MASTER'S THESIS

Master of Science in Advanced Mathematics and Mathematical Engineering

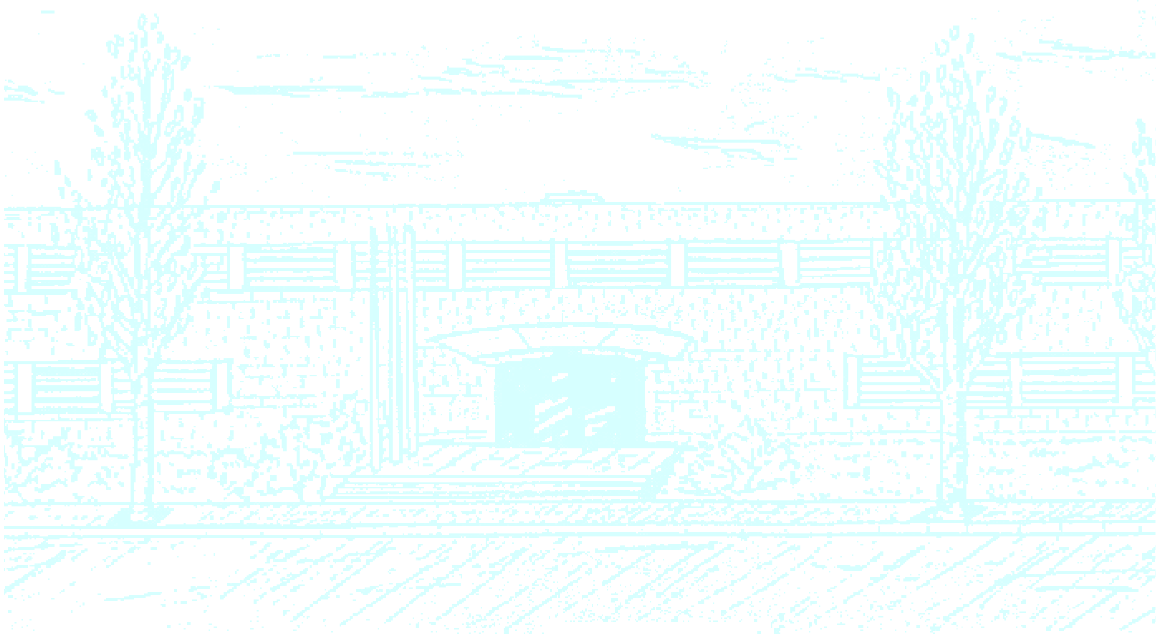
Title: Transparent Live Migration of Container Deployments in Userspace

Author: Carlos Segarra González

Advisor: Jordi Guitart Fernández

Department: Departament d'Arquitectura de Computadors

Academic year: 2019-2020



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Technical University of Catalonia

School of Mathematics and Statistics - FME UPC

Transparent Live Migration of Container Deployments in Userspace

Spring Semester - July 2020

Author:
Carlos Segarra González[1]
`carlos.segarra@estudiant.upc.edu`

Supervisor:
Jordi Guitart[1,2]
`jguitart@ac.upc.edu`

[1] Universitat Politècnica de Catalunya BarcelonaTech, Barcelona, Spain
[2] Barcelona Supercomputing Center, Barcelona, Spain

In partial fulfillment of the requirements for the
Master in Advanced Mathematics and Mathematical Engineering

Like most of my generation, I was brought up on the saying: 'Satan finds some mischief for idle hands to do.' Being a highly virtuous child, I believed all that I was told, and acquired a conscience which has kept me working hard down to the present moment. But although my conscience has controlled my actions, my opinions have undergone a revolution. I think that there is far too much work done in the world, that immense harm is caused by the belief that work is virtuous, and that what needs to be preached in modern industrial countries is quite different from what always has been preached. Everyone knows the story of the traveler in Naples who saw twelve beggars lying in the sun (it was before the days of Mussolini), and offered a lira to the laziest of them. Eleven of them jumped up to claim it, so he gave it to the twelfth. This traveler was on the right lines. But in countries which do not enjoy Mediterranean sunshine idleness is more difficult, and a great public propaganda will be required to inaugurate it. I hope that, after reading the following pages, the leaders of the YMCA will start a campaign to induce good young men to do nothing. If so, I shall not have lived in vain.

Bertrand Russell, *In Praise of Idleness*

Note from the Author

The work here presented is my Master's Thesis for the Master in Advanced Mathematics and Mathematical Engineering (MAMME), from the School of Mathematics in the Technical University of Catalonia (FME-UPC). It has been developed during a year-long collaboration with Jordi Guitart, from the Computer Architecture Department, who has lead the development and advised my research. Part of this work has been funded by a research collaboration grant *"Beca de col·laboració en departaments universitaris - Curs 2019-2020"* funded by the *"Ministerio de eduación y formación profesional"*.

Declaration of Authorship

I hereby declare that, except where specific reference is made to the work of others, this Master's thesis has been composed by me and it is based on my own work. None of the contents of this dissertation have been previously published nor submitted, in whole or in part, to any other examination in this or any other university.

Signed:

Date:

Acknowledgments

This work ends my Master in Advanced Mathematics and Mathematical Engineering and, at least for some years, my time at the Technical University of Catalonia. I must admit that it does not feel as special as almost a year ago, when I was submitting my Bachelor's thesis. It must be that I am getting used to it, or getting old, or maybe both.

In every personal success there's always a long list of people to be grateful, and I will try my best not to leave anyone out.

I will start the list with non-other than Jordi Guitart, my advisor. I cold-emailed him a year ago and in the subsequent months he: has been flexible to adapt to my, sometimes picky, preferences. Has helped me receive a grant to fund my work with him. Has helped me in the arduous task of finding a PhD. And overall has been a great advisor. On the same line, I would like to thank the *"Agencia de Gestión de Ayudas Universitarias y de Investigación"* for the project funding. Lastly, I must add some words for Juanjo Rué, the Master's coordinator. Right from the day I pre-enrolled he has always been welcoming, flexible, and responsive to all my concerns.

I would also like to take this moment to thank my parents for their relentless support, year in and year out. This work ends a six-year-long chapter of my life, and kickstarts a new one at a different university, different city, and different country. My last acknowledgment goes to those whose love I have learnt to appreciate thanks to a virus. And sure they know who they are.

Carlos Segarra González
Barcelona, June 25, 2020

Abstract

Transparent Live Migration of Container Deployments in Userspace

by Carlos Segarra González

Containers have become the go-to technology for managing application's lifecycle in the cloud. As a consequence, cloud-tenants are becoming increasingly interested in advanced load-balancing strategies to optimize resource usage and guarantee good quality of service. Live migration is a technique to halt the execution of a program and resume it in the same state in a different location without disrupting the program's availability. It relies on checkpoint-restore tools to snapshot an application's state. Checkpoint-Restore in Userspace (CRIU) is one of such tools, designed to work transparently to the user, entirely from userspace, and specialized for containers.

In this Master thesis we present a tool to perform live migration of `runC` containers using CRIU. Our solution is efficient in terms of resource utilization, memory and disk, and minimizes downtime when compared to naive migration through checkpoint-transfer-restore and native virtual machine (VM) migration. We also provide support to checkpoint memory and network intensive containers with established TCP connections and external namespaces. The implementation is accompanied by a thorough background research, together with a set of micro benchmarks to justify each of our design choices. It is open sourced and available in the project's repository.

Our evaluation results show that, by adding a very small overhead (0.1s to the baseline of checkpoint-transfer-restore) we improve scalability with regard to allocated memory by a factor of 10. Additionally, all our results are an order of magnitude faster than traditional virtual machine migration. Lastly, our benchmarking of network intensive server-side application's migration reported a < 0.1s throughput downtime, negligible with more moderate workloads. As a consequence, we believe our migration technique for CRIU and `runC` is a feasible replacement for VM migration and, as the technology matures, will be ready for deployment in production.

Keywords: checkpoint, restore, live migration, CRIU, runc, container, load-balancing

Resum

Transparent Live Migration of Container Deployments in Userspace

per Carlos Segarra González

Els contenidors de programari han esdevingut la tecnologia referent per gestionar aplicacions al núvol. Com a conseqüència, els principals proveïdors de servei estan cada vegada més interessats en solucions per gestionar dits contenidors, i oferir garanties de qualitat als seus usuaris. La migració d'aplicacions consisteix en aturar un procés i reiniciar-lo a un altre entorn d'execució en el mateix punt en el que s'havia aturat sense interrompre'n el procés d'execució. Es basa en la capacitat de generar captures de l'estat d'execució d'un procés. *Checkpoint-Restore in Userspace (CRIU)* és un projecte de codi obert que permet obtenir dites captures de manera transparent a l'usuari, sense modificar-ne el kernel, i especialitzada en contenidors.

En aquesta tèsis de Màster, presentem una eina per realitzar migracions de contenidors tipus `runC` emprant CRIU. La nostre solució és eficient en termes d'utilització de recursos, memòria i disc, i minimitza el temps de migració quan comparada amb una migració basada en capturar-transferir-reiniciar i amb la migració nativa de màquines virtuals oferida pels seus proveïdors. En afegit, la nostra eina permet migrar aplicacions que fan ús intensiu tant de memòria com de xarxa, amb connexions TCP establertes, i *namespaces* externs. La implementació està acompanyada d'una recerca bibliogràfica en profunditat, així com d'una sèrie d'experiments que motiven els nostres criteris de disseny. El codi és de lliure accés i es pot trobar a la pàgina web del projecte.

Els nostres resultats mostren que, afegint una petita redundància (0.1s al temps de referència de capturar-transferir-reiniciar) millorem l'escalabilitat del sistema en termes d'utilització de memòria en un factor 10. En afegit, tots els nostres resultats són un ordre de magnitud més ràpids que les migracions tradicionals de màquines virtuals. Per últim, els nostres experiments amb aplicacions que fan ús intensiu de xarxa mostren una caiguda del servei inferior als 0.1 segons, imperceptible per clients amb càrregues de treball més moderades. A mode de conclusió, creiem que la tècnica de migració que proposem en aquest projecte per CRIU i `runC` és una alternativa viable a la migració de màquines virtuals i, a mesura que la tecnologia maduri, estarà llesta per entorns de producció.

Paraules Clau: migracio, contenidor, CRIU, runc, captura d'estats, *checkpoint*

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API	Application Programing Interface.
BLCR	Berkeley Lab Checkpoint/Restart.
CRIU	Checkpoint Restore in Userspace.
DMTCP	Distributed Multi-Threaded CheckPointing.
GID	Group Identifier.
HPC	High Performance Computing.
ISA	Instruction Set Architecture.
KVM	Kernel-based Virtual Machine.
LXC	Linux Containers.
NFS	Network File System.
OCI	Open Container Initiative.
OS	Operating System.
PID	Process ID.
PIE	Position Independent Code.
PTE	Page Table Entry.
QoS	Quality of Service.
RPC	Remote Procedure Call.

TCP Transmission Control Protocol.

UID User Identifier.

VM Virtual Machine.

VMM Virtual Machine Monitor.

Chapter 1

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. Having the ability to manage workloads efficiently, cloud providers can provide high-priority tasks the resources they demand, and starve less important ones until other resources become available. Additionally, sharing and managing physical resources utilization, in particular ensuring that there are no severe imbalance among computing nodes, yet only the necessary ones are not idle, has a direct impact on energy savings for the data center. The virtual machine (VM) placement problem [1, 2] has studied this same issue for decades, the appearance of containers includes yet another variable to the optimization task.

Checkpoint-Restore is, through live migration, a technique to provide application-level 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. Originally developed for the High Performance Computing (HPC) domain, checkpointing was used to save intermediate long-running job's state. In the event of an unexpected failure, the job could be restarted from the last stored checkpoint, rather than restart and effectively lose several hours or days worth of work [3]. Checkpoint-Restore in Userspace (CRIU) [4] is a software tool to dump and restore processes transparently to the user. It does so entirely from userspace, by strongly leveraging interfaces exposed by the Linux kernel. If such interfaces do not exist, the project's contributors have a long history of accepted kernel patches [5]. CRIU is an open-source project and it targets specially applications running inside containers. As of June 2020, most container engines offering checkpoint-restore functionalities such as DOCKER, PODMAN, or LXC, rely on CRIU at a lower level. In the cloud-computing and load-balancing domain, the project is used in a variety of companies such as Google [6] within their Borg project [7].

This Master thesis is an initial approach to efficient transparent live migration of container deployments from userspace using CRIU. We study the different tools to checkpoint and restore containers and their integration with different container engines. Then, we provide a library implementing live migration of running and connected containers transparently to the end user using CRIU and `runC` [8] as our container runtime of choice. Our implementation is very easy to use, has minimal dependencies, requires minimal set up, and differs from other existing solutions [9] in the fact that no listening process needs to be running in the remote end. We support diskless, iterative (pre-copy) migration of memory intensive containers with established TCP connections and external namespaces. Moreover, we back all of our design choices with an extensive evaluation in the form of micro and macro benchmarks and a comparison with traditional virtual machine migration. Our system is open-source, still under development, and available at `https://github.com/live-containers/live-migration`.

1.1 Objectives, Tasks, and Contributions

The main goal of this work is to implement efficient live migration of running containers. The terms *efficient* and *live* are vague in the absence of concise metrics, and the variety of running containers is also huge, as a consequence we specify a set of objectives we want our system to fulfill. In particular, our key metric of success is downtime. Downtime measures for how long a migrated application is not running, and as a consequence it is a direct indicator of liveness. Additionally, we measure efficiency as the (lack of) redundancy and overhead our system introduces, usually in terms of allocated (and duplicated) memory and disk usage. We evaluate it on absolute terms, and also relatively when compared with virtual machine migration and native (manual) container migration. Our main objectives for the project can be summarized in the following list:

O1 Implement a fully-featured live migration library for containers.

O2 Support memory-intensive server-oriented containers.

O3 Have live migration be: efficient, live, transparent, and easy to use.

In order to achieve **O1**, **O2**, and **O3**, we define a series of tasks our implementation presented in §4 must fulfil. Note that these objectives and tasks are only implementation-oriented. Part of the contribution of this work is also bibliographical in the sense that we cover all the relevant material that helps us in the process of achieving the objectives. These non-tangible tasks and objectives are included in the final list of contributions.

T1 Implement support to migrate interactive, memory-intensive, containers.

T2 Implement support to migrate containers with established TCP connections and external namespaces.

T3 Minimize downtime and resource utilization by using diskless and iterative migration techniques.

T4 Motivate each of our design choices with detailed micro-benchmarks to ensure we are aligned with **O3**.

Contributions

To put it in a nutshell, the main contributions of the work here presented are listed beneath:

C1 An exhaustive micro-benchmark of different CRIU features, their performance, and their integration with `runC`.

C2 An open-source library for live migration of `runC` containers using CRIU.

C3 An easy to use binary to transparently migrate containers from one host to another with minimal dependencies and set up.

C4 An evaluation of our solution and a comparison with virtual machine migration.

1.2 Project Structure

The structure of the rest of this document is as follows. In Chapter 2 we introduce the foundational background concepts required to understand our contributions. We do a deep dive in the topics of containers (§2.1), checkpointing (§2.2), and CRIU in particular (§2.3). Note that `runC` is covered in detail when discussing different container engines and container runtimes.

In Chapter 3, we cover relevant bibliography and related work. As the goal of a Master's thesis is educational in nature, we have decided to include not only other scientific contributions related to container checkpointing and live container migration, but also all the bibliographical material that we have used. These references are both informative and educational, and will enable an interested reader to follow our same learning process which, in a document of this sort, is not to be underestimated. In particular, we cover relevant bibliography on containers (§3.1), CRIU and C/R (§3.2), and applications in live migration (§3.3).

