Live Migration of Container Networks with CRIU

Carlos Segarra
Universitat Politècnica de Catalunya
Barcelona, Spain
carlos.segarra@estudiant.upc.edu
ORCID: 0000-0003-3455-7563

Jordi Guitart
Computer Architecture Department
Universitat Politècnica de Catalunya
Barcelona, Spain
jguitart@ac.upc.edu

***Abstract*—One of the major hurdles for containers to become the cloud tenants' choice for application sandboxing is fault-tolerancy and load balancing. Checkpoint/Restore provides an efficient solution as it enables to save snapshots of running programs from which execution can be restored. Checkpointing and live migration for virtual machines has been around for several years now and, albeit less efficient, it provides with the robustness that a datacenter requires. However, checkpoint/restore (C/R) of running containers is still an open and active area of research. The biggest attempt to filling such a gap is Checkpoint-Restore in Userspace (CRIU) a software tool designed to dump, manage, and restore running processes by leveraging existing interfaces of the Linux Kernel. It does so completely from user-space and fully transparently.**

In this work we present a proof of concept implementation for live migration of established TCP connections of runC containers. Although C/R of established sockets is already available with CRIU, migrating such a connection is left to the user, hence the novelty of our holistic approach. We also present a benchmark of the downtime experienced by a client when a network-intensive server is dumped and restored. Lastly, this short paper is a work in progress towards achieving C/R of container clusters to provide tools such as KUBERNETES with fast restore times in the event of a node failure.

***Index Terms*—containers, checkpoint, restore, criu, runc**

I. INTRODUCTION

Containers have become the *de-facto* alternative for managing application's life cycle in the cloud. With the progressive shift from bare-metal, to virtualized servers, and now with containerization, cloud tenants aim to find the balance between optimal resource usage and quality of service (QoS) perceived by the user. A key aspect to achieving a good QoS is the ability to manage resources efficiently, in particular load-balancing.

Checkpoint-Restore is, through live migration, a technique to provide load-balancing capabilities to cloud tenants transparently to the user. By dumping the state of an application and restoring it in another physical instance, it will resume from the exact point it was dumped at. As a consequence the user will perceive a minimal downtime and the tenant will have re-allocated resources. Checkpoint-Restore in Userspace (CRIU) [1] is a software tool to dump and restore processes transparently to the user. It targets specially applications running inside containers, and is already used in a variety of projects and companies such as Google [2].

Identify applicable funding agency here. If none, delete this.

In this paper we present a proof of concept implementation of live migration of containers with established TCP connections. Although the checkpointing of established TCP sockets is already supported in CRIU [3], the work we present covers the necessary additions to safely migrate it to a different environment: namely routing filters and address reuse. We test migration between different virtual machines and network namespaces, and provide a benchmark of the throughput downtime perceived by the client in a network intensive application.

Our results suggest that the technology is mature enough to be embedded in container orchestration engines where this additional feature could be of particular interest towards distributed migration of container clusters.

II. BACKGROUND CONCEPTS

This project builds on two foundational concepts of computer systems: containers, and checkpoint-restore.

A. CRIU: *Checkpoint-Restore in Userspace*

Checkpoint/Restore in Userspace (CRIU) is an open-source C/R tool [1]. Introduced in 2011, it's distinctive feature is that it is mainly implemented in user space, rather than in the kernel, by using existing interfaces [4]. One of the most important interface is `ptrace` [5], as it relies on it for seizing the target process. For other interfaces, several patches have been pushed to the mainline kernel by CRIU developers [6]. The project is currently under active development [7], and its main focus is to support the migration of containers.

The checkpointing process starts with the process identifier (PID) of a process group leader provided by the user through the command line using the `--tree` option [8].

Once all tasks are frozen, CRIU collects all the information it can about the task's resources. File descriptors and registers are dumped through a `ptrace` interface and are parsed from `/proc/$pid/fd` and `/proc/$pid/stat` respectively.

During the restore process, CRIU morphs into the to-be-restored task. Since we checkpoint process trees rather than single processes, CRIU must `fork` itself several times to recreate the original PID tree. Then CRIU restores all basic task resources such as file descriptors and namespaces, and resumes the process.

B. runC: OCI-compliant container runtime

Originally developed at Docker, runC is a lightweight container runtime aimed to provide low-level interaction with containers. In 2015 [9], Docker open-sourced the component and transferred ownership to the Open Container Initiative (OCI), who has since then lead the project in a fashion similar to that of the Linux Foundation. Since then, several container engines such as PODMAN (https://podman.io/) and CRI-O (https://cri-o.io/) have made runC their runtime of choice.

The OCI has since then released specifications for container runtimes, engines, images, and image distribution. runC is nowadays an OCI-compliant container runtime (it is, in fact, the reference implementation).

