optimization of these stages can make sense. The following options are possible:

1. Second file system sync optimization – decrease the number of files being compared during the second sync. This could be done with the help of *file system changes tracking* mechanism.
2. Decreasing the size of a dump file:
   (a) *Lazy migration* – migration of memory after actual migration of container, i.e., memory pages are transferred from the source server to the destination on demand.
   (b) *Iterative migration* – iterative migration of memory before actual migration of container.

These three optimizations are described below.



1. Request a page from swap.
2. Resend the request to the source server.
3. Find the page on the source server.
4. Transfer the page to the destination server.
5. Load the page to memory.

Figure 2: Lazy migration

6.1 File System Changes Tracking

The idea is that when this system is activated, it begins to collect the names of the files being changed and stores them in a list. The list of modified files is to be used during the second file system synchronization. It can dramatically decrease second file system synchronization time. Tracking file system changes can not be implemented as a separate loadable kernel module, as it requires core kernel changes.

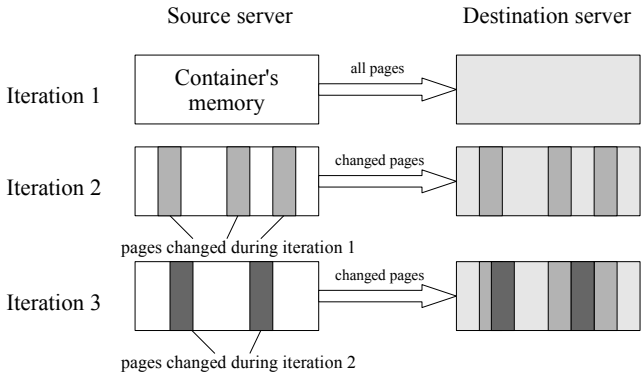


Figure 3: Iterative migration

6.2 Lazy Migration

During live migration, all processes' private data are saved to a dump file, which is then transferred to the destination server. In the case of large memory usage, the size of the dump file can be huge, resulting in an increase of dump file transfer time, and thus in an increased delay in service. To handle this case, another type of live migration can be used—*lazy migration*. The idea is the following—all the memory pages allocated by processes are marked with a special flag, which is cleared if a page is changed. After that, a container can be frozen and its state can be dumped, but in this case only pages without this flag are stored. That helps to reduce the size of a dump file.

The only problem which should be also solved here is how to transfer all the remaining memory pages from the source server to the destination. A special *page-in* swap device on the destination server and a *page-out* daemon on the source server are proposed to solve this problem.

During processes restart on the destination server, all the pages which are not saved to the dump file are marked as swapped to a page-in device. When a process resumed on the destination server accesses a page which is marked as swapped, a request to the swap device is generated. The page-in device resends this request to the page-out daemon on the source server. The page-out daemon sends the requested page to the destination server, and then this page is loaded into memory on the destination server. See Figure 2 for details. During the first few minutes pages, are transferred to the destination server on demand. After a while, the swap-out is forced, and all the pages are transferred from the source server to the destination.

6.3 Iterative Migration

Another way to decrease the size of the dump file is to transfer memory pages in advance. In this case, all

the pages are transferred to the destination server before container freeze. But as processes continue their normal execution, pages can be changed and transferred pages can become outdated. That is why pages should be transferred *iteratively*. On the first step, all pages are marked with a *clean* flag and transferred to the destination server. Some pages can be changed during this process, and the *clean* flag will be removed in this case. On the second step, only the changed pages are transferred to the destination server. See Figure 3 for details. This iterative process stops if the number of the changed pages becomes zero, or the number of the changed pages becomes more than $\frac{N}{2^i}$, where N is the total number of pages and i is the iteration number.

All the transferred pages temporarily stored on the destination server are used during the restart process. All the pages changed during the last iteration are stored in a dump file and restored from it during the restart process.

7 Conclusion

The checkpointing and restart mechanism for containers has been designed and implemented in the OpenVZ Linux kernel. On top of this mechanism, the live migration feature has been implemented, allowing the movement of containers from one server to another without a reboot. The efficiency of the system has been proven on various real-world applications. Possible optimizations of the migration algorithm have been proposed to decrease the delay in service.

References

[1] O.O. Sudakov, Yu.V. Boyko, O.V. Tretyak, T.P. Korotkova, E.S. Meshcheryakov, *Process checkpointing and restart system for Linux*, Mathematical Machines and Systems, 2003.

[2] Eduardo Pinheiro, *Truly-Transparent Checkpointing of Parallel Applications*, Federal University of Rio de Janeiro UFRJ.

[3] Eduardo Pinheiro, Ricardo Bianchini, *Nomad*, COPPE Systems Engineering, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil.

[4] Werner Almesberger, *TCP Connection Passing*, In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, July, 2004).

[5] Hua Zhong, Jason Nieh, *CRAK: Linux Checkpoint/Restart As a Kernel Module*, Department of Computer Science, Columbia University, Technical Report CUCS-014-01, November 2001.

[6] Steven Osman, Dinesh Subhraveti, Gong Su, Jason Nieh, *The Design and Implementation of Zap: A System for Migrating Computing Environments*. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 9–11, 2002.

[7] Jason Duell, *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*, Lawrence Berkeley National Laboratory

[8] Xen. `http://www.xen.org`

[9] VMware, Inc. `http://www.vmware.com`

[10] The Sprite Operating System. `http://www.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html`

[11] OpenVZ. `http://openvz.org`