In Chapter 4 we present the building blocks of our system (§4.1): diskless migration, iterative migration, and TCP sockets and namespace migration. For each concept, we present it's underlying theory together with a shell script to leverage it, it's implementation details in CRIU, and the integration with `runC`. Additionally, we micro-benchmark it's functionality comparing the vanilla performance with CRIU's and `runC`'s. Then, in §4.2 we cover in detail the implementation details of our solution, covering the most relevant code snippets, motivating our design choices (mostly with results from the previous section) and covering some helper modules we also implemented.

In Chapter 5 we put our system to work and, instead of micro-benchmarking particular features, we evaluate it as a whole through two different macro-benchmarks. First, in §5.1, we assess the impact different design choices have on the overall application downtime. Downtime is the key metric in live migration as it assesses the time the application is not running. Minimizing it is, in turn, the ultimate goal of efficient live migration. Secondly, in §5.2, we measure our system's scalability with regard to the memory allocated by the container. If downtime is the key metric to optimize, memory dumps and network latency are the two biggest bottlenecks. Efficient network transport is out of the scope for this project, hence in this second macro-benchmark we focus on the efficiency of memory dumps.

Lastly, in Chapter 6, we cover the most relevant conclusions and lessons learnt from the project, together with future lines of work we would like to continue in the coming months.

Chapter 2

Background Concepts

The main goal of this project is to implement live migration of running containers. It builds on the concepts of containers, checkpointing, and its implementation in the CRIU project. This chapter provides a detailed introduction to these concepts as they are necessary to understand the contributions we present later on.

2.1 Containers

A Linux container is a set of isolated processes with a limited view of their environment. They build on traditional concepts of virtualization, and provide an alternative to virtual machines (VM). A container is usually faster, lighter, and more flexible than a VM. As a consequence they are becoming the technology of choice in multi-tenant settings, such as data centers.

2.1.1 An Introduction to Virtualization

Virtualization is a recurrent technique in systems design in computer science which aims to provide processes the illusion that they interact with a defined interface, hiding the real implementation behind. Some of the most relevant features facilitated by virtualization are: process isolation from other processes and the underlying system, fine-grained dynamic resource provisioning, multiple virtually dedicated subsystems on the same physical instance, among others. We classify virtualization techniques according to the type of interface being virtualized.

Emulation. Emulators allow applications written for a certain computer architecture to run on a different one. They do so by translating (*i.e.* virtualizing) the Instruction Set Architecture (ISA). An example of such a system is QEMU (`https://www.qemu.org/`).

Hardware Virtualization. Hardware virtualization interfaces a complete system which enables to run a fully-featured operating system within a different one. It has traditionally been one of the most user-friendly virtualization tools in the form of *virtual machines* such as the Linux-Kernel

Virtual Machine KVM (`https://www.linux-kvm.org/page/Main_Page`, VMWare Workstation (`https://www.vmware.com/`) or Oracle's VirtualBox (`https://www.virtualbox.org/`). We differentiate between full virtualization and paravirtualization. The former adds an hypervisor or virtual machine monitor (VMM) which creates the illusion of multiple virtual machines, which are multiplexed across the physical resources, and allow to run an *unmodified* guest OS. The latter modifies the guest OS' source code and replaces sensitive calls with *hypercalls*, which are direct calls to the hypervisor.

OS-level Virtualization. Operating System-level virtualization allows for multiple isolated userspace instances, called **containers** which share a single operating system. In comparison to traditional virtual machines, containers add little overhead, require minimal startup, and have a low resource requirement, these factors make them highly scalable. Containers have experienced an exponential increase in usage, specially with the advent of open-source highly-available container engines such as Docker (`https://www.docker.com/`), Linux Containers LXC (`https://linuxcontainers.org/`), Podman (`https://podman.io/`, among others. Given that the goal of this project is to perform efficient live migration of running containers, the following section provides further technical details on containerization.

2.1.2 Working Principles of Containers

As previously introduced, a Linux container is a set of processes that are isolated from the rest of the machine. To achieve this isolation, they rely on two kernel features: control groups and namespaces [10].

Namespaces

As greatly phrased by Michael Kerrisk in his series of articles on namespaces [11], the purpose of namespaces is to wrap a global system resource and abstract it in a way that each process within the namespace thinks it has its own isolated instance of such resource. As of Kernel 5.6, there are eight different types of namespaces, which we present together with a brief description in Table 2.1. In order to create a new namespace of a given type, we can follow two approaches. With the `clone` system call, we can create a new child process, in a similar way to `fork` but with higher control of what pieces of execution context are inherited [12]. More specifically, with the `unshare` system call we can unshare a namespace from it's parent process [13]. To join an existing namespace, we can use the `setns` syscall, which, given a file descriptor referring to a namespace, it links the calling process to it [14]. These operations require the `CAP_SYS_ADMIN` capability. In Listings 2.1 and 2.2 we include examples of usage of `unshare` and `setns` respectively.

Kind	Description
`mnt`	**Mount namespaces** provide isolation of the list of mount points seen by the process in each namespace instance. It allow processes to have their own root file system and mount and unmount file systems without affecting the rest of the system.
`pid`	The **process ID namespace** isolates the PID number space. This means that two processes in different PID namespaces can have the same identifier. It is very useful in container migration as it allows to restore the processes with the same PID they were dumped with regardless of whether that ID might be taken in the target machine or not.
`net`	**Network namespaces** provide isolation of the whole network stack. In particular network devices, interfaces, routing tables, iptables rules, and sockets.
`ipc`	The **Interprocess Communication namespace** provides isolation for POSIX semaphore queues, semaphore sets and shared memory segments.
`uts`	The **UNIX Time Sharing namespace** allows processes to set a host-name or domain name for that particular namespace without affecting the rest of the system.
`user`	**User namespaces** isolate security-related identifiers such as user and group identifiers (UID, GID) and capabilities. This allows for a process to have privileges within a certain namespace but not outside its scope.
`cgroup`	The **Control Group namespace** virtualizes the contents of `/proc/self/cgroup`. As a consequence, each different namespace has a different `cgroup` root directory.
`time`	The **time namespace** has been the latest addition to the group. Included in Kernel 5.6, it allows different namespaces to have different offsets to the system monotonic and boot-time clock.

Table 2.1: List of the different namespaces supported in Kernel 5.6. and a brief description of the isolation they provide.

Control Groups

Control groups (`cgroups`) are a resource management kernel feature that allows handling of processes in hierarchical groups. This way, fine-grained resource metering and limiting can be applied on a per-group basis. Typical resources monitored using this technique are memory, CPU usage, I/O network, among others.

These constraints are enforced through the usage of kernel subsystems. Each different subsystem, mapped to one of the resources to manage, has an independent hierarchy. Each process then belongs to exactly one group per subsystem. For instance, the memory control group keeps track of the pages used and imposes different limits for physical, kernel, and total memory.

```c
#define _GNU_SOURCE
#include <errno.h>
#include <sched.h>
#include <stdio.h>

// Unshare from parent namespace
int main(int argc, char *argv[])
{
    // Specify the required flags
    // (bit-wise or)
    flags = CLONE_NEWNET || CLONE_NEWPID;

    // Dettach from parent namespace
    if (unshare(flags) == -1)
    {
        perror("unshare failed");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Listing 2.1: Snippet to unshare the calling thread from a namespace using `unshare`.

```c
#define _GNU_SOURCE
#include <errno.h>
#include <sched.h>
#include <stdio.h>

// Attach calling process to an existing
// network namespace.
int main(int argc, char *argv[])
{
    // Get namespace FD
    fd = open("/proc/330/ns/net", O_RDONLY);

    // Join the namespace
    if (setns(fd, CLONE_NEWNET) == -1)
    {
        perror("setns failed");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Listing 2.2: Snippet to attach to an existing network namespace.

Container Terminology

With the rapid increase in popularity, a wide-range of terminology has also been introduced in the container ecosystem. These terms are commonly misused [15] and, even though they don't cover the technical principles, are useful to differentiate the services different tools offer.

A *container* is the (set of) isolated Linux process. It is a running instance of a *container image*, a (set of) files that are used locally as a mount point. To enhance portability and vendor interoperability, images are stored using a standardized format by the Open Container Initiative (OCI, `https://opencontainers.org/`), an open governance structure for container-related standards. The *container engine* turns the image into a running container and acts as the interaction point with the user. However, engines don't tend to actually instantiate the containers themselves, and rather rely on a *container runtime*. The runtime is the lower level component that interacts with the kernel, its specification [16] is also maintained and developed by the OCI. `runC` is its reference implementation, and our tool of choice to implement live migration. We have chosen to skip the engine layer and interact with the runtime as support for advanced CRIU features is lacking in higher-level tools (more details on CRIU are presented in §2.3).

Different Container Engines and Runtimes

There are a variety of container engines available. `Docker` was the company that started exploiting containers commercially. It is most likely the best-known tool and the main responsible for the rapid adoption of the technology. However, there are several alternatives.

Given its pivotal importance, we differentiate between container engines that rely on `runC` as their runtime of choice, and those which don't.

runC-based Container Engines.

Other than Docker, cri-O (https://cri-o.io/) is a container engine focused on the integration of lightweight containers with the Kubernetes orchestrator. Podman (https://podman.io/getting-started/) is another alternative to Docker. Its main distinguishing factor is that it is a *daemonless* container engine. Lastly, rkt (https://github.com/rkt/rkt) is a project with pluggability, interoperability, and customization in mind. Unfortunately, it has reached end-of-life and is not currently maintained.

Non-runC Container Engines.

Katacontainers (https://katacontainers.io/) is a container runtime that runs on lightweight virtual machines. This is, instead of relying on standard container isolation techniques, they use VM-native isolation (hardware-backed) to provide further security guarantees. In particular, each container is hypervisor-isolated, meaning two different containers have different kernels. crun (https://github.com/containers/crun) is a re-implementation of runC in C. Its main focus is on performance and reduced memory footprint. In both cases crun surpasses runC. railcar (https://github.com/oracle/railcar) is another runtime implementation born from the idea that Go might not be the best programming language to implement a container runtime. With memory safety in mind, Oracle, the company behind railcar, decided to implement an OCI-compliant runtime in Rust. Unfortunately, the project is nowadays archived.

runC: the reference runtime implementation

Originally developed at Docker, runC is a lightweight container runtime aimed to provide low-level interaction with containers. In 2015 [8], 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 releases 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. The pair `config.json` and `rootfs` is referred to as OCI bundle.

2.2 Checkpointing

Imagine a long-running job in a cluster or in the cloud. Several hours into execution, the job unreasonably fails. Even in bug-free software, programs may crash from time to time due to, for instance, hardware failures. This happens even more in multi-tenant environments where different users are sharing the same physical resources [3].

Losing hours worth of computation is not only a loss of time for developers and scientists, but also a loss of money. A possible solution would be programatically saving data every certain time, or every certain number of iterations. This approach requires additional work by the developer, who has to implement not only the saving procedure, but the resuming one, in the event the application needs to be started from an intermediate state. Alternatively, highly parallel workloads could run processes separately on different chunks of data, and aggregate results afterwards. However, these solutions are *ad-hoc*, error-prone, and most importantly require additional work from the developer.

2.2.1 Checkpoint/Restore

Checkpointing refers to the ability of storing the state of a computation such that it can be continued later at that same state without covering the preceding ones. The saved state is called a *checkpoint* and the resumed process a *restart* or *restore* [17]. It provides systems with additional fault-tolerance and fast rollback times. C/R tools snapshot an application's state regardless of the software running and without requiring, in general, any additional work from the application developer. Even though they originated in the High Performance Computing environment, these tools are also useful for debugging, skipping long initialization times, and, as in this work, live process migration.

During checkpoint, and in order to save the process' state, all the essential information such as the program's memory, file descriptors, sockets and pipes, among others are dumped. In the distributed scenario, additional logic is required to coordinate the checkpoint across all processes [18].

Checkpoint-Restore became popular in the setting of virtual machines [19], but had already been thoroughly studied in the context of rollback and recovery strategies [20]. VM checkpointing is easier when compared to arbitrary process checkpointing as VMs are already isolated. Nowadays, and other than CRIU which is our tool of choice, there are several mature C/R projects that checkpoint virtually any running process transparently to the user. A comparison among some of them is presented later in §2.3.

2.2.2 Live Migration

A prominent application of Checkpoint/Restore is live migration. Live migration allows moving a running process from a physical host to another with negligible downtime and transparently to the end user. It is clear that live migration is a desirable feature for cloud tenants as it drastically increases their load-balancing capabilities with minimal impact to the perceived quality of service.

A naive approach would checkpoint the process in one host, transfer the checkpoint dump through the network, and restore it later on the other host. Unfortunately, this approach incurs in very high downtimes. Other more refined and more popular mechanisms minimize this downtime, we highlight *pre-copy* and *post-copy* migration.

Pre-Copy Migration.

Pre-copy migration, or sometimes called iterative migration, is a live migration technique that transfers most of the checkpoint information previously to stopping the running process, and only stops it once the information to transfer to the other end is minimal, achieving a very low downtime [21]. As memory pages tend to be the largest resource to dump and transfer, most pre-copy implementations iteratively transfer the memory that has changed between subsequent iterations. Additionally, and depending on whether the different nodes share a common file system or not, files are also incrementally dumped. When the information to transfer is lower than a specified threshold, the application is stopped in one end, the remaining bits sent over the wire, and resumed in the other one.

Post-Copy Migration.

Post-copy migration follows a radically different approach. Initially, it transfers the minimal information for the process to be able to resume on the destination host [22]. Then, page faults in the new host are resolved over the network, sending a request for a page that was not sent in the initial batch. This approach tends to be faster (lower total migration time) than pre-copy, but incurs in service degradation as page faults become extremely costly. In the literature, post-copy migration is also referred to as **lazy migration**.

2.2.3 Distributed Checkpointing

A less explored area is that of checkpointing a distributed application as a whole. This is, given a process running in different physical hosts, how to coordinate the checkpoint in order to get a consistent execution state. Reaching consensus among a set of distributed processes is a well-studied topic in the distributed systems field, and an existing algorithm for distributed snapshotting was presented in 1985 by Chandy and Lamport [23]. Since then, more and more algorithms have been presented [18, 24] optimizing certain aspects of the process. However, transparent distributed C/R is yet to be implemented in a software tool. The work here presented aims to be a step in this direction by facilitating live migration of containers with established TCP connections and external namespaces (external in the sense that they are not created by the migrated process).

2.3 CRIU: Checkpoint Restore in Userspace

Checkpoint/Restore in Userspace (CRIU) is an open-source C/R tool [4]. Introduced in 2011, its distinctive feature is that it is mainly implemented in userspace, rather than in the kernel, by using existing interfaces [25]. One of the most important interface is `ptrace` [26], as CRIU relies on it for seizing the target process. For other interfaces, several patches have been pushed to the mainline kernel by CRIU developers [5]. The project is currently under active development [27], and its main focus is to support the checkpoint and migration of containers.

2.3.1 A Technical Overview on CRIU

The main goal of CRIU is to perform a snapshot of the current process' tree state to a set of image files, so that it can be later restored at that exact point in time, without reproducing the steps that led to it.

Checkpoint

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 [28]. However, before it can actually start, we need to ensure that the process does not change its state during checkpoint. This includes: opening file descriptors, changing sessions, or even producing new child processes [29]. To achieve this transparently, instead of sending a stop signal (which could affect the process' state) CRIU freezes tasks using `ptrace`'s `PTRACE_SEIZE` command [26]. In order to find all active tasks descendant of the parent PID, the `$pid` dumper iterates through each `/proc/$pid/task/` entry, recursively gathering threads and their children from `/proc/$pid/task/$tid/children`.

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. In order to dump the contents of memory and credentials, a novel technique is introduced, the **parasite code**.

The parasite code is a binary blob built as a position independent executable (PIE) for execution inside another process' address space. Its purpose is to execute CRIU calls from within the dumpee's task address space [30]. To achieve this goal, CRIU must:

1. Move the task into seized state calling `ptrace(PTRACE_SEIZE, ...)`. Note that the task is stopped without it noticing, hence not altering its state.