Users are encouraged to interact with containers through container engines, but runC itself provides an interface to create, run, and manage containers natively. Integration with CRIU has to be done on a per-project basis, and runC has the most advanced and stable integration. Therefore, we decided to use it to manage our containers.

Running a container with runC is slightly different than doing it in, let's say, Docker, as the user's interaction with the underlying system is more direct. In particular, in runC there is no notion of *images*. To run a container, a user must provide a specification file (config.json) and a root file system in a directory (rootfs). Through the specification file several low-level options such as namespaces, control groups, and capabilities can be configured.

III. CHECKPOINT/RESTORE OF ESTABLISHED CONNECTIONS

The ability to checkpoint established TCP connection is mainly due to the inclusion of the TCP_REPAIR socket option to kernel version 3.5 [10].

A. Implementation in CRIU

Similarly to other resources and as introduced in §II, basic information about sockets is obtained by parsing the adequate files in the /proc filesystem. However, there are some internals of active network connections (namely negotiatied parameters such as send and receive queues, and sequence numbers) that require putting the socket in the TCP_REPAIR state using the setocketopt() syscall (note that this action requires CAP_NET_ADMIN capabilities). Then, if the connection is closed whilst the socket is in TCP_REPAIR mode, no FIN nor RST packets are sent to the other peer, what means that his endpoint is effectively still open [11].

To re-establish the connection from the newly generated socket, the first thing to do is put it, again, in TCP_REPAIR mode. Then, the previously dumped parameters can be set, and upon connect() the socket goes directly into ESTABLISHED mode without acknowledgment from the other end, and a RST packet is sent to resume communication.

The last missing piece is what happens if the remote end tries to send a packet to its, seemingly open, TCP socket whilst the other peer is down. Were we to ignore this fact, once the packet reached our kernel this, given that the socket is closed, would send a RST to the other end, and our whole illusion would collapse. To overcome this issue, upon checkpoint we include a set of rules to the netfilter [12] IP routing table to drop all packets. We include the set of rules in Table I.

```
Chain INPUT (policy ACCEPT)
target    prot  opt   source dest
CRIU      all   --    .../0  .../0

Chain FORWARD (policy ACCEPT)
target    prot  opt   source dest

Chain OUTPUT (policy ACCEPT)
target    prot  opt   source dest
CRIU      all   --    .../0  .../0

Chain CRIU
target    prot  opt   source dest
ACCEPT    all   --    .../0  .../0  mark 0xc114
DROP      all   --    .../0  .../0
```

TABLE I
OUTPUT OF RUNNING IPTABLES -T FILTER -L -N.

B. How to efficiently reuse IP addresses?

A caveat of restoring established TCP connections is that, without bringing down both peers, we can not circumvent the negotiated IP:PORT pairs. As a consequence, the same IP address and port must be available at restore time. Otherwise, when the remote peer receives the RST package it will immediately close the connection. Re-using an IP address is achievable using locally scoped addresses or network namespaces. In our experiments we tested both.

Firstly, if we are migrating into a different machine (as the experiments presented in §IV), we need to assign addresses using ip's addr subcommands. In particular, we are using a host-only [13] subnet to manage our (virtual) machines.

Alternatively, we have also tested process migration within the same machine, from one network namespace to a different one. This situation is particularly interesting as it recreates what happens under the hood in CRIU's binding for runC, as containers rely on namespaces for isolation. We set up a bridge device in the host namespace, two network namespaces, and two virtual ethernet devices with one peer tied to the bridge, and the other one inside a namespace. Adequately setting up addresses and default gateway routes, we achieve the setup we depict in Figure 1.

C. Live migration and integration with runC

Integration with runC is two-fold. For the TCP connection CRIU's binding for runC includes a --tcp-established flag that does most of the socket management. If we are interested in restoring the connection in a different machine or namespace, we must manually recreate the filter table from Table I using the iptables command. Lastly, to restore into an existing namespace,

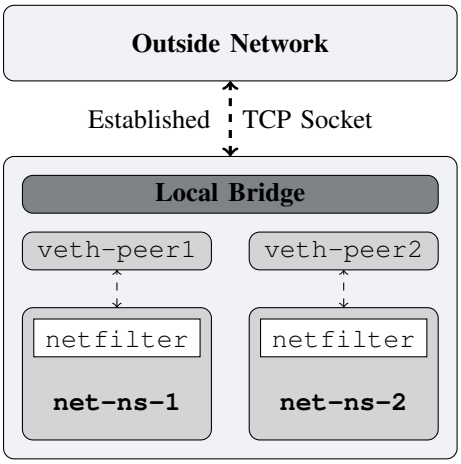

Fig. 1. Architecture of three different namespaces connected through virtual ethernet pairs.

the container must be restored with the adequate open file descriptors using CRIU's `--external` [14] and `--inherit-fd` [15].

