Checkpoint and Restoration of Micro-service in Docker Containers

Chen Yang

School of Information Security Engineering, Shanghai Jiao Tong University, China 200240

chen.yang@sjtu.edu.cn

Keywords: Lightweight Virtualization, Checkpoint/restore, Docker.

Abstract. In the present days of rapid adoption of micro-service, it is imperative to build a system to support and ensure the high performance and high availability for micro-services. Lightweight virtualization, which we also called container, has the ability to run multiple isolated sets of processes under a single kernel instance. Because of the possibility of obtaining a low overhead comparable to the near-native performance of a bare server, the container techniques, such as openvz, lxc, docker, they are widely used for micro-service [1]. In this paper, we present the high availability of micro-service in containers. We investigate capabilities provided by container (docker, openvz) to model and build the Micro-service infrastructure and compare different checkpoint and restore technologies for high availability. Finally, we present preliminary performance results of the infrastructure tuned to the micro-service.

Introduction

Lightweight virtualization, named the operating system level virtualization technology, partitions the physical machines resource, creating multiple isolated user-space instances. Each container acts exactly like a stand-alone server. A container can be rebooted independently and have root access, users, IP address, memory, processes, files, etc. Unlike traditional virtualization with the hypervisor layer, containerization takes place at the kernel level. Most modern operating system kernels now support the primitives necessary for containerization, including Linux with openvz, vserver and more recently lxc, Solaris with zones, and FreeBSD with Jails [2]. From the kernel point of view, a container is the separate set of processes completely isolated from the other containers and the host system.

To guarantee applications' high availability despite and in presence of a disastrous and disruptive event, high availability feature such as monitoring, live migration and checkpoint/restore, are integrated into the virtualization technologies. Checkpoint-restore of Micro-service application is the ability to save the state of a running application so that it can later resume its execution from the time of the checkpoint. Application checkpoint-restore can provide many benefits, including fault recovery by rolling back applications to a previous checkpoint, better response time by restarting applications from checkpoints instead of from scratch, and better system utilization by suspending jobs on demand. Application migration is useful for dynamic load balancing by moving applications to less loaded hosts, fault resilience by migrating applications off of faulty hosts, and improved availability by evacuating applications before host maintenance so that they continue to run with minimal downtime [3].

This paper present a high availability solution for the micro-service application in a container. The high availability solution is achieved by the checkpoint/restore approach which allows one to checkpoint the state of a running container and restore it later on the same or a different host, in a way transparent for micro-service. With the help of implementing this checkpoint/restore feature to docker containers, micro-services could be deployed more convenient and high availability in the cloud infrastructure based on container.

Related Work

Checkpoint-restore has been the subject of extensive research, spanning all four approaches: application level, library mechanisms, operating system mechanisms, and hardware virtualization.

© 2015. The authors - Published by Atlantis Press

There are many application checkpoint-restore projects, some in userspace and others in the kernel. DMTCP implements checkpoint/restore of a process on a library level. This means, that if you want to C/R some application you should launch one with DMTCP library (dynamically) linked from the very beginning. Restoration of process set is also tricky, as it frequently requires restoring an object with the predefined ID and kernel is known to provide no APIs for several of them [4]. Berkeley Lab Checkpoint/Restart (BLCR) is a part of the Scalable Systems Software Suite, developed by the Future Technologies Group at Lawrence Berkeley National Lab under SciDAC funding from the United States Department of Energy [5]. BLCR is aimed primarily at HPC users. It consists of a library and a kernel module. Applications must be checkpoint-aware so as to discard unsupported resources. PinLIT (c) is a checkpointing tool built on top of Intel's proprietary PIN binary instrumentation tool [6]. It records the processor's (big) architectural register state and all pages of memory that contain application and shared library code, optimizing size by only storing memory used during a desired interval. Legacy OpenVZ (RHEL4, RHEL5, RHEL6 based kernels) has in-kernel checkpoint/restore, sources can be found in kernel/cpt/ [7]. Linux Checkpoint/Restart was a project from around 2008 to around 2010 to implement checkpoint/restart of Linux processes [3]. CRIU (Checkpoint/Restore in Userspace) is a project to implement checkpoint/restore functionality for Linux in userspace. CRIU doesn't require any libraries to be pre-loaded. It will checkpoint and restore any arbitrary application, as long as kernel provides all needed facilities. Kernel support for some of CRIU features were added recently, essentially meaning that a recent kernel version might be required [8].

Proposed Approach

The checkpoint and restore approach for Docker container with CRIU can be divided into two parts, checkpoint and restore.

Part 1: Checkpoint. The checkpoint procedure relies heavily on /proc file system (it's a general place where criu takes all the information it needs). Which includes the files descriptors information, pipes parameters and memory maps. The process dumper does the following steps during checkpoint stage:

Step 1: Collect process tree and freeze it. By using this $pid the dumper walks though /proc/$pid/task/ directory collecting threads and through the /proc/$pid/task/$tid/children to gathers children recursively. While walking tasks are stopped using the ptrace's PTRACE_SEIZE command.

Step 2: Collect tasks' resources and dump them. In this stage, CRIU reads all the information (it knows) about collected tasks and writes them to dump files. Then CRIU injects a parasite code into a task via ptrace interface.

Step 3: Cleanup. After everything dumped (such as memory pages, which can be written out only from inside dumped address space) we use ptrace facility again and cure dumped by dropping out all our parasite code and restoring original code. Then CRIU detaches from tasks and they continue to operate.