2. Inject an `mmap` syscall in the current stack's instruction pointer, and allocate memory for the whole code blob. At this stage, space for exchanging parameters and results is also allocated within the dumpee's process address space. CRIU is now ready to run parasite service routines.

3. The external dumping process retrieves information about the dumpee's address space through the parasite code either through *trap* mode (one command at a time) or *daemon* mode (in which the parasite behaves as a UNIX socket).

4. With information about used memory areas and important flags read from `/proc/$pid/smaps/` and `/proc/$pid/pagemap`, the parasite code transfers the actual content outside through a set of pipes, which in turn gets translated into image files.

Lastly, the target process is cured from the parasite by closing it, unmapping its allocated memory area, and reverting to the original frozen state.

Restore

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. In particular, and in order to be completely transparent, CRIU requires that the restored tasks have the same PID they had before dump. To achieve this goal, older versions of CRIU had to perform very time-sensitive and race condition-prone PID handling, what was referred to as the PID dance [31, 32]. Starting with kernel 5.3 and the new `clone3()` system call, it becomes now possible to clone a process and specify the desired PID for it [33].

Then CRIU restores all basic task resources such as file descriptors, namespaces, maps, ... The only resources that are *not* restored at this stage are, most notably, memory mappings. In order to restore memory areas, and since the morphing is done *in-place*, before exiting CRIU would have to `unmap` itself and map the application code. To overcome this issue, a similar approach to the parasite code one is followed, the **restorer blob**. The restorer blob is a piece of PIE code, to which CRIU transfers control to unmap itself and map the appropriate code and memory areas for the process to restore successfully.

Live Migration with CRIU

As CRIU operates by design on a single system, support for live migration requires further user interaction [34]. In particular, it is up to the user to ensure that the dump files are on the remote host upon restore. Furthermore, IP addresses used by the application in the original host, must also be available in the new one. With that said, CRIU developers implemented P.Haul [35], a library specially targeted for live migration using CRIU. At the time of the writing, the project is currently inactive.

The most natural way to manually perform a live migration is to use the iterative approach as we will cover later. However, support for lazy migration and a page server is also available [36]. A major drawback with iterative migration is that, as explained before, CRIU freezes the process while the snapshot takes place. As a consequence, recurrent snapshots of a memory intensive application might cause it to freeze during long periods of time. Lastly, and in order to prevent file duplication, it is encouraged to use a technique called diskless migration [37] - which we will cover in detail in §4.

2.3.2 Comparison with Other C/R Tools

The main differences between C/R tools are the way they interact with the kernel and the type of applications they target. CRIU is implemented completely in userspace, and as a consequence relies heavily on existing kernel interfaces, otherwise execution fails. Additionally, CRIU's target application are containers.

Other open-source tools that implement C/R are `DMTC` [38], and `BLCR` [39]. They both focus on high performance computing, what motivates some of their design choices.

DMTCP.

Distributed Multi-Threaded Checkpointing (DMTCP) is an active project lead by Prof. Cooperman at Northeastern University that implements C/R on a library level. This means that if a user wants to checkpoint its application, this must be dynamically linked from the very beginning and executed with custom wrappers (which decreases transparency). DMTCP intercepts all system calls instead of assuming existing kernel interfaces, as CRIU does, and is, as a consequence, more robust and reliable. It is very popular in HPC environments and is present in a variety of publications [40, 41].

BLCR

Berkeley Lab Checkpoint/Restart (BLCR) is a system-level checkpointing tool aimed also at High Performance Computing jobs. It requires loading an additional kernel module and is currently not maintained (last supported kernel version is 3.7).

A detailed table comparing the software here presented, and some other solutions, is maintained by the CRIU foundation [42].

Chapter 3

Related Work

In this chapter we introduce the most relevant pieces of related work we have based our work on, together with similar approaches to tackle live migration of processes or containers. We also include, given the educational nature of this work, references on the bibliography we have based our claims on, together with the materials used throughout our learning process as we understand it is relevant in the frame of a Master's thesis.

3.1 Containers: Overview, Internals, and Terminology

The main goal of this work is to perform efficient live migration of running containers. As a consequence, understanding the working principles of the latter is of prominent importance. Nowadays, there are dozens of articles covering containers available online but most either confuse general concepts with particular implementations or misuse terminology. The first problem becomes apparent when, given the widespread use of `Docker` as a container engine, one can wrongly assume that DOCKER containers are the only sort of containers. The second one stems from the fact that containers are a relatively new technology (less than 10 years of usage) and their governing body (Open Container Initiative, OCI) is still in the process of establishing itself. As a consequence there is a lack of formalism in the definition of terms like: *container*, *container image*, *container registry*, among several others.

A great article from 2018 by McCarty [15] published in the RedHat blog aims to provide OCI-based definitions on terms like: *container*, *image*, *image layer*, *tag*, *base image*, and *layer*. It also covers different container engines and container runtimes. Fortunately, there's already much more to that than `Docker` and `runc`. For a technical introduction to the working principles of containers (namely namespaces) we have leveraged a series of articles on LWN by Michael Kerrisk [11]. They first cover the patch history of different namespaces right at the same time user's namespaces where merged into the mainline kernel (Linux 3.8). Then, it introduces the different namespaces available to that date: mount namespaces, UTS namespaces, IPC namespaces, PID namespaces, network namespaces, and user namespaces. To each one of these, the author also devotes a complete article and includes snippets to give practical examples of different use cases.

Whenever we deal with system-related tools or calls, the manual pages themselves offer great resources of information, albeit sometimes quite advanced. In particular, we make use of and cite the manual pages for namespaces [10], the `clone` system call [12] and the `setns` one [14]. Lastly, for both namespaces and control groups we have also used a set of slides on *Process Virtualization* from the *Operating Systems* course of the Master in Innovation and Research in Informatics (MIRI) from the Technical University of Catalonia. The contents of this course are not available for open distribution and were crafted by the course instructor, Jordi Guitart, who is at the same time the advisor for this work.

With regard to `runC`, the OCI reference implementation of a container runtime, we would highly recommend the introductory post by Solomon Hykes, `Docker`'s founder and former CEO. `runC` was originally a proprietary component of the `Docker`'s stack, but was open-sourced in 2015 and donated to the Open Container Initiative. In this article, Hykes introduces what exactly is `runC`, the motivation behind it and the new governance model. The project is very active on GitHub [43], and the different issues there posted together with the available documentation (from the repository itself) are the best source of information for the project. Lastly, `runC` is on track with the OCI container runtime specification, which is also available on GitHub [44].

3.2 Checkpoint Restore and CRIU

Checkpoint-Restore being a technique (application checkpointing) rather than a tool, makes the topic that much vague for its research. A good starting point is the definition in the Encyclopedia of Parallel Computing [17]. We have also greatly leveraged a set of slides by Brandon Barker from Cornell University [3]. There, the author does a non-scientific introduction and motivation for C/R, and goes on to cover the different available tools as of December 2014. Albeit slightly old, most of the works he cites (DMTCP, CRIU, and BLCR) are still the *de facto* alternatives when doing application checkpointing. The author's approach is biased towards high performance computing, where DMTCP [38] is the usual software of choice, but it does a great job at pointing out the differences between each solution. For a detailed survey on the origins of C/R in the context of rollback recovery strategies for fault-tolerant systems, we highlight the work by Elnozahy *et al.* from 2002 [20]. The techniques there described started gaining traction with Virtual Machine's migration, a topic for which Clark's survey is also a great source of information [19].

We chose to use CRIU as our C/R tool, as it was the most suitable one in the container scenario, and was already used by the major container engines and runtimes (although with different degrees of integration and active maintenance). Checkpoint-Restore in Userspace [4] is an open-source community-driven project. Therefore, it has a very actively maintained wiki covering all related topics. An exhaustive list of these topics is also available [45]. Adrian Reber is a maintainer of the project in charge of, among others, part of the integration with `runC`, and has a set of very interesting an easy-to-follow articles on CRIU. The earliest ones from 2016 cover the tool as a whole [25], and different types of available migrations [46]. One of the most delicate parts of process restore is how to handle old, new, and dependent process identifiers (PIDs). Reber also has an article describing how this is done in CRIU [31]. For our technical dive on the tool's internals in §2, we relied mostly on the wiki articles. In particular we would like to highlight the one covering checkpoint-restore in a broad sense [28]. From it, the reader can easily jump to other linked articles if needed. We also read in detail the articles covering diskless migration [37], live migration [34], and the memory tracking ones both in the wiki [47] and in LWN [48]. Lastly, for the C/R of established TCP connections, we highly recommend the article by Corbet from 2012 [49], as it covers everything one needs to know to understand the approach followed in CRIU.

When comparing different C/R tools, and in addition to the previously mentioned work by Barker [3], CRIU's developers themselves maintain a comparative table where they list the pro's and con's of each different tool [42] (namely CRIU, DMTCP, and BLCR). In spite of the natural bias they may have, the resource has plenty of detail and is of great use. The main alternative to CRIU for C/R is the Distributed-MultiThreaded Checkpointing project [40] (DMTCP). Developed under the supervision of professor Gene Cooperman from Northeastern University, the project has a long-standing record of successes in the high performance computing domain, being the tool of choice by several national laboratories in the US. We based on their home page [38] to complete our section comparing them. Additionally, the Berkeley Lab's Checkpoint-Restart [39] (BLCR) is also an HPC-focused tool, although it has lost some traction during the last years.

A stretch goal for this project was to implement live migration of distributed container deployments, for which distributed checkpointing algorithms would be crucial. Even though we have not had time to address the implementation of such a concept, we have used several well-established resources for documentation purposes. We would like to highlight the works by Raynal [18] and Kshemkalyani [24].

3.3 Applications of C/R and Live Migration

Even though C/R and live migration are a relatively mature topic of research, scientific contributions covering particular applications are way more scarce. This was, among others, one of our initial motivations for this work. Some of the earlier pieces of research stem from the same research group developing DMTCP. Being HPC the target community for the project, one of the most popular applications of DMTCP is checkpointing of MPI applications. The first references date from 2002 [50], but the first contribution from the DMTPC team did not appear until 2016 in the work by Arya *et al.* [51], which finally resulted in *MANA for MPI: MPI-Agnostic Network-Agnostic*

Transparent Checkpointing [41]. This last piece of work summarizes all the previous contributions as it allows for transparent checkpointing of any MPI implementation and network combination. The current lines of research focus on proving this approach's scalability and GPU checkpointing.

For application-oriented projects leveraging CRIU we would like to highlight the work by Venkatesh [52]. This contribution is relevant to our work as it focuses on fast and efficient checkpoint-restore for DOCKER containers. In particular, the authors present an optimization to the file-based image procedure using the new (as of 2019) kernel support for multiple independent virtual addresses space (MVAS). We can not leverage the findings in our project as it only focuses on single-machine C/R.

Another interesting application of CRIU for DOCKER container migration is the recent article by Antonio Barbalace *et al.* [53]. In this work, the authors aim to overcome the limitation in migration for edge computing imposed by the different instruction set architectures (ISAs) each node may have. To achieve this goal they rely on containerization and CRIU. Our work focuses on live migration of server-side oriented services, hence why the edge computing use case is not directly applicable. Similarly, the work by Machen [54] also targets the edge computing scenario. In particular, the author presents a layered framework to perform live migration of services in mobile edge clouds. These services can be encapsulated in either virtual machines or containers within VMs, and the authors rely on native VM migration with LXC and KVM.

A contribution to CRIU which stemmed from an application use case and which we could leverage in the project was presented by Stoyanov in 2018 [55]. The author optimizes downtime during container live migration by utilizing CRIU's newly added feature: the image cache/proxy. Another article covering efficient live migration of DOCKER containers was presented in late 2019 by Zeynep and Pelin [56]. The authors focus on the support for C/R implemented in DOCKER and focus on securing the migration process to protect against potential attacks. Although the security approach is novel in nature, the fact that the authors use the outdated and not-maintained DOCKER integration of CRIU's C/R makes it not-reusable for our project.

Also in the HPC domain, but focused on container migration, lie the pieces of work by Berg [57], and Sindi [58]. The former follows more of a survey-like style, in which the authors proof the feasibility of using C/R for containers in HPC. Similarly, the latter showcases different applications of CRIU's migration capabilities in HPC. In particular, the authors present a migration of an MPI application, although they don't compare it with the work here described previously.

Lastly, the contribution that most closely matches our goal of providing an efficient, transparent, easy-to-use migration library for running containers is the Process Hauler (P.HAUL) project [59, 9]. Initiated by the same CRIU developers, the work was an early attempt to wrap all the technical details behind efficient live migration and deliver it as a solution to the end user. Unfortunately, the development stopped in late 2017, what greatly motivated this work.

Chapter 4

Implementing Efficient Live Migration

In this chapter we present our implementation of live migration of `runC` containers using CRIU. In order to achieve efficiency, liveness, and mimic a realistic setting, we explore disk-less and iterative migrations, and checkpointing established TCP connections and external namespaces. First, in §4.1, we cover the implementation of each of these features in CRIU and their integration with `runC`. Then, we perform a set of micro-benchmarks to assess their impact on performance, and finish with a snippet showcasing their usage. Lastly, in §4.2, we provide insights on our final open source implementation available at `https://github.com/live-containers/live-migration`.

4.1 Building Blocks

In this first section we study the implementation of diskless and iterative migration in CRIU. The former allows fast checkpoint/restore without writing to disk. The latter allows for incremental dumps, which in turn reduces downtime when migrating an application as the load can be divided among subsequent dumps. We also introduce how to checkpoint and migrate established TCP connections and established namespaces.

For each different feature, we prepare a set of experiments. Unless otherwise stated, we run each one in a Debian machine with kernel version `4.19.0-6` and use CRIU version 3.13 and `runC` version `1.0.0-rc8`, both built from source.

4.1.1 Diskless Migration

As previously detailed, CRIU builds the snapshot of a running process using image (`.img`) files, which are stored in a user-specified path. As a consequence, it relies heavily on the underlying storage facility provided which, in most commodity PCs, tends to be the disk-backed file system. It is of no surprise then, that reading and writing from and to disk can quickly become the bottleneck in live migration's performance. It gets even worse when writes are duplicated, *i.e.* we write once to disk to dump the process state, and a second time to transfer image files wherever they need to be restored. To overcome the former, we rely on `tmpfs` a virtual memory file system [60]. For the latter, we leverage CRIU's `page-server`.

First presented by Sun Microsystems in 2007 [61], `tmpfs` is a memory-based file system that uses resources from the virtual memory subsystem. According to the Linux Programmer's Manual [60], this file system can employ swap space if memory pressure is high, only consumes as much memory as required to store the current files (regardless of the allocated size), and unmounting it destroys the contents therein. Since the files actually reside in memory, the user benefits from memory-like read/write performance. A notable use of `tmpfs` is `/dev/shm`, used in the POSIX-compliant implementation of shared memory and in POSIX semaphores. One such file system can be easily created and destroyed using `mount` and `umount` as detailed in Listing 4.1

```bash
#!/bin/bash
# Mount a tmpfs file system rooted in the /tmp/my-tmpfs directory with maximum size 100 MB
mkdir /tmp/my-tmpfs
sudo mount -t tmpfs -o size=100M tmpfs /tmp/my-tmpfs

# Check the new file system appears in the list of mounted devices
sudo mount | grep /tmp/my-tmpfs

# Unmount the file system
sudo umount /tmp/my-tmpfs # CAUTION: THIS WILL DESTROY THE CONTENTS
```

Listing 4.1: Mounting and dismounting a `tmpfs` file system.