IV. Evaluation

In order to evaluate the impact of migrating a process with an established TCP connection, we are interested in assessing how quickly can communication resume after restore.

To achieve this goal we set up the following testbed. We first deploy two identical virtual machines running Linux Debian with kernel version `4.19.0-6`. Each one has CRIU version 3.13 and `runC` version `1.0.0-rc8`, both built from source. Additionally, and in order to conduct the experiments, we make use of `iPerf3` (version `3.7+`) a network bandwidth benchmarking tool [16]. In particular, we start an `iPerf3` client-server pair, one in each VM, and record the perceived throughput by the client. Each experiment is repeated running the bare processes and checkpointing them using CRIU, or isolating them within a `runC` container, to assess the introduced overhead. We measure from the client side since we are interested in dumping and restoring the server. This situation makes more sense from the cloud-provider/load-balancing standpoint.

Re-connection after a down period. The first experiment simulates the scenario in which the server is restored some time after the dump occured. In particular, we let the client saturate the link for 10 seconds, then dump the server, and restore it 2 seconds later, all of which transparently to the client (whose connection is never closed). In Figure 2 we present the throughput perceived by the client as a function of time. The first observation we make from the plot, is that it takes almost a full second to get the connection back to full speed. To understand this behaviour we must recall what is `iPerf3` actually doing. The client tries to saturate the link, sending as many packets as it can, and reports the measured capacity. As the socket is never closed, and packets are just discarded by the network filters, to the client it will be as if

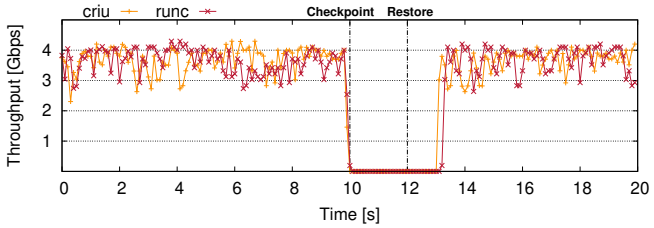

Fig. 2. Throughput perceived from the client as a function of time, when we checkpoint the server once, and restore it after two seconds. We compare the results of CRIU and `runC`.

those packets were never acknowledged, and hence will try to retransmit them. The TCP protocol specifies [17] that the retransmission timeout must be doubled every time a packet is not acknowledged, therefore the recurrent outage of ACKs might cause the client to back-off for the perceived full second. This implies that checkpointing established TCP connections only makes sense in the scenario in which the service is soon going to be restored. The next experiment tackles the behaviour under this situation.

Reactivity to immediate restore. To prove our hypothesis that the large delay after a restore is due to the protocol itself rather than our implementation, we set up an experiment in which we perform a sequence of dumps and immediate restores of the same established TCP connection. We again present the throughput as a function of time in Figure 3. In

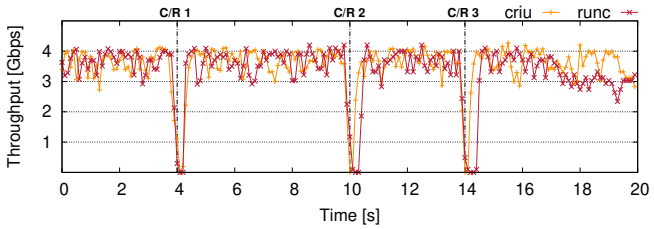

Fig. 3. Throughput from the client as a function of time.

this case the measured throughput downtime does not exceed 0.1 seconds, an order of magnitude better than the previous experiment. This reduced value, together with the fact that the application studied is very network-intensive, makes us believe that our proposed technique is suitable for most client-server scenarios and won't have an impact in the overall quality of service.

Lastly, in both Figures we observe that, albeit being the experiments running bare processes with CRIU slightly faster to restore, the overhead introduced by `runC` is negligible.

V. Related Work

Other than the technical references introduced in §II a great starting point for namespaces and containers are a series of articles by Michael Kerrisk published in LWN [18]. Similarly, for Checkpoint/Restore Barker [19] draws clearly the different approaches different software tools take.

Reber has a series of articles that describe CRIU's architecture and how it can be leveraged in different use-cases [4], [20], [20], [21]. Venkatesh presents a system for fast in-memory migration of Docker containers using CRIU. The author focuses on same-machine restore, hence why his approach is not applicable in our case.