Part 2: Restore. The restore procedure is done by CRIU morphing itself into the tasks it restores. On the top-level it consists of 4 steps:

Step 1: Resolve shared resources. At this step CRIU reads in image files and finds out which processes share which resources. Later shared resources are restored by someone process and all the others either inherit one on the 2nd stage (like session) or obtain in some other way.

Step 2: Fork the process tree. At this step CRIU calls fork() many times to re-created the processes needed to be restored.

Step 3: Restore basic tasks resources. Here CRIU restores all resources but memory mappings exact location, timers, credentials, threads. On this stage CRIU opens files, prepares namespaces, maps (and fills with data) private memory areas, creates sockets, calls chdir() and chroot() and dome some more.

Step 4: Switch to restorer context, restore the rest and continue. Since criu morphs into the target process, it will have to unmap all its memory and put back the target one. While doing so some code

should exist in memory (the code doing the munmap and mmap). At the same place we restore timers not to make them fire too early, here we restore credentials to let criu do privileged operations (like fork-with-pid) and threads not to make them suffer from sudden memory layout change.

Experiment and Performance Evaluation

The measurements was conducted on two independent hosts, and each one had an Ubuntu 14.04 system with Intel SandyBridge CPU (E5-2620) clocked at 2GHz and 8GB DRAM. First of all, a CentOS7 image was used to launch a minimal version of a container in each host, then openssh, openmpi and the micro-service application were installed. Unfortunately, a lot of standard software packages are installed not in usual place so before compilation there is a need to export PATH and libs, but once it was done, this prepared image was saved. This is a great advantage of container-based virtualization, because all steps were done only once.

For the measurements we used the program which allows a caller specify the number of child process, a memory size for each task to map, or a memory size for each task to map and then dirty. We repeated the tests specific times (eg: 50), and report average values.

Application Image size	Checkpoint Time (to file)	Checkpoint Time (no I/O)	Restore time
56 MB	509 ms	173 ms	492 ms
118 MB	1063 ms	313 ms	972 ms
256 MB	2183 ms	679 ms	1998ms

Table 1. Checkpoint/restore performance on different applications

Table 1 presents the results in terms of checkpoint different micro-service applications with different image size, checkpoint time and restore time. The results show that the checkpoint and restore time scale linearly with the micro-service application image size.

To look further at the impact of the micro-service application image size, we measured the checkpoint for the specific application with memory sizes increasing from 1MB to 1GB. In one container the application only allocate memory but do not change it. In the other container, the application provide service to outside which would write memory to the allocated space. In both cases, the output was redirected to avoid expensive I/O. The results are given in Table 2. The results show strong correlation between the memory change and checkpoint time. Checkpoint times increase substantially with changed memory as it requires the contents to actually be stored in the checkpoint image.

Memory Size	Checkpoint Time (memory untouched)	Checkpoint Time (memory changed)
1 MB	3.6 ms	9.8 ms
10 MB	31.1 ms	103 ms
100 MB	297 ms	982 ms
1 GB	2763 ms	9312 ms

Table 2. Checkpoint time and memory size

Summary

A generic checkpoint/restore mechanism was presented based on the criu tool for container based micro-service architecture. With criu, you can freeze a running application (or part of it) and checkpoint it to a hard drive as a collection of files. You can then use the files to restore and run the application from the point it was frozen at. CRIU provides transparent, reliable, flexible, and efficient application checkpoint-restore. We investigated some current checkpoint-restore projects for processes and gave a simple comparison. On top of this mechanism, we presented the procedure of checkpoint/restore an application in a docker container. In addition, the experiment and performance

evaluation shows that the checkpoint and restore time scale linearly with the micro-service application image size.

References

[1] Kratzke N. A lightweight virtualization cluster reference architecture derived from open source paas platforms[J]. Open J. Mob. Comput. Cloud Comput, 2014, 1: 17-30.

[2] Information on https://www.docker.com/

[3] Laadan O, Hallyn S E. Linux-CR: Transparent application checkpoint-restart in Linux[C]//Linux Symposium. 2010: 159.

[4] Ansel J, Aryay K, Coopermany G. DMTCP: Transparent checkpointing for cluster computations and the desktop[C]//Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009: 1-12.

[5] Jason Duell. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Lawrence Berkeley National Laboratory, 2005.

[6] Narayanasamy S, Pereira C, Patil H, et al. Automatic logging of operating system effects to guide application-level architecture simulation[J]. ACM SIGMETRICS Performance Evaluation Review, 2006, 34(1): 216-227.

[7] Perkov L, Pavković N, Petrović J. High-availability using open source software[C]//MIPRO, 2011 Proceedings of the 34th International Convention. IEEE, 2011: 167-170.

[8] CRIU. Checkpoint/Restore In Userspace. http://criu.org/

[9] Mirkin A, Kuznetsov A, Kolyshkin K. Containers checkpointing and live migration[C] //Proceedings of the Linux Symposium. 2008: 85-92.

[10] Garg R, Sodha K, Cooperman G. A generic checkpoint-restart mechanism for virtual machines[J]. arXiv preprint arXiv:1212.1787, 2012.

[11] Tang X, Zhang Z, Wang M, et al. Performance Evaluation of Light-Weighted Virtualization for PaaS in Clouds[M]//Algorithms and Architectures for Parallel Processing. Springer International Publishing, 2014: 415-428.

[12] Li W, Kanso A. Comparing Containers versus Virtual Machines for Achieving High Availability[J].

[13] Pickartz S, Gad R, Lankes S, et al. Migration Techniques in HPC Environments[C]//Euro-Par 2014: Parallel Processing Workshops. Springer International Publishing, 2014: 486-497.

[14] Ljubuncic I, Rozenfeld A, Goldis A, et al. Be Kind, Rewind[J].