The page server is a component of CRIU that allows to send memory dumps directly through the network, saving disk read/writes on the origin, writing them once they reach the destination system [62]. Note that the page server is used only to migrate memory files, which tend to be the largest ones, whereas other image files still need to be transferred when migrating. The current implementation uses only TCP sockets and no encryption nor compression is used on the network transfer. It is also worth mentioning that `criu page-server --port` is a one-shot command, *i.e.* if we perform multiple dumps, a page server must be started for each one of them. Observe that, even though it introduces a small overhead, our results (see Figure 4.1) show that for migrations within the same machine, setting up a page server in `localhost` outperforms the double-copying approach for larger applications.

As introduced in the previous paragraphs, the key pieces to achieve efficient diskless migrations are making use of a `tmpfs` file system and CRIU's page server. The former is in another level of abstraction than `runC`, and for the latter we need to start the page server separately and then checkpoint passing address and port as a parameter to the `--page-server` flag. In Listings 4.2 and 4.3 we include snippets to perform a checkpoint with a page server using `runC` and CRIU respectively.

In order to benchmark the performance of diskless migration when compared to disk-based one and the benefits of using a page server, we set up two different experiments. In one hand we have a counter program written in C (see Listing A.1 for the full implementation) that increments a

```bash
#!/bin/bash
# Start the Page Server
sudo criu page-server \
    --port 9999 \
    --images-dir /path/to/dst/images &

# Checkpoint using the page-server
sudo runc checkpoint \
    --image-path /path/to/src/images \
    --page-server 127.0.0.1:9999 \
    <container_name>

# To finish the migration we would need to
# copy the remaining files
# This should be fast as memory dumps are
# already at destination
cp /path/to/src/images/* \
    /path/to/dst/images/
```

Listing 4.2: Commands to perform a checkpoint in `runC` using a page server.

```bash
#!/bin/bash
# Start the Page Server
sudo criu page-server \
    --port 9999 \
    --images-dir /path/to/dst/images &

# Checkpoint using the page-server
sudo runc checkpoint \
    --image-path /path/to/src/images \
    --page-server 127.0.0.1:9999 \
    <container_name>

# To finish the migration we would need to
# copy the remaining files
# This should be fast as memory dumps are
# already at destination
cp /path/to/src/images/* \
    /path/to/dst/images/
```

Listing 4.3: Commands to perform a checkpoint in CRIU using a page server.

value and prints it to `stdout`. On the other hand we have an instance of a Redis in-memory database that we pre-load with $1e7$ keys. The total weight of the memory image dump is 912 MB. For each experiment, we measure the time to checkpoint the process and transfer the remaining images to a different directory, either locally or on a different machine in the same local network. We compare the performance when using `tmpfs` directories to store the images (diskless) or not (file-based) and when using a page server or not. Each test is run 100 times and we present the average and standard deviation values obtained in Figure 4.1.

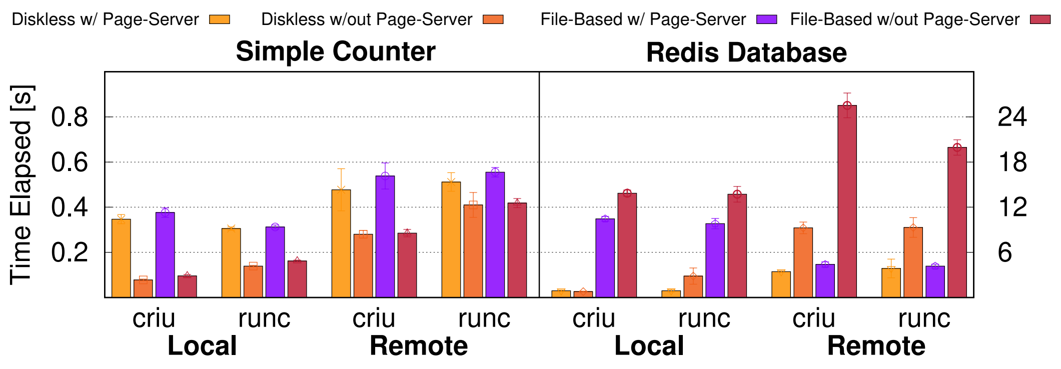


Figure 4.1: Time elapsed checkpointing and migrating a running process when using file-based or diskless migration, with and without a page server. We compare the results for a small application (around 100 kB, left) and a big one (around 1 GB, right).

.

The first and most important conclusion we draw from our results is that there is no one-size-fits-all solution when choosing the best setting to migrate our application. It seems clear that diskless is always equal or better than non-diskless. This was to be expected, as for the same setting, `tmpfs` gives better raw read/write performance. For instance, when transferring image files from one machine to the other, the perceived end-to-end throughput between `tmpfs` directories is in the order of 100-120 Mbps compared to the 60-70 Mbps for regular directories. However, there might be situations, or systems, which simply don't have that much free memory. The Redis dump files alone already take up 1 GB of memory, unacceptable in constrained devices.

If the application is sufficiently small (a dump for an instance of the counter process is around 90 kB), the overhead of running a page server is higher than simply writing the files twice, both in the local and remote setting. However, for large applications, diskless outweighs the page server in the local case, whereas if we have to send files over the network, running a page server is more important than using the diskless approach (although a combination of both yields the best performance). We include the full evaluation scripts in Listing B.1.

4.1.2 Iterative Migration

Implemented in CRIU we find a series of features that enable us to perform iterative migrations of running processes. This is, periodically snapshot the state of the process without altering it until some condition is triggered, that in turn checkpoints the container and restores it elsewhere. The key idea being that all the heavy work for the snapshot (*i.e.* capturing the memory state and transferring it) will have already been done in previous iterations, hence minimizing the application downtime.

In the previous paragraph we have assumed that transfers across consequent snapshots will be smaller in size, otherwise the n-th dump would not be any faster than the first one, and we would be wasting a lot of bandwidth since the same information would be sent repeatedly. This reduction in size can be achieved through memory tracking, a procedure through which memory pages written between dumps are marked as *dirty* and hence included in the following transfer. Therefore, to implement efficient iterative migration we need:

1. **Pre-Dump:** A procedure to snapshot the memory of the process without stopping it (note that, at this point, we don't need all the other details).

2. **Memory Tracking:** A procedure to keep track of the memory changes in a process' address space.

3. **Parent Directory:** A procedure to link together subsequent dumps so that they can be correctly re-interpreted at restore time.

The first step during an iterative migration consists on dumping *all* of the process memory to an image file. This allows for a baseline from which smaller *incremental* dumps are performed. Note that, at this point, we are not interested in capturing the whole state, hence the usage of the `pre-dump` command in CRIU.

Memory tracking in CRIU [47] is done by means of a kernel functionality introduced in 2013 [48]. It consists of two steps: first we ask the kernel to keep track of memory changes on a per-process basis by writing a 4 to `/proc/pid/clear_refs` and, after a while, reading the `/proc/pid/pagemap` file and checking the *soft-dirty* bit for each page table entry (PTE). Internally, in the first step the kernel clears all soft-dirty bits *and* the writable ones per each PTE for the given process id (PID). Subsequent writes to any page will trigger a page fault, a call to `pte_mkdirty`, and therefore the *soft-dirty* bit will be set. During the second step, at memory dump time, if this bit has not been set, the memory page needs not to be transferred again. To enable this functionality in CRIU, we must use the `--track-mem` flag.

One last key step required to achieve efficiency and correctness upon restore is to link the actual dump (or pre-dump) with the one preceding it, it's *parent*. For a pre-dump, `--prev-images-dir` indicates CRIU to look for existing dumps in the specified path, and perform the bit-checking described in the previous paragraphs. Upon restore, links among successive dumps are pieced together to successfully restore the freshest version of the running program.

The integration with `runC` is seamless. The pre-dump functionality is triggered with the `--pre-dump` flag which, in turn, sets the memory tracking flag automatically [43]. Lastly, the `--parent-path` flag can be used to achieve the correct linkage between dumps. In Listings 4.4 and 4.5 we include the different scripts to perform the three consecutive dumps both in CRIU and `runC`. The complete scripts used for the benchmarking are included in Listing B.2.

```bash
#!/bin/bash
# First Pre-Dump
sudo criu pre-dump \
    -t  PROCESS_PID \
    --images-dir images/1 \
    --track-mem \
    --shell-job

# Second Pre-Dump
sudo criu pre-dump \
    -t PROCESS_PID \
    --shell-job \
    --images-dir images/2 \
    --prev-images-dir ../1 \
    --track-mem

# Last Dump
sudo criu dump \
    -t  PROCESS_PID \
    --images-dir images/3 \
    --prev-images-dir ../2 \
    --shell-job \
    --track-mem

# Process is now stopped
```

Listing 4.4: Scripts to perform two pre-dumps and a dump of a running process using CRIU.

```bash
#!/bin/bash
# First Pre-Dump
sudo runc checkpoint \
    --pre-dump \
    --image-path ./images/1/ \
    <container_name>

sudo runc list # Container running

# Second Pre-Dump
sudo runc checkpoint \
    --pre-dump \
    --parent-path ../1/ \
    --image-path ./images/2/ \
    <container_name>

sudo runc list # Still running

# Last Dump
sudo runc checkpoint \
    --parent-path ../2/ \
    --image-path ./images/3/ \
    <container_name>

# Container is now stopped
```

Listing 4.5: Scripts to perform two pre-dumps and a dump of a running container using `runC`.

In order to perform a micro-benchmark of this functionality we consider two different scenarios: a simple counter written in C, and a Redis in-memory database, as introduced in the previous section. For each scenario we perform two pre-dumps, and a final dump, and report the size of the `pages-1.img` file (which contains the memory dump). We test a static setting in which we don't change the memory during successive dumps which acts as a baseline, and a dynamic one in which, between each dump, we modify the contents of the process memory. For the counter, the static setting starts the program and goes to sleep, whereas the dynamic one indeed updates the counter every other second. For the database, we initially pre-load it with $1e7$ key-value pairs (around 300 MB of data) and then either do nothing, or run a `redis-benchmark` which alters around 1% of the key pairs. Lastly, we compare the results of running the experiments with vanilla CRIU or through `runC`.

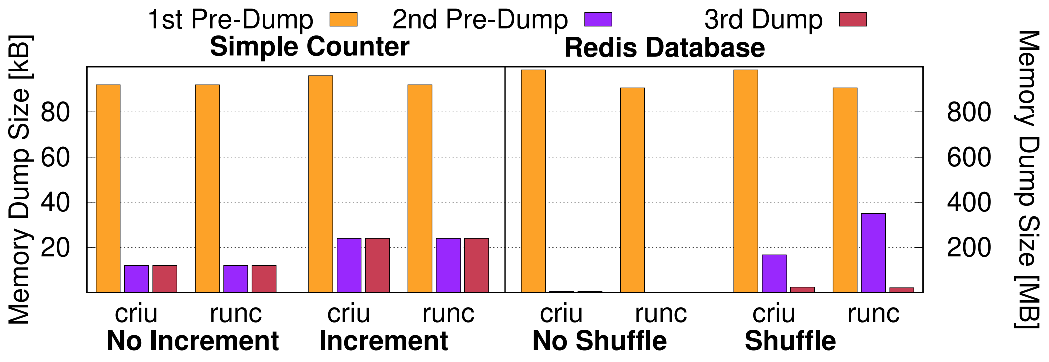


Figure 4.2: Size of the dumped memory image when performing iterative dumps. For the counter experiment we report the results in kB (left axis) and for the redis one we report the results in MB (right axis). We compare the results when using `runC` or purely CRIU.

.

We present our results in Figure 4.2. First of all, note how we use different scales for the counter application (left) and the Redis one (right). We observe that, as expected, if we make no changes to the process' memory after the first dump, the amount of information to be re-transferred is very little, which we attribute it to CRIU's metadata. In the counter case, the initial dump is around 90 kB and subsequent ones are 12 kB, whereas in the Redis one, the size decreases from 900 MB to just 1 MB. Once we modify the memory, additional pages need to be transferred. In the counter case, between successive dumps we just increase the value of a variable and alter the state of `stdout`, what translates in a 10 kB increase in the image size every time. In the Redis one, the `redis-benchmark` is non-deterministic in nature, but it's worth observing how shuffling a percent of the total key-store propagates to higher percentages of memory re-use. We conclude that memory tracking is a necessary feature if any application considers even near-live migration of production applications, and the technology presented allows for an easy way to do so.

4.1.3 Checkpointing TCP 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 [63].

Similarly to other resources and as introduced in §2, basic information about sockets is obtained by parsing the adequate files in the /proc file system. However, there are some internals of active network connections (namely negotiated 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 [49].

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 [64] IP routing table to drop all packets. We include the set of rules in Table 4.1.

```
Chain INPUT (policy ACCEPT)
target          prot   opt    source         dest
CRIU            all    --     <source_IP>    <dest_IP>

Chain FORWARD (policy ACCEPT)
target          prot   opt    source         dest

Chain OUTPUT (policy ACCEPT)
target          prot   opt    source         dest
CRIU            all    --     <source_IP>    <dest_IP>

Chain CRIU
target          prot   opt    source         dest
ACCEPT          all    --     <source_IP>    <dest_IP>     mark match !  0xc114
DROP            all    --     .../0          .../0
```

Table 4.1: Output of running iptables -t filter -L -n.

.

Efficient IP Address Re-Use

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 below), we need to assign addresses using `ip`'s `addr` subcommands. In particular, we are using a `host-only` [65] 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 4.3.

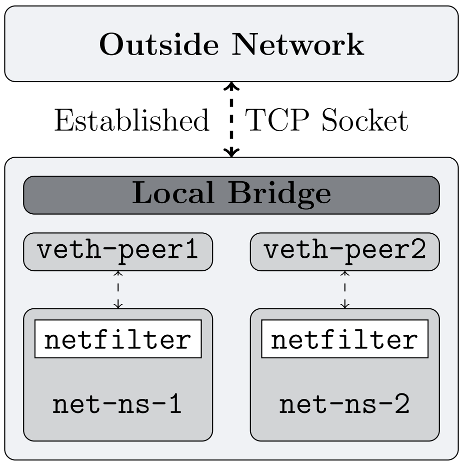


Figure 4.3: Architecture of three different namespaces connected through virtual ethernet pairs.

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 4.1 using the `iptables` command. Lastly, to restore into an existing namespace, the container must be restored with the adequate open file descriptors using CRIU's `--external` [66] and `--inherit-fd` [67]. In Listings 4.6 and 4.7 we include excerpts of snippets to checkpoint and restore an established TCP connection without or within a network namespace respectively. The complete scripts for the evaluation are included in Listings B.3 and B.4 for CRIU's downtime and reactivity experiments, and in Listings B.5 and B.6 for `runC`'s.

Benchmarking

```bash
#!/bin/bash
# CRIU Dump and Restore, one after the other
# but in the BG (not affecting time)
(sudo criu dump \
    -t `SERVER_PID` \
    --images-dir `IMAGES_DIR` \
    --tcp-established; \
echo "Restoring server..."; \
sudo criu restore \
    --images-dir `IMAGES_DIR` \
    --tcp-established) &

# Similarly with runC
(sudo runc checkpoint \
    --image-path `IMAGES_DIR` \
    --tcp-established \
    eureka; \
cd /container/directory; \
sudo runc restore \
    --image-path `IMAGES_DIR` \
    --tcp-established \
    eureka; \
cd `CWD`) &
```

Listing 4.6: Checkpoint and restore an established TCP connection using CRIU and `runC`.

```bash
#!/bin/bash
# Two namespaces with path NS_1 and NS_2
INO_1=$(ls -iL ${NS_1} | awk '{ print $1 }')
INO_2=$(ls -iL ${NS_2} | awk '{ print $1 }')
exec 33< ${NS_1}
exec 34< ${NS_2}

# To checkpoint we mark as an external
# resource both NS
sudo criu dump \
    -t ${PID_1} \
    --images-dir images \
    --tcp-established \
    --external net[${INO_1}]:${NS_1} \
    --external net[${INO_2}]:${NS_2}

# At restore, we match the file
# descriptors with the NS
sudo criu restore \
    --images-dir images \
    --tcp-established \
    --inherit-fd fd[33]:${NS_1} \
    --inherit-fd fd[34]:${NS_2} -d
```

Listing 4.7: Excerpt of a script to checkpoint a connection within an existing namespace.