Lastly, and with regard to live migration of running processes, Clark *et. al* [22] have a great survey on the topic. An effort to optimize image transferring in the event of a live migration using CRIU was presented in 2018 [23], and the findings there presented are soon to be integrated in mainline CRIU. Two different models for efficient live migration of containers in HPC were also presented through 2019 [24], [25]. Our approach focuses on established TCP connections, which was a overlooked by these previous approaches. An in-depth walkthrough of the kernel internals that allow for such a migration were presented by Corbet in 2012 [11].

VI. Conclusions & Future Work

In this short paper we have presented a preliminary evaluation of live migration of established TCP connections using Checkpoint-Restore In Userspace (CRIU).

In particular, we have implemented proofs of concept for migration between different machines and different namespaces within the same machine. In addition to that, we have presented a benchmark of the impact of migrating a network-intensive server with regard to the perceived throughput from the client endpoint. Our results show that migration introduces little overhead whereas checkpointing and restoring at a later point in time has an impact on connection restore time. Lastly, our results also show that choosing to run the to-be-migrated server within a `runC` container introduces minimal overhead when compared to checkpoint-restore of standalone processes. As a consequence, we consider CRIU to be mature enough to be integrated in bigger container stacks, particularly in container orchestrators were the ability to checkpoint established network connections could be of great use for load-balancing purposes.

Moving forward, we want to replicate the results here presented when migrating to different environments where migration speed could be the bottleneck, and the impact of traditional live migration techniques (such as pre-copy or lazy migration) could have to speed up the process. We are also interested in evaluating the system on a more realistic use-case like an online gaming service, and integrate our work in an orchestrator tool to add support for distributed migration.

In particular, this work represents the first stepping stone towards efficient C/R for distributed container deployments or container clusters. In this scenario, a manager could easily restore the whole application state in the event of a node failure.

References

[1] CRIU Foundation. Criu - main page. https://criu.org/Main_Page, 2019.

[2] A. Tucker. Task migration at Google using CRIU. https://www.linuxplumbersconf.org/event/2/contributions/209/attachments/27/24/Task_Migration_at_Google_Using_CRIU.pdf, 2018.

[3] A. Reber. CRIU - TCP connection. https://criu.org/TCP_connection, 2019.

[4] Adrian Reber. Criu - checkpoint/restore in userspace. https://access.redhat.com/articles/2455211, 2016.

[5] Linux Programmer's Manual. ptrace - Process Tree. http://man7.org/linux/man-pages/man2/ptrace.2.html, 2019.

[6] A. Avagin. Criu - upstream kernel commits. https://criu.org/Upstream_kernel_commits, 2019.

[7] CRIU Foundation. CRIU - A project to implement checkpoint/restore functionality for Linux. https://github.com/checkpoint-restore/criu, 2019.

[8] CRIU Foudnation. CRIU - Checkpoint/Restore. https://criu.org/Checkpoint/Restore, 2017.

[9] S. Hykes. Introducing runC: a lightweight universal container runtime. https://iperf.fr/, 2017.

[10] P. Emelyanov. Tcp connection repair (v4). https://lwn.net/Articles/493983/, 2012.

[11] J. Corbet. TCP connection repair. https://lwn.net/Articles/495304/, 2012.

[12] The Netfilter Project. netfilter: firewalling, NAT, and packet managing for linux. https://www.netfilter.org/, 2020.

[13] VirtualBox User Manual. 6.7. Host-only networking. https://www.virtualbox.org/manual/ch06.html#network_hostonly, 2012.

[14] CRIU Foundation. Criu - external resources. https://criu.org/External_resources, 2020.

[15] CRIU Foundation. Criu - inheriting fds on restore. https://criu.org/Inheriting_FDs_on_restore, 2016.

[16] iPerf Foundation. iPerf - The ultimate speed test tool for TCP, UDP, and SCTP. https://iperf.fr/, 2017.

[17] V. Paxson, M. Allman, J. Chu, and Sargent W. Computing tcp's retransmission timer. RFC 6298, RFC Editor, June 2011.

[18] M. Kerrisk. Namespaces in operation (series). https://lwn.net/Articles/531114/, 2013.

[19] B. Barker. Autosave for research: Where to start with checkpoint/restart. 2014.

[20] Adrian Reber. Combining pre-copy and post-copy migration. https://lisas.de/~adrian/posts/2016-Oct-14-combining-pre-copy-and-post-copy-migration.html, 2016.

[21] A. Reber. Criu and the pid dance. 2019.

[22] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI05, page 273286, USA, 2005. USENIX Association.

[23] R. Stoyanov and M. J. K. Efficient live migration of linux containers. In *High Performance Computing*, pages 184–193, Cham, 2018. Springer International Publishing.

[24] Sindi M. and Williams J. R. Using container migration for hpc workloads resilience. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10, 2019.

[25] Zeynep Mavus and Pelin Angin. A secure model for efficient live migration of containers. volume 10, pages 21–44, 2019.