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 [68]. 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 occurred. 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 4.4 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 those packets were never acknowledged, and hence will try to retransmit them. The TCP protocol specifies [69] that

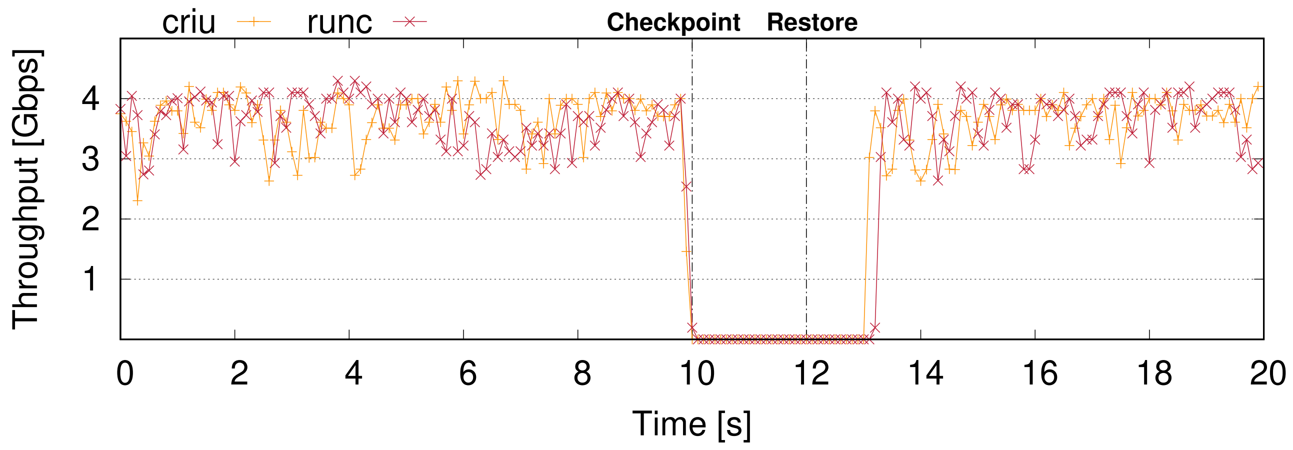


Figure 4.4: Throughput perceived from the client as a function of time, when we checkpoint the server once, and restore it after two seconds. We compare the results of CRIU and `runC`.

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 4.5. In this case

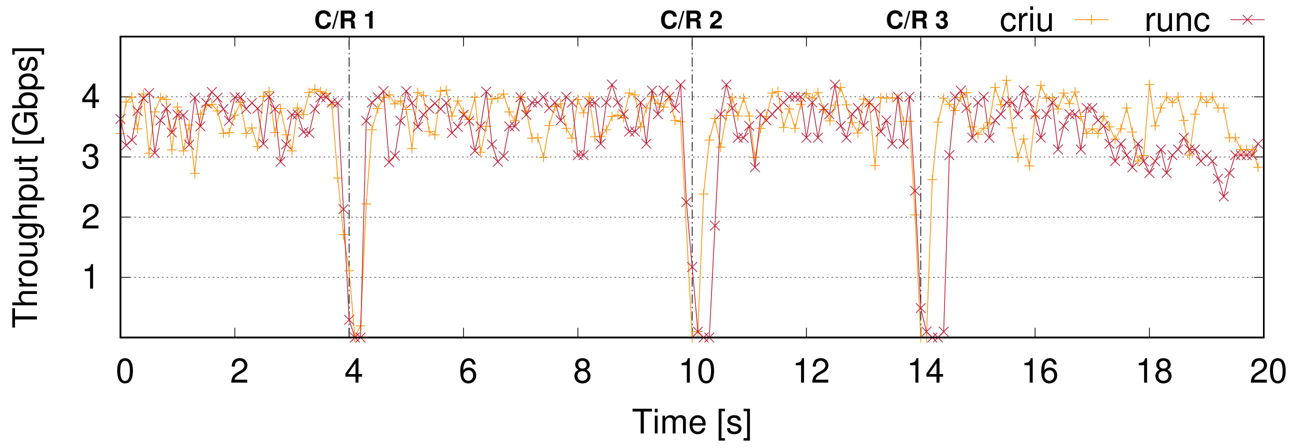


Figure 4.5: Throughput from the client as a function of time as we iteratively checkpoint and restore the service immediately after.

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.

4.2 Putting it All Together

In this section we cover our implementation of live migration of running `runC` containers using CRIU. As previously mentioned, the complete source code is available on Github: `https://github.com/live-containers/live-migration`. The implementation is fully in `C` and it amounts to around 1300 LoC. We chose `C` to have a cleaner interaction with the underlying system and a smoother integration with CRIU, which is also written in `C`. However, CRIU exposes all its services via an RPC client, hence it should be compatible with a variety of other programming languages. Additionally, all the namespace and ip tables handling can be done natively importing the adequate libraries. Lastly, `runC` is written in `Go` and we had to interact with it spawning a shell from the main process. We would like to eventually try `crun`, an OCI-compliant runtime written in `C`, and see whether performance could be improved. We leave this for future work.

4.2.1 High Level Specification

We have implemented a tool that, given a container name and a remote server, migrates execution from the host where the command is run to the one specified in the argument list. In particular, it checkpoints the container, transfers the image files over the network, and restores remotely the execution. As optional parameters, the user might specify whether they prefer diskless or file-based migration (defaults to diskless), iterative or one shot (defaults to iterative), and the path where intermediate files will be stored.

We make two important assumptions in the current implementation. First, we assume that the user running the migration has access to the machine specified in the argument list. In particular we assume SSH access. We have chosen to rely on a single execution process, rather than having a client-server architecture like other solutions do [35]. Hence why we don't need a server process running in the remote machine, but require execution privileges there. Second, we assume the required container bundle (`rootfs` directory and `config.json` specification file) to be available in the remote end. These are required to restore the container. Note that this assumption could be easily circumvented pre-generating the bundle before executing the migration. The procedure to create a `rootfs` bundle is very straightforward and we include an example in Listing 4.8.

```bash
#!/bin/bash
# Create a new directory for the container
mdkir ./my-container && cd my-container

# Create a root file system
```

```
mkdir rootfs

# Export the docker image into the root file system
docker export ‘docker create <image-tag>‘ | tar -C rootfs -xvf -

# Generate the config file using the OCI runtime tool
oci-runtime-tool generate <args>
```

Listing 4.8: Commands to generate an OCI bundle to run a container using `runC`.

Lastly, it is worth mentioning that the support for rootless containers is still work-in-progress in the CRIU project [70]. Therefore, to run the software the user must have `root` permissions in both machines (in particular `CAP_SYS_ADMIN`).

We have also implemented two additional modules. One performs all the interactions with the remote host such as transferring files, creating directories, and running CRIU commands. In particular, we use `libssh` [71] to interact with the other node, and implement our methods basing on their low-level primitives. The second is a benchmarking module that through conditional compilation adds profiling instructions and generates reports to populate the plots we present throughout this document.

4.2.2 Implementation Details

Migration Module

The first step consists in processing the user input. We require the container name, which must be a running container (*i.e.* must have a matching entry in `sudo runc list`) and an IP address (more on that on the networking module). We also allow for a series of optional parameters. Firstly, the user might choose to perform a one-shot migration, rather than an iterative one, and the path to store intermediate results can also be determined at this stage. Second, the user can opt to not use diskless migration and persist intermediate files. In the event of a diskless migration, the parameters (address and port) where the page server runs also can be determined. Lastly, the user can also specify a directory where to mount the `tmpfs` file system. Otherwise the system will default to, if available, `/dev/shm`. For the remaining of the walkthrough we will assume the default parameters are set: diskless, iterative migration, with a page server running in the remote end and relying on `/dev/shm` as our `tmpfs` file system of choice.

The most important part in the migration procedure is a loop that periodically and while the amount of memory to transfer exceeds a threshold does the following:

1. **Create Directories.** The first step is to create the local and remote directories where image files will be stored. This has to be done before anything else as otherwise the page server will report an error and crash.

2. **Start Page Server.** As previously introduced, CRIU's page server is a one-shot command.

This means that once it serves a request it terminates. As a consequence at the start of every iteration the command must be re-run. We start it with the following snippet: `sudo criu page-server -d --images-dir <path> --prev-images-dir <prev-path> --port <port>`. Note that the `prev-images-dir` is crucial to avoid memory duplication.

3. **Checkpoint.** Now we are ready to checkpoint the container. We use the following command: `sudo runc checkpoint --pre-dump --image-path <path> --page-server <host>:<port> <container_name>`. Note that the `--pre-dump` flag is crucial to keep the container running and only dump the contents of the memory, which will be automatically over the network to the page server (and not written to disk).

4. **Transfer Remaining Files.** Even though we are only running a `pre-dump`, there are a few files that need to be transferred to the other node, and we do so at this stage. We additionally perform housekeeping duties cleaning temporary files.

5. **Update Directory Counter.** The last step, and not to be overlooked, consists in updating the directory chain that links iterative dumps. In point 2, we need to specify the path to the preceding directory, as a consequence we keep a linked list of directory names.

We include a simplified version of this loop in Listing 4.9. Most of the technical details have been omitted for simplicity, and should be directly inspected in GitHub.

```c
while (size_to_xfer > MEMORY_THRESHOLD)
{
    /* Prepare Migration: create directories and start page server */
    if (prepare_migration(args, i == 0) != 0)
    {
        fprintf(stderr, "iterative_migration: prepare migration failed at \
                         iteration %i.\n", i + 1);
        return 1;
    }
    memset(cmd_dump, '\0', MAX_CMD_SIZE);
    if (i == 0)
        sprintf(cmd_dump, "sudo runc checkpoint --pre-dump --image-path %s \
                --page-server %s:%s %s", args->src_image_path,
                args->dst_host, args->page_server_port, args->name);
    else
        sprintf(cmd_dump, fmt_cmd_dump, args->src_image_path,
                args->src_prev_image_dir, args->dst_host,
                args->page_server_port, args->name);

    /* Run Pre-Dump */
    if (system(cmd_dump) != 0)
    {
        fprintf(stderr, "iterative_migration: pre-dump #%i failed.\n", i);
        return 1;
    }

    /* Transfer the Remaining Files */
    if (sftp_copy_dir(args->session, args->dst_image_path,
                      args->src_image_path, 0, &dir_size) != SSH_OK)
    {
        fprintf(stderr, "migration: error transferring from '%s' to '%s'\n",
                args->src_image_path, args->dst_image_path);
    }

    /* Swap Dirs */
    if (increment_dirs(i) != 0)
    {
        fprintf(stderr, "migration: error incrementing dirs\n");
        return 1;
    }
    i++;
}
```

Listing 4.9: Iterative migration internal loop.

Once we exit the loop, it means the remaining files to transfer are sufficiently small. We then proceed to checkpoint and stop the container, transfer the remaining files, and restore it in the other node. We present a simplified version of the code in Listing 4.10.

```c
/* Prepare Environment: create directories and start page server */
if (prepare_migration(args, 0) != 0)
{
    fprintf(stderr, "migration: prepare_migration failed.\n");
    return 1;
}

/* Craft Checkpoint and Restore Commands */
char *fmt_cp = "sudo runc checkpoint "
               "--parent-path %s "
               "--image-path %s "
               "--tcp-established "
               "--page-server %s:%s %s";
char *fmt_rs = "cd %s && echo %s | sudo -S runc restore --image-path %s \
                %s-restored &> /dev/null < /dev/null &";
sprintf(cmd_cp, fmt_cp, last_dir, args->src_image_path, args->dst_host,
        args->page_server_port, args->name);
sprintf(cmd_rs, fmt_rs, RUNC_REDIS_PATH, REMOTE_PWRD, args->dst_image_path, args->name);

/* Checkpoint the Running Container */
if (system(cmd_cp) != 0)
{
    fprintf(stderr, "migration: error checkpointing w/ command: '%s'\n",
            cmd_cp);
    return 1;
}

/* Copy the Remaining Files (should be few as we are running diskless) */
if (sftp_copy_dir(args->session, args->dst_image_path,
                  args->src_image_path, 0, &dir_size) != SSH_OK)
{
    fprintf(stderr, "migration: error transferring from '%s' to '%s'\n",
            args->src_image_path, args->dst_image_path);
    return 1;
}

/* Restore the Running Container */
if (ssh_remote_command(args->session, cmd_rs, 0) != SSH_OK)
{
    fprintf(stderr, "migration: error restoring w/ command: '%s'\n",
            cmd_rs);
    if (clean_env(args) != 0)
    {
        fprintf(stderr, "migration: clean_env method failed.\n");
        return 1;
    }
    return 1;
}

/* Clean Environment Before Exitting */
if (clean_env(args) != 0)
{
    fprintf(stderr, "migration: clean_env method failed.\n");
    return 1;
}
```

Listing 4.10: Snippet for the last (stopping) checkpoint and remote restore.

Note that before exiting we call the `clean_env` routine that removes temporary files and cleans the remaining processes.

Networking Module

From the code snippets presented in the previous lines, the reader might observe a series of calls to some seemingly unfamiliar methods like `sftp_copy_dir` or `ssh_remote_command`. One of the first design choices we faced was to whether follow a client-server architecture, in which a listening process would have to be running in advance in the destination machine, or run all commands from the same process. We decided to follow the second approach as it required less dependencies on participating nodes (none other than CRIU and `runC`). This decision implied that we would need programmatic access from the main execution process to the remote node in order to: manipulate the file system, transfer files, and execute privileged commands.

For enhanced control, we decided to implement routines to transfer files and execute remote commands using `libssh`'s API [71, 72]. In particular, we expose the following API calls,

- `ssh_remote_command`: execute a command remotely. We open an `ssh_channel`, use the `ssh_channel_request_exec` method, and capture the output. Alternatively we can also run the command in non-blocking mode. Lastly, and in order to execute `root` commands, we used two different workarounds. One first option is to, for the user we authenticate with (part of our initial assumptions), enable password-less sudo. Another one is to pass the password, in clear, over the encrypted SSH channel with a snippet like: `echo <PWD> | sudo -S <cmd>`.
- `sftp_copy_file`: copies a file to the remote end's specified path. `libssh` only exposes very

low level primitives, so we have to serialize and chunk the file, transfer it over an established `sftp` channel, rebuild it in the other end, and store it at the desired location.

- `sftp_copy_dir`: recursively copy a directory using the previously introduced method.

The full implementation is again available on GitHub, but we include the signatures and a simplified implementation of the methods listed above (without the helper functions) in Listing A.2.

4.2.3 Usage

The usage of the tool is very straightforward. First, the user must ensure that the previously mentioned assumptions are met. This is, the container is running in the host machine, and the user has SSH access to the destination node. The only dependencies for the software to run are CRIU and `runC`. `libssh` might or might not be available depending on the Linux flavour the user is running in. At the time of the writing the system has only been tested in `Debian` machines. When all the dependencies are met, the code can be compiled using `make all` and executed via `./migration <container> <host>`. This ease of use is not to be overlooked, as similar tools are way more complicated to set up and get running. For instance, in order to migrate a process using the old P.Haul interface (`https://github.com/checkpoint-restore/p.haul/`), one must set up a NFS mount between source and destination, specify the file descriptors for both RPC and memory socket, and repeat the procedure in both nodes before even starting the script. Moreover, the newer interface (`https://github.com/checkpoint-restore/go-criu/tree/master/phaul`) does not even allow for standalone execution as it has to be included in a Go project.

Chapter 5

Evaluation

In the previous chapter, §4, we have presented our implementation of live migration of containers using CRIU and `runC`, and have motivated our design choices with micro-benchmarks that fulfil the initial objectives specified in the introduction. In this chapter, we move on to evaluate the system as a whole, and how the different features interlace and work together, and how does our system compare to traditional virtual machine live migration.

In all the experiments here presented, and unless otherwise stated, we use the same experimental setup than in the micro-benchmark chapter. Two (if necessary) Linux Debian machines with kernel version `4.19.0-6` running on a host-only network in VirtualBox version `5.2.34`. CRIU version `3.13` and `runC` version `1.0.0-rc8`, both built from source.

5.1 Application Downtime

In this section we study the impact of the threshold value we set to stop the iterative migration and dump the application on the total downtime. As previously introduced in the *Iterative Migration* section from §4.1, we establish an arbitrary parameter to decide when to stop transferring only memory dumps (*i.e.* pre-dumping the container process) and checkpoint, hence freeze, and migrate the application to the remote host. A reasonable rule of thumb is to establish a memory cap, and whenever the successive dumps are smaller than said cap, trigger the threshold and exit the loop. However, a hard-coded value would be very *ad-hoc* to our experimental setup. Therefore, we have benchmarked the impact of setting a variable memory cap to the total downtime. The values we choose are proportional to the size of the initial allocated memory pool.

In order to test this feature we set up the following experiment. We deploy an in-memory Redis database with $1e6$ key pairs which result in an allocated memory (for all the container) of several hundreds of MB. We use `redis-client` and `redis-server` version 5.0.3. For each run, we set a percentage of the initial memory as our threshold value, and report the total application downtime and a breakdown of the time spent (in percentage) during the last dump of each migration to the remote host. In particular we measure the time spent dumping the memory (`dump`), preparing the migration (`prepare`), transferring files to the other host (`transfer`) and during restore (`restore`). We present our results in Figure 5.1.

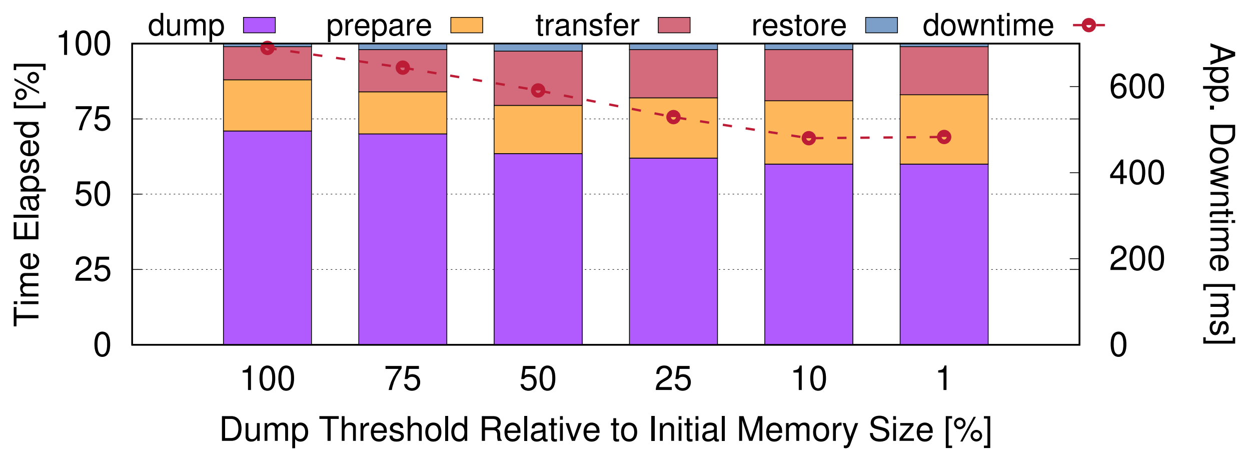


Figure 5.1: Application downtime relative to the memory threshold cap (dotted red), and stacked histogram of the time spent in each phase during the last dump.

We observe that, for our particular setting, once the memory cap reaches 10% of the original size, the application downtime reaches the baseline value. Moreover it is also interesting to see that, as this value gets closer to the baseline, the time spent (absolute and relative) dumping memory decreases. This results are specific to our setting with two virtual machines and limited memory, but a similar benchmark could be reproduced in production to estimate an adequate threshold value.

5.2 Scalability regarding the Container's Memory Size

In this section we study the scalability of our approach with respect to the memory allocated by the to-be checkpointed container. For this experiment we set up a Redis in-memory database which we pre-load with a variable number of key-value pairs. We use, similarly to the previous section, `redis-client` and `redis-server` version `5.0.3`, and pre-load the keys using the `--pipe` flag for bulk data upload. In Listing 5.1 we include the script for doing so. Note that this same strategy was also used in other benchmarks using Redis.

```
#!/bin/bash

# Start a runc container named redis-db
# The ip the service runs on is stored in the .ip file
sudo runc run -d redis-db &> /dev/null < /dev/null

# Populate DB with data
cat "./data/file.dat" | redis-cli -h ‘< .ip‘ --pipe
```

Listing 5.1: Snippet for bulk data upload to a Redis database.

Our goal is to evaluate the impact of the size of the memory allocated to the overall container downtime. In order to compare against a naive live migration (checkpoint, transfer files, and restore on the remote end) and traditional virtual machine migrations, we choose not to use iterative migration in our system. This is due to the fact that it would be an unfair advantage against the other two systems. Additionally, and in order to avoid noise introduced by the network latency, we run migrations locally (*i.e.* transferring files to `127.0.0.1`). We measure the application's downtime (time to checkpoint plus time to restore) when running an in-memory Redis database pre-loaded with key-value pairs of sizes raging 16 B for one pair to 265 MB for $1e7$ pairs.

Comparison Against Naive Live Migration.

We first compare against a *naive* live migration. We define by *naive* the process of dumping (hence stopping) a process, transferring the dump files, and restarting it in the remote end. We report the downtime for this approach and our system when ran with only one iteration in Figure 5.2.

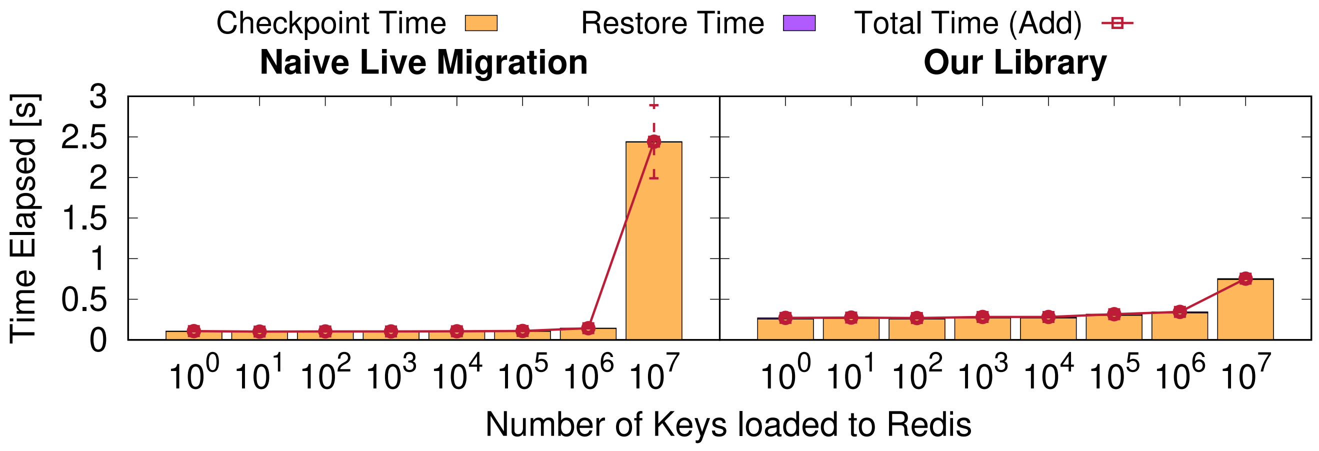


Figure 5.2: Scalability with respect to the memory to transfer. We compare our system with manual live migration when running in the same machine with a one-shot migration.

From our results we are able to extract different conclusions. First of all, the time to restore is negligible when compared to the time to checkpoint/dump the container. Secondly, even though our library introduces a small overhead, around 0.1 seconds, to the baseline, this proofs effective in the long run when the application downtime is reduced by a factor of five. Lastly, we also observe that a user will specially benefit from using a specialized live migration library like ours over the manual approach whenever the container to checkpoint is resource-eager.

Comparison Against Virtual Machine Migration

In this second comparison, we compare against traditional virtual machine migration. Given that our test nodes were already running in VirtualBox version `5.2.3`, we have opted to use

VirtualBox's native live migration solution: teleporting [73]. From the user manual we read that teleporting boils down to moving one virtual machine from one VirtualBox host to another one over TCP/IP.

```bash
#!/bin/bash

# Configure target machine to wait for a teleport request to arrive
VBoxManage modifyvm 'CRIU-Debian-Teleport-Target' --teleporter on --teleporterport 6000

# Iterate over the different number of keys
for num_keys in 1 10 100 1000 10000 100000 1000000 10000000
do
    # Start the host machine as usual
    ssh <HOST_VM>
    cd ~/runc-containers/redis && ./run_redis.sh 100000

    # Start the target machine, if using a normal start, a process dialog will appear

    # Run the migration
    time VBoxManage controlvm 'CRIU-Debian' teleport --host localhost --port 6000

    # Shut down both VMs
done
```

Listing 5.2: Script to teleport a VirtualBox VM, and run the macro-benchmark.

In our particular experiment, the VirtualBox host would be the same, as we are migrating to `localhost`, but to a different virtual machine (a pre-made clone of the origin one). In Listing 5.2 we include the evaluation snippet we used to launch the experiments and teleport the machines. Note that some additional commands had to be made through the GUI, and that the `./run_redis.sh` script is very similar to the one presented in Listing 5.1.

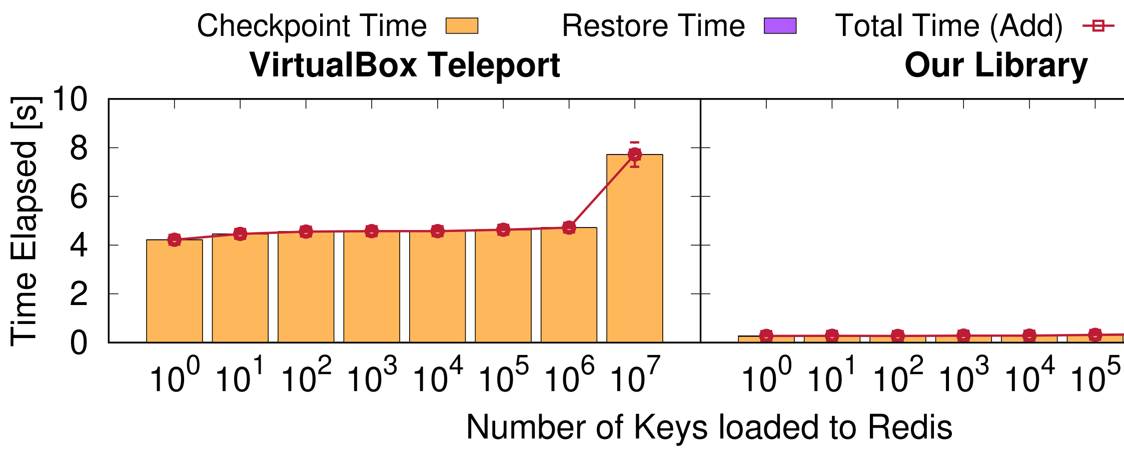


Figure 5.3: Scalability with respect to the memory to transfer. We compare our system with VM live migration using VirtualBox's Teleport functionality.

We present our results in Figure 5.3. The first observation we can make, is that VM live migration has a significantly higher overhead in the baseline case. This was to be expected as overall overhead and slow boot times are the main argument in favour of containers and against virtual machines. And in this experiment we are running one inside of the other. However, what is also worth noticing is the scalability. Whilst our system (for which we re-plot the same results from Figure 5.2), experiences an increase in downtime of $\times 1.5$, VirtualBox's experiences a downtime closer to $\times 2$ (whilst naive migration had one closer to $\times 10$). This result shows that, even with a drastically smaller overhead, our system also showcases a better scalability when compared to traditional VM migration.

The improvements from our library with respect to naive and VM migration can be understood from the following facts. When compared to naive migration, our system does not rely on files written to disk, neither duplicates memory allocation. This adds a small overhead to the baseline, but drastically improves scalability. When compared to VM migration, the difference in baselines comes from the way larger set of files VirtualBox has to copy in order to restore the machine. Furthermore, the slight gains in scalability probably come from the completely diskless migration we employ.

Chapter 6

Conclusions and Future Work

In this Chapter we summarize the work presented, and critically assess whether our contributions match the objectives we initially planned. We also provide an objective overview of the results presented in order to establish if the techniques we use are ready for a production environment. Bear in mind that checkpoint-restore as a load-balancing tool is still a very novel technique, and even more in the context of containerized applications. Lastly, in §6.2 we cover the stones we left unturned, the things we would have liked to include in this thesis but have not been able to, and the research lines we believe this initial approach heads us to.

6.1 Conclusions and Lessons Learnt

Our initial goal was to implement a tool for efficient live migration of containers. It was surprising to us that, such an apparently useful technique (checkpointing of containers), had been abandoned for several years in DOCKER's `experimental` branch. Even though the reasons for this are still unclear to us, what has become apparent is that migrating a container is not an easy task. CRIU is an incredibly complex tool, with a very helpful community, but whose intricate relation with the kernel makes it hard to debug whenever things don't go as expected. Luckily, the integration with other container engines (other than DOCKER), is way more maintained, resourceful, documented, and tested. These other container engines target a different audience than DOCKER does, and hence why the feature may not be mainline in the latter.

Live migration turned out not to be only checkpointing, transferring files, and restoring from another node. At least not if we were looking for performance even in the event of resource-intensive containers with established TCP connections and external namespaces. The process of optimizing (less resource usage), reducing downtime, and integrating further capabilities, was done by micro-benchmarking each different feature to motivate our design choices as presented in §4.1. It has lead us to use, by default, iterative, diskless migration with a special handling of namespaces and IP tables filtering in the event of existing connections. And to our ultimate goal of a single binary file that, given a container name and a target IP, migrates efficiently said container. The current implementation can be downloaded and tested from `https://github.com/live-containers/live-migration`.

Our experimental results presented in Chapter 5, validate our design. Firstly, resource utilization, and in particular disk usage and memory duplication, is drastically reduced when compared to a naive migration approach. Secondly, scalability with the size of the allocated memory is better than both naive migration and VM migration, whilst maintaining a baseline of approximately $0.2 - 0.3$ seconds (bear in mind that this time takes dump, restore, and network transfer and latency into account). Thirdly, the throughput in downtime perceived by a network-intensive client when migrating the server to the same location (*i.e.* simulating a maintenance reboot) was under 0.1 seconds when flooding the link. We therefore conclude that this would be close to negligible for a regular client. Lastly, we provide a procedure and benchmarking technique to estimate a threshold value necessary for iterative migration.

As a consequence, we believe that the techniques we have used and the work here presented are mature enough to be used, at least, in replacement of traditional virtual machine migration. We are also confident that migration of containers, their native support within engines and orchestrators, and their integration with larger frameworks, will see a drastic increase in the coming years.

6.2 Future Work

Unfortunately, and as it tends to be the case, there has been much work we would have liked to include in the present work but we have not been able to. Either due to a lack of time or a lack of expertise and experience, there are some areas of this research that we would like to polish, and some ones which we would like to push forward in the future.

From a technical standpoint, there are some implementation and evaluation details we would like to complete. Firstly, as mentioned in §4.2, there are some assumptions we make on both source and destination hosts. With regard to the authentication of the user in the remote host, we have found no shortcoming. Even if we were to use a client-server architecture, the listening counterpart would also need to run in privileged mode. Support for rootless containers and rootless restore in CRIU is not available [74, 75], so our pre-requisites in this regard are necessary and sufficient. As for the pre-provisioning of the OCI bundle to start a `runC` container and the image transfer optimization, the former could be easily scripted, and the latter is an active open research topic [76] in CRIU, which we could leverage in the near future. Secondly, we would like to perform further benchmarking against other live migration tools. In particular, we were unable to compare against P.Haul due to the project not being actively maintained (as an executable) and only distributed as a library. With some additional time we could indeed compare both solutions, together with other VM migration tools (other than VirtualBox's) like those offered by LXC and KVM. We believe that with these additional contributions, a trimmed version of the material here presented is suitable to be submitted to a dedicated conference, and we plan to do so throughout the coming months.

On a broader scope, the over-arching goal of this project was to support live migration for distributed container deployments. We believe the work here presented is a necessary first step towards achieving it, but there's still much work to be done. From an algorithmic standpoint, distributed checkpointing and coordination algorithms need to be implemented. From an infrastructure standpoint, distributed container deployments are managed through an orchestrator. The integration of CRIU with such a tool is, to the best of our knowledge, unexplored territory and something we look forward to doing in the future.

Bibliography

[1] M. Masdari, S. Nabavi, and V. Ahmadi. An overview of virtual machine placement schemes in cloud computing. *Journal of Network and Computer Applications*, 66, 01 2016.

[2] A. Strunk. Costs of virtual machine live migration: A survey. In *2012 IEEE Eighth World Congress on Services*, pages 323–329, June 2012.

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

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

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

[6] 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.

[7] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[8] S. Hykes. Introducing runC: a lightweight universal container runtime. `https://www.docker.com/blog/runc/`, 2017.

[9] CRIU Foundation. Process Haul - Github. `https://github.com/checkpoint-restore/go-criu`, 2019.

[10] Linux Programmer's Manual. unshare: run program with some unshared namespace from parent. `https://linux.die.net/man/1/unshare`, 2017.

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

[12] Linux Programmer's Manual. clone: create child process. `https://linux.die.net/man/2/clone`, 2018.

[13] Linux Programmer's Manual. unshare: run program with some unshared namespace from parent. `https://linux.die.net/man/1/unshare`, 2017.

[14] Linux Programmer's Manual. setns: reassociate thread with namespace. `http://man7.org/linux/man-pages/man2/setns.2.html`, 2020.

[15] S. McCarty. A practical introduction to container terminology. `https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/`, 2018.

[16] Open Container Initiative. Open Container Initiative Runtime Specification. `https://github.com/opencontainers/runtime-spec/blob/master/spec.md`, 2019.

[17] M. Schulz. *Checkpointing*, pages 264–273. Springer US, Boston, MA, 2011.

[18] M. Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer Publishing Company, Incorporated, 2013.

[19] 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*, NSDI'05, page 273–286, USA, 2005. USENIX Association.

[20] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[21] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, page 169–179, New York, NY, USA, 2007. Association for Computing Machinery.

[22] A. Shribman and B. Hudzia. Pre-copy and post-copy vm live migration for memory intensive applications. In *Euro-Par 2012: Parallel Processing Workshops*, pages 539–547, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[23] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[24] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

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

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

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

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

[29] P. Emelyanov. CRIU - Freezing the tree. https://criu.org/Freezing_the_tree, 2017.

[30] CRIU Foundation. Criu - parasite code. https://criu.org/Parasite_code, 2018.

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

[32] A. Reber. Criu and the pid dance. https://linuxplumbersconf.org/event/4/contributions/472/attachments/224/397/2019-criu-and-the-pid-dance.pdf, 2019.

[33] C. Brauner. fork: add clone3. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7f192e3cd316ba58c, 2019.

[34] Criu Foundation. Criu - live migration. https://criu.org/Live_migration, 2019.

[35] CRIU Foundation. Criu - p. haul. https://criu.org/P_haul, 2018.

[36] CRIU Foundation. Criu - lazy migration. https://criu.org/Lazy_migration, 2018.

[37] CRIU Foudnation. Comparison with other CR Projects - CRIU. https://criu.org/Comparison_to_other_CR_projects, 2019.

[38] DMTCP. Distributed MultiThreaded Checkpointing. http://dmtcp.sourceforge.net/, 2019.

[39] Computer Language and Systems Software Group. Berkeley Lab Checkpoint/Restart (BLCR) for LINUX. https://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/, 2013.

[40] J. Ansel, K. Arya, and G. Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, page 1–12, USA, 2009. IEEE Computer Society.

[41] R. Garg, G. Price, and G. Cooperman. Mana for mpi: Mpi-agnostic network-agnostic transparent checkpointing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, page 49–60, New York, NY, USA, 2019. Association for Computing Machinery.

[42] CRIU Foudnation. Comparison with other CR Projects - CRIU. `https://criu.org/Comparison_to_other_CR_projects`, 2019.

[43] Open Containers. Runc - Automatic Memory Tracking. `https://github.com/opencontainers/runc/blob/d4a6a1d99875aa571ebe95babf0ac14f54079282/libcontainer/container_linux.go#L1034`, 2019.

[44] Open Containers Initiative. OCI - Runtime Specification. `https://github.com/opencontainers/runtime-spec`, 2020.

[45] CRIU Foudnation. CRIU - All articles. `https://criu.org/Articles`, 2017.

[46] 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.

[47] CRIU Foundation. Memory changes tracking. `https://criu.org/Memory_changes_tracking`, 2019.

[48] P. Emelyanov. mm: Ability to monitor task memory changes (v3). `https://lwn.net/Articles/546966/`, 2013.

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

[50] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, page 1–18, Washington, DC, USA, 2002. IEEE Computer Society Press.

[51] K. Arya, R. Garg, A. Polyakov, and G. Cooperman. Design and implementation for checkpointing of distributed resources using process-level virtualization. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 402–412, 2016.

[52] S. Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 53–65, New York, NY, USA, 2019. Association for Computing Machinery.

[53] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran. Edge computing: The case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 73–87, New York, NY, USA, 2020. Association for Computing Machinery.

[54] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis. Live service migration in mobile edge clouds. *Wireless Commun.*, 25(1):140–147, February 2018.

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

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

[57] Gustaf L. Berg and Magnus Brattlöf. Distributed checkpointing with docker containers in high performance computing. 2017.

[58] 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.

[59] CRIU Foundation. Process Haul - CRIU Wiki. `https://criu.org/P.Haul`, 2019.

[60] Linux Programmer's Manual. tmpfs - A virtual memory filesystem. `http://man7.org/linux/man-pages/man5/tmpfs.5.html`, 2019.

[61] P. Snyder. tmpfs: A virtual memory file system. 2007.

[62] R. Stoyanov. Criu - page server. `https://criu.org/Page_server`, 2019.

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

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

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

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

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

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

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

[70] R. Stoyanov. CRIU - User Mode. `https://criu.org/User-mode`, 2019.

[71] libssh Org. libSSH - The SSH Library. https://www.libssh.org/, 2019.

[72] libssh Org. The libSSH API. https://api.libssh.org/master/group__libssh.html, 2019.

[73] VirtualBox User Manual. 7.2. Teleporting. https://www.virtualbox.org/manual/ch07.html#teleporting, 2019.

[74] GitHub Issue. runc: checkpointing a rootless container. https://github.com/opencontainers/runc/issues/2009, 2019.

[75] G. Scrivano and A. Suda. Rootless containers. https://rootlesscontaine.rs/, 2019.

[76] GitHub. CRIU Image Streamer. https://github.com/checkpoint-restore/criu-image-streamer, 2020.

Appendix A

Implementation Code Snippets

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h> /* atoi */
#include <unistd.h>

static volatile int keep_running = 1;

/* Handler to graciously stop with Ctrl+C */
void int_handler(int tmp)
{
    keep_running = 0;
}

/* Compile with: gcc counter.c -o counter */
int main(int argc, char *argv[])
{
    int count = 0;
    int inc = 0;
    if (argc > 1)
        inc = atoi(argv[1]);
    signal(SIGINT, int_handler);

    fprintf(stdout, "Current count: %i\n", count++);
    if (inc)
    {
        while (keep_running)
        {
            fprintf(stdout, "Current count: %i\n", count++);
            sleep(2);
        }
    }
    else
    {
        while (keep_running)
            sleep(20);
    }

    return 0;
}
```

Listing A.1: Simple counter in C.

```c
int sftp_copy_file(ssh_session session, char *dst_path, char *src_path)
{
    sftp_session sftp;
    int rc;

    sftp = sftp_new(session);
    /* Allocate SFTP Session */
    if (sftp == NULL)
    {
        fprintf(stderr, "sftp_copy_file: Error allocating SFTP session: %s\n",
                ssh_get_error(session));
        return SSH_ERROR;
    }

    /* Initialize SFTP Client */
    rc = sftp_init(sftp);
    if (rc != SSH_OK)
    {
        fprintf(stderr, "sftp_copy_file: Error initializing SFTP session: %d\n",
                sftp_get_error(sftp));
        sftp_free(sftp);
        return rc;
    }

    if (sftp_xfer_file(sftp, dst_path, src_path, NULL) != SSH_OK)
        return SSH_ERROR;

    sftp_free(sftp);
    return SSH_OK;
}

/* Copy the Contents of the Source Directory to the Destination One
 *
 * ssh_session session: current authenticated ssh_session.
 * char *dst_path: path to the (existing) destination directory.
 * char *ori_path: path to the (existing) origin directory from where to copy.
 * int rm_ori: if set to 1, it will remove the contents of the origin directory.
 * int *dir_size: if not null, will return the size of the dir xfered.
 */
int sftp_copy_dir(ssh_session session, char *dst_path, char *src_path,
                  int rm_ori, double *dir_size)
{
    sftp_session sftp;
    int rc;

    sftp = sftp_new(session);
    /* Allocate SFTP Session */
    if (sftp == NULL)
    {
        fprintf(stderr, "sftp_copy_dir: Error allocating SFTP session: %s\n",
                ssh_get_error(session));
        return SSH_ERROR;
    }

    /* Initialize SFTP Client */
    rc = sftp_init(sftp);
    if (rc != SSH_OK)
    {
        fprintf(stderr, "sftp_copy_dir: Error initializing SFTP session: %d\n",
                sftp_get_error(sftp));
        sftp_free(sftp);
        return rc;
    }

    /* Iterate over source directory. */
    DIR *d;
    struct dirent *src_dir;
    //struct stat src_stat;
    d = opendir(src_path);
    if (d)
    {
        /* Create remote copy of directory. */
        /* TODO make this optional?
        if (sftp_mkdir(sftp, dst_path, 0755) != 0)
        {
            fprintf(stderr, "sftp_copy_dir: Error creating remore directory %d\n",
                    sftp_get_error(sftp));
            sftp_free(sftp);
            return SSH_ERROR;
        }
        */
        char resolved_path[PATH_MAX + 1];
        char src_rel_path[PATH_MAX + 1], dst_rel_path[PATH_MAX + 1];
        memset(src_rel_path, '\0', PATH_MAX + 1);
        memset(dst_rel_path, '\0', PATH_MAX + 1);
        memset(resolved_path, '\0', PATH_MAX + 1);
        while ((src_dir = readdir(d)) != NULL)
        {
            if (src_dir->d_type == DT_REG)
            {
                /* Generate full paths */
                strncpy(src_rel_path, src_path, strlen(src_path));
                strcat(src_rel_path, "/");
                strcat(src_rel_path, src_dir->d_name);
                strncpy(dst_rel_path, dst_path, strlen(dst_path));
                strcat(dst_rel_path, "/");
                strcat(dst_rel_path, src_dir->d_name);
                if (realpath(src_rel_path, resolved_path) == NULL)
                {
                    fprintf(stderr, "sftp_copy_dir: Error obtaining file's real path: %s\n",
                            src_dir->d_name);
                    sftp_free(sftp);
                    return SSH_ERROR;
                }
                if (sftp_xfer_file(sftp, dst_rel_path, resolved_path, dir_size)
                        != SSH_OK)
                {
                    fprintf(stderr, "sftp_copy_dir: error copying %s \
                                     to %s\n. %i\n", resolved_path,
                                     dst_rel_path, sftp_get_error(sftp));
                    sftp_free(sftp);
                    return SSH_ERROR;
                }
                if (rm_ori && remove(resolved_path) != 0)
                {
                    fprintf(stderr, "sftp_copy_dir: error removing local \
                                     file %s (remove flag set)\n",
                                     resolved_path);
                    sftp_free(sftp);
                    return SSH_ERROR;
                }
                memset(src_rel_path, '\0', PATH_MAX + 1);
                memset(dst_rel_path, '\0', PATH_MAX + 1);
                memset(resolved_path, '\0', PATH_MAX + 1);
            }
            else if (src_dir->d_type == DT_LNK)
            {
                /* On iterative migration, each intermediate checkpoint dir
                 * has a symbolic link to its "parent". Copying it
                 * programatically is more verbose than crafting it ourselves.
                 */
                strncpy(src_rel_path, src_path, strlen(src_path));
                strcat(src_rel_path, "/");
                strcat(src_rel_path, src_dir->d_name);
                if (remove(src_rel_path) != 0)
                {
                    fprintf(stderr, "sftp_copy_dir: error removing \
                                     symlink.\n");
                    return 1;
                }
                memset(src_rel_path, '\0', PATH_MAX + 1);
            }
        }
        closedir(d);
    }
    else
    {
        fprintf(stderr, "sftp_copy_dir: Error listing source directory!\n");
        sftp_free(sftp);
        return SSH_ERROR;
    }

    if (rm_ori && (rmdir(src_path) != 0))
    {
        fprintf(stderr, "sftp_copy_dir: failed removing origin directory \
                         '%s'\n", src_path);
        sftp_free(sftp);
        return SSH_ERROR;
    }
    sftp_free(sftp);
    return SSH_OK;
}
\end{listing}
int ssh_remote_command(ssh_session session, char *command, int read_output)
{
    ssh_channel channel;
    int rc;
    char buffer[256];
    int nbytes;

    /* Open a new SSH Channel */
    channel = ssh_channel_new(session);
    if (channel == NULL)
    {
        fprintf(stderr, "ssh_remote_command: Error allocating new SSH channel.\n");
        return SSH_ERROR;
    }
    rc = ssh_channel_open_session(channel);
    if (rc != SSH_OK)
    {
        fprintf(stderr, "ssh_remote_command: Error opening new SSH channel.\n");
        ssh_channel_free(channel);
        return rc;
    }

    /* Execute Remote Command
     *
     * We need to run the commands as sudo in the remote system as well
     * (criu needs to run as root) so I thought of two different ways
     * of tackling the problem:
     * 1. Passing the password as plain text.
     * 2. Manually setup each host to allow rootless sudo.
     * */
    /*
    char sudo_command[MAX_CMD_SIZE];
    memset(sudo_command, '\0', MAX_CMD_SIZE);
    sprintf(sudo_command, "echo %s | sudo -S %s", REMOTE_PWRD, command);
    */
    rc = ssh_channel_request_exec(channel, command);
    if (rc != SSH_OK)
    {
        fprintf(stderr, "ssh_remote_command: Error executing remote command: %s\n",
                command);
        ssh_channel_close(channel);
        ssh_channel_free(channel);
        return rc;
    }

    /* Check the Exit Status of the Remote Command */
    rc = ssh_channel_get_exit_status(channel);
    switch (rc)
    {
        case 0:
            printf("DEBUG: command '%s' exitted succesfully!\n", command);
            break;

        case -1:
            printf("DEBUG: still no exit code received!\n");
            break;

        default:
            fprintf(stderr, "ssh_remote_command: remote command '%s' failed w/ exit status %i\n",
                    command, rc);
            return SSH_ERROR;
    }

    if (read_output)
    {
        /* Read Output in chunks */
        nbytes = ssh_channel_read(channel, buffer, sizeof buffer, 0);
        while(nbytes > 0)
        {
            fprintf(stdout, "%s", buffer);
            /* FIXME check for errors
            if (fprintf(stdout, "%s", buffer) != (unsigned int) nbytes)
            {
                fprintf(stderr, "Error printing results.\n");
                ssh_channel_close(channel);
                ssh_channel_free(channel);
                return SSH_ERROR;
            }
            */
            nbytes = ssh_channel_read(channel, buffer, sizeof buffer, 0);
        }

        if (nbytes < 0)
        {
            ssh_channel_close(channel);
            ssh_channel_free(channel);
            return SSH_ERROR;
        }
    }

    ssh_channel_send_eof(channel);
    ssh_channel_close(channel);
    ssh_channel_free(channel);
    return SSH_OK;
}

int sftp_copy_file(ssh_session session, char *dst_path, char *src_path)
{
    sftp_session sftp;
    int rc;

    sftp = sftp_new(session);
    /* Allocate SFTP Session */
    if (sftp == NULL)
    {
        fprintf(stderr, "sftp_copy_file: Error allocating SFTP session: %s\n",
                ssh_get_error(session));
        return SSH_ERROR;
    }

    /* Initialize SFTP Client */
    rc = sftp_init(sftp);
    if (rc != SSH_OK)
    {
        fprintf(stderr, "sftp_copy_file: Error initializing SFTP session: %d\n",
                sftp_get_error(sftp));
        sftp_free(sftp);
        return rc;
    }

    if (sftp_xfer_file(sftp, dst_path, src_path, NULL) != SSH_OK)
        return SSH_ERROR;

    sftp_free(sftp);
    return SSH_OK;
}

/* Copy the Contents of the Source Directory to the Destination One
 *
 * ssh_session session: current authenticated ssh_session.
 * char *dst_path: path to the (existing) destination directory.
 * char *ori_path: path to the (existing) origin directory from where to copy.
 * int rm_ori: if set to 1, it will remove the contents of the origin directory.
 * int *dir_size: if not null, will return the size of the dir xfered.
 */
int sftp_copy_dir(ssh_session session, char *dst_path, char *src_path,
                  int rm_ori, double *dir_size)
{
    sftp_session sftp;
    int rc;

    sftp = sftp_new(session);
    /* Allocate SFTP Session */
    if (sftp == NULL)
    {
        fprintf(stderr, "sftp_copy_dir: Error allocating SFTP session: %s\n",
                ssh_get_error(session));
        return SSH_ERROR;
    }

    /* Initialize SFTP Client */
    rc = sftp_init(sftp);
    if (rc != SSH_OK)
    {
        fprintf(stderr, "sftp_copy_dir: Error initializing SFTP session: %d\n",
                sftp_get_error(sftp));
        sftp_free(sftp);
        return rc;
    }

    /* Iterate over source directory. */
    DIR *d;
    struct dirent *src_dir;
    //struct stat src_stat;
    d = opendir(src_path);
    if (d)
    {
        /* Create remote copy of directory. */
        /* TODO make this optional? */
        if (sftp_mkdir(sftp, dst_path, 0755) != 0)
        {
            fprintf(stderr, "sftp_copy_dir: Error creating remore directory %d\n",
                    sftp_get_error(sftp));
            sftp_free(sftp);
            return SSH_ERROR;
        }
        */
        char resolved_path[PATH_MAX + 1];
        char src_rel_path[PATH_MAX + 1], dst_rel_path[PATH_MAX + 1];
        memset(src_rel_path, '\0', PATH_MAX + 1);
        memset(dst_rel_path, '\0', PATH_MAX + 1);
        memset(resolved_path, '\0', PATH_MAX + 1);
        while ((src_dir = readdir(d)) != NULL)
        {
            if (src_dir->d_type == DT_REG)
            {
                /* Generate full paths */
                strncpy(src_rel_path, src_path, strlen(src_path));
                strcat(src_rel_path, "/");
                strcat(src_rel_path, src_dir->d_name);
                strncpy(dst_rel_path, dst_path, strlen(dst_path));
                strcat(dst_rel_path, "/");
                strcat(dst_rel_path, src_dir->d_name);
                if (realpath(src_rel_path, resolved_path) == NULL)
                {
                    fprintf(stderr, "sftp_copy_dir: Error obtaining file's real path: %s\n",
                            src_dir->d_name);
                    sftp_free(sftp);
                    return SSH_ERROR;
                }
                if (sftp_xfer_file(sftp, dst_rel_path, resolved_path, dir_size)
                        != SSH_OK)
                {
                    fprintf(stderr, "sftp_copy_dir: error copying %s \
                                 to %s\n. %i\n", resolved_path,
                                 dst_rel_path, sftp_get_error(sftp));
                    sftp_free(sftp);
                    return SSH_ERROR;
                }
                if (rm_ori && remove(resolved_path) != 0)
                {
                    fprintf(stderr, "sftp_copy_dir: error removing local \
                                 file %s (remove flag set)\n",
                                 resolved_path);
                    sftp_free(sftp);
                    return SSH_ERROR;
                }
                memset(src_rel_path, '\0', PATH_MAX + 1);
                memset(dst_rel_path, '\0', PATH_MAX + 1);
                memset(resolved_path, '\0', PATH_MAX + 1);
            }
            else if (src_dir->d_type == DT_LNK)
            {
                /* On iterative migration, each intermediate checkpoint dir
                 * has a symbolic link to its "parent". Copying it
                 * programatically is more verbose than crafting it ourselves.
                 */
                strncpy(src_rel_path, src_path, strlen(src_path));
                strcat(src_rel_path, "/");
                strcat(src_rel_path, src_dir->d_name);
                if (remove(src_rel_path) != 0)
                {
                    fprintf(stderr, "sftp_copy_dir: error removing \
                                     symlink.\n");
                    return 1;
                }
                memset(src_rel_path, '\0', PATH_MAX + 1);
            }
        }
        closedir(d);
    }
    else
    {
        fprintf(stderr, "sftp_copy_dir: Error listing source directory!\n");
        sftp_free(sftp);
        return SSH_ERROR;
    }

    if (rm_ori && (rmdir(src_path) != 0))
    {
        fprintf(stderr, "sftp_copy_dir: failed removing origin directory \
                         '%s'\n", src_path);
        sftp_free(sftp);
        return SSH_ERROR;
    }
    sftp_free(sftp);
    return SSH_OK;
}
```

Listing A.2: Signature and schematic implementation of remote execution methods.

Appendix B

Evaluation Code Snippets

```bash
#!/bin/bash
HOME=$(pwd)
#IP=127.0.0.1
IP=192.168.56.103
NUM_TESTS=100
acc=0
acc2=0
for (( i=1; i<=$NUM_TESTS; i++ ))
do
    # Choose application to run
    #cd /home/carlos/runc-containers/counter/ && sudo ./run.sh && cd $HOME
    cd /home/carlos/runc-containers/redis/ && sudo ./run_redis.sh 10000000 && cd $HOME
    # Clean working Environment
    sudo ./clean.sh
    ssh carlos@${IP} "/home/carlos/runc-diskless/clean.sh"
    # Start (or not) the page server
    #sudo ./page_server.sh &
    #ssh carlos@${IP} "/home/carlos/runc-diskless/page_server.sh &> /dev/null < /dev/null &"
    # Start timing
    ts=$(date +%s%N)
    # Dump the process
    sudo ./dump.sh
    # Copy the remaining images
    #scp -r ./src-images/* ./dst-images/
    scp -r ./src-images/* carlos@${IP}:runc-diskless/dst-images/
    time_elapsed=$((($(date +%s%N) - $ts)/1000000))
    acc=$(($acc + $time_elapsed))
    acc2=$(($acc2 + $time_elapsed * $time_elapsed))
    echo "Test $i: $time_elapsed"
done
# Compute average and standard deviation
avg=$(bc <<<"scale=2; $acc / $NUM_TESTS")
std=$(bc <<<"scale=2; sqrt($acc2 / $NUM_TESTS - $avg * $avg)")
echo "Average: $avg"
echo "Std: $std"
```

Listing B.1: Full evaluation script for the diskless migration micro-benchmark.

```bash
#!/bin/bash
# Run Process
# Redis Test (Pre-loading omitted)
redis_server --port 9999 &> /dev/null < /dev/null &
# Counter Test
sudo ./counter &> /dev/null < /dev/null &
# If running with runC
cd container-dir && sudo runc run -d container_name &> /dev/null < /dev/null & && cd -
PID=$!

# Clean Environment
sudo ./clean.sh
echo "Clean Done"

# First pre-dump
sudo ./pre-dump.sh ${PID}
echo "Pre-Dump Done"

# If running the Redis test, run the benchmark
redis-benchmark -p 9999 -n 10000 &> /dev/null

# Second pre-dump
sudo ./middle-dump.sh ${PID}
echo "Middle-Dump Done"

# If running the Redis test, run the benchmark
redis-benchmark -p 9999 -n 10000 &> /dev/null

# Last Dump
sudo ./dump.sh ${PID}
echo "Dump Done"

# Print results
D1=$(ls -lah ./images/1/pages-1.img | awk '{ print $5; }')
D2=$(ls -lah ./images/2/pages-1.img | awk '{ print $5; }')
D3=$(ls -lah ./images/3/pages-1.img | awk '{ print $5; }')
echo "Test finished: -D1: $D1 -D2: $D2 -D3: $D3"
```

Listing B.2: Full evaluation script for the iterative migration micro-benchmark.

```bash
#!/bin/bash
# Declare Variables
CLIENT_IP=192.168.56.103
IPERF3=/home/carlos/iperf/src/iperf3
LOG_DIR=./iperf3-log
IMAGES_DIR=./images

# Set up Environment
mkdir -p ${LOG_DIR}

# Run iPerf3 server
echo "Bootstrapping Server..."
setsid ${IPERF3} \
    -s --port 9999 \
    --json \
    --interval 0.1 \
    --logfile ${LOG_DIR}/server.json \
    --one-off &> /dev/null < /dev/null &
```

```
SERVER_PID=$!

sleep 3

# Run iPerf3 client in remote machine
echo "Bootstrapping Client..."
ssh carlos@${CLIENT_IP} "/home/carlos/tcp-established/iperf3_client.sh"

sleep 10

# CRIU Dump
echo "Dumping server..."
sudo criu dump \
    -t ${SERVER_PID} \
    --images-dir ${IMAGES_DIR} \
    --tcp-established &

sleep 2

# CRIU Restore
echo "Restoring server..."
sudo criu restore \
    --images-dir ${IMAGES_DIR} \
    --tcp-established
```

Listing B.3: Evaluation script for the TCP connection downtime micro-benchmark using CRIU.

```
#!/bin/bash
# Declare Variables
CLIENT_IP=192.168.56.103
IPERF3=/home/carlos/iperf/src/iperf3
LOG_DIR=./iperf3-log
IMAGES_DIR=./images

# Set up Environment
mkdir -p ${LOG_DIR}

# Run iPerf3 server
echo "Bootstrapping Server..."
setsid ${IPERF3} \
    -s --port 9999 \
    --json \
    --interval 0.1 \
    --logfile ${LOG_DIR}/server.json \
    --one-off &> /dev/null < /dev/null &
SERVER_PID=$!

sleep 3

# Run iPerf3 client in remote machine
echo "Bootstrapping Client..."
ssh carlos@${CLIENT_IP} "/home/carlos/tcp-established/iperf3_client.sh"

sleep 4

# CRIU Dump and Restore, one after the other but in the BG (not affecting time)
```

```bash
echo "Dumping server for the first time..."
(sudo criu dump \
    -t ${SERVER_PID} \
    --images-dir ${IMAGES_DIR} \
    --tcp-established; \
echo "Restoring server..."; \
sudo criu restore \
    --images-dir ${IMAGES_DIR} \
    --tcp-established) &

sleep 6

# CRIU Dump and Restore, one after the other but in the BG (not affecting time)
echo "Dumping server for the second time..."
(sudo criu dump \
    -t ${SERVER_PID} \
    --images-dir ${IMAGES_DIR} \
    --tcp-established; \
echo "Restoring server..."; \
sudo criu restore \
    --images-dir ${IMAGES_DIR} \
    --tcp-established) &

sleep 4

# CRIU Dump and Restore, one after the other but in the BG (not affecting time)
echo "Dumping server for the last time..."
(sudo criu dump \
    -t ${SERVER_PID} \
    --images-dir ${IMAGES_DIR} \
    --tcp-established; )
echo "Restoring server..."; \
sudo criu restore \
    --images-dir ${IMAGES_DIR} \
    --tcp-established)
```

Listing B.4: Evaluation script for the TCP connection reactivity micro-benchmark using CRIU.

```bash
#!/bin/bash
# Declare Variables
CLIENT_IP=192.168.56.103
IPERF3=/home/carlos/iperf/src/iperf3
LOG_DIR=./iperf3-log
IMAGES_DIR=/home/carlos/criu-lm/experiments/tcp-established/images
CWD=$(pwd)

# Set up Environment
mkdir -p ${LOG_DIR}

# Run iPerf3 server
cd /home/carlos/runc-containers/iperf3-server
sudo runc run eureka &> /dev/null < /dev/null &
cd ${CWD}

sleep 3

```

```bash
# Run iPerf3 client in remote machine
echo "Bootstrapping Client..."
ssh carlos@${CLIENT_IP} "/home/carlos/tcp-established/iperf3_client.sh"

sleep 10

# CRIU Dump
echo "Dumping server..."
sudo runc checkpoint \
    --image-path ${IMAGES_DIR} \
    --tcp-established \
    eureka

sleep 2

# CRIU Restore
echo "Restoring server..."
cd /home/carlos/runc-containers/iperf3-server
sudo runc restore \
    --image-path ${IMAGES_DIR} \
    --tcp-established \
    eureka-restored
cd ${CWD}
```

Listing B.5: Full evaluation script for the TCP connection downtime micro-benchmark using `runC`.

```bash
#!/bin/bash
# Declare Variables
CLIENT_IP=192.168.56.103
IPERF3=/home/carlos/iperf/src/iperf3
LOG_DIR=./iperf3-log
IMAGES_DIR=/home/carlos/criu-lm/experiments/tcp-established/images
CWD=$(pwd)

# Set up Environment
mkdir -p ${LOG_DIR}

# Run iPerf3 server
cd /home/carlos/runc-containers/iperf3-server
sudo runc run eureka &> /dev/null < /dev/null &
cd ${CWD}

sleep 3

# Run iPerf3 client in remote machine
echo "Bootstrapping Client..."
ssh carlos@${CLIENT_IP} "/home/carlos/tcp-established/iperf3_client.sh"

sleep 4

# CRIU Dump
echo "Dumping server..."
(sudo runc checkpoint \
    --image-path ${IMAGES_DIR} \
    --tcp-established \
    eureka; \
cd /home/carlos/runc-containers/iperf3-server; \
sudo runc restore \
    --image-path ${IMAGES_DIR} \
    --tcp-established \
    eureka; \
cd ${CWD}) &

sleep 6

# CRIU Dump
echo "Dumping server..."
(sudo runc checkpoint \
    --image-path ${IMAGES_DIR} \
    --tcp-established \
    eureka; \
cd /home/carlos/runc-containers/iperf3-server; \
sudo runc restore \
    --image-path ${IMAGES_DIR} \
    --tcp-established \
    eureka; \
cd ${CWD}) &

sleep 4

# CRIU Dump
echo "Dumping server..."
(sudo runc checkpoint \
    --image-path ${IMAGES_DIR} \
    --tcp-established \
    eureka; \
cd /home/carlos/runc-containers/iperf3-server; \
sudo runc restore \
    --image-path ${IMAGES_DIR} \
    --tcp-established \
    eureka; \
cd ${CWD})
```

Listing B.6: Full evaluation script for the TCP connection reactivity micro-benchmark using `runC`.