Efficient Live Migration of Linux Containers

Honours Report

Radostin Stoyanov
51443372

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Engineering
of the
University of Aberdeen.

Department of Computing Science

May 04, 2018

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Radostin Stoyanov
May 04, 2018

ABSTRACT

The purpose of the thesis is to investigate live migration for Linux containers and implement a Proof-of-Concept live migration for the Libvirt-LXC driver. The container migration is based on a checkpoint/restore mechanism performed in userspace. Live migration is an important feature for many IT services and hosting providers to keep a low amount of hardware resources in use (dynamic load balancing, power saving), and to handle hardware and software failures without affecting running applications (fault tolerance, high availability).

Virtualisation is one of the fundamental technologies in cloud computing for improving resource utilisation. It supports the role of cloud computing as one of the most widely used technologies to provide numerous IT services. In recent years, the use of operating system-level virtualisation, also known as container-based virtualisation, has grown in popularity due to its capability to isolate multiple userspace environments and to allow for their co-existence within a single OS kernel instance. From a kernel point of view, a container is a process isolation feature - a group of processes are isolated from those in other containers, as well as processes of the host operating system.

Checkpoint-restore in Userspace (CRIU) is a tool that allows to store the CPU state of a hierarchy of processes (the container), along with all relevant memory pages, as a collection of image files. These files can be used to effectively restore this process hierarchy on the same (or different) physical computer, and continue their execution from the timepoint when the checkpoint was stored. This functionality is used to provide means for live migration of Linux containers.

Live migration of containers is the process of detaching a set of applications that run in the context of a container, transfer them to a remote host and reattach them back to the new OS kernel. Migration of Linux containers consumes a significant amount of time and resources, which results in performance overhead. This problem is specifically hard when applications running inside a container modify memory faster than it can be transferred over the network to a remote host.

In this work, we propose a novel approach to address this issue by utilising a recently published CRIU feature, the so-called “image-cache – image proxy” approach. This approach allows to simplify the implementation for live container migration by using the transfer mechanism of image-proxy, which is essentially a piping mechanism that sends image data from one host to the next without intermediate storage. In comparison with other implementations, such as, e.g., Docker, LXC and RunC, this approach is more efficient and results in lower total migration time and down time of the container applications that are migrated. The essential difference is that memory pages relevant to processes running inside a container are transferred to a remote host without performing expensive I/O operations. This approach is simpler because CRIU transfers

the CPU state and all memory pages directly from the local host to the remote host. The container runtime system only has to synchronise files associated with the container between local and remote host, and the container configurations.

As a starting point for this project, several existing migration algorithms and techniques that were proposed in the literature over the past thirty years, have been critically evaluated.

In this thesis, we used “image-cache” and “image-proxy” in combination with pre-copy and post-copy optimisations for live migration. The evaluation results show that this provides a great improvement when used in live migration. In particular, the evaluation results show that pre-copy preparation by this approach reduces the time it takes to freeze a process tree of a container to prepare it for migration and reduces the time it takes to transfer all the image files that are created by this pre-copy freeze. In terms of implementation, this approach provides means to reduce the complexity of the code performing live migration of containers and allows more control over the checkpoint and restore process.

The development of the live-migration features in this thesis has been characterised by an ongoing communication with the libvirt and CRIU development community. The results of this thesis will be merged into a future release of libvirt and, therefore, used in future cloud computing applications.

ACKNOWLEDGEMENTS

I would like to sincerely thank all those who have provided me with their time, assistance, and guidance during the course of this project, and throughout my degree, for which I am enormously grateful.

Firstly, I would like to thank my project supervisor Dr Martin Kollingbaum for his invaluable support and guidance throughout the process of developing this project.

Secondly, I would also like to thank Cedric Bosdonnat (cbosdo) and Michal Privoznik (mprivozn), who offered a huge amount of support, expertise, guidance and code reviews starting from my summer internship, through the project planning stages, and right through to the end of this project. There is no doubt that without their continued support this project would not have been possible.

Thirdly, I would like to thank Katerina Koukiou (kkoukiou), not only for her work which I was able to reuse, but also for completing her project to such a high standard. Katerina was also on-hand for any queries I had about the checkpoint/restore of Linux containers.

I am very grateful to Andrew Vagin (avagin), Adrian Reber (adrianreber), Christian Brauner (brauner), Kirill Tkhai (tkhai) and Pavel Emelyanov (xemul) who have responded on questions and helped me debug and resolve issues encountered during my work with CRIU and its integration with Libvirt.

I would also like to thank Daniel P. Berrangé (danpb), Cole Robinson (crobinso) and the rest of Libvirt, virt-tools and CRIU communities who have been extremely helpful.

I would also like to thank Naveed Khan and Michael Chung for their continued support and assistance with hardware resources used during this and other research projects over the course of my degree. I would like to acknowledge the efforts of the staff in the Computing Science department, and all those who have kindly imparted their knowledge and contributed to my education.

Finally, I would like to thank the server team at IT Services and the staff at Google Summer of Code, where I conducted my summer internships. Without the experience I gained whilst in their employ, I doubt this project would have been as detailed and involved as it was.

TERMINOLOGY

Program or an *application* is a set of machine code instructions and data stored in an executable image on disk and is a passive entity.
Process is a computer program in action that can carry out tasks within the operating system.
Linux container refers to a collection of processes, separated from the main host processes via a set of resource namespaces and constrained via control group resource tunables.
Linux namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace as if they have their own isolated instance of a global resource, for example, the process identifiers of the host system are unique for each PID namespace. Changes to the global resource are visible to processes that are members of the namespace, but are invisible to other processes outside that namespace.
The terms *server*, *host*, and *node* are used interchangeably throughout this thesis. Generally a server refers to a physical or virtual unit that runs software dedicated to a specific purpose. A node is a server within a larger group, typically a software cluster or a rack of servers, and has its own unique IP address.
The terms *hypervisor* and *virtual machine manager (VMM)* are used interchangeably throughout this thesis and both refer to a layer of software allowing to virtualise a node in a set of virtual machines with possibly different configurations than the node itself.
A *domain* is an instance of an operating system (or subsystem in the case of container virtualisation) running on a virtualised machine provided by the hypervisor.
In this thesis the terms *checkpoint*, *dump* and *save* are used interchangeably and refer to saving a state of running processes to a set of image files.
The phrases *process restore* and *process restart* refer to the creation of a process from a previously stored set of image files (checkpoint).

NOTATIONAL CONVENTIONS

The notational conventions used throughout this report are described here.

Symbol	**Full name**	**Derivation**
KiB	kibibyte	2^{10} byte == 1024 byte
MiB	mebibyte	2^{20} byte == 1048576 byte
GiB	gibibyte	2^{30} byte == 1073741824 byte
KiB/s	kibibyte per second	2^{10} byte / second
MiB/s	mebibyte per second	2^{20} byte / second
GiB/s	gibibyte per second	2^{30} byte / second
kHz	kilohertz	10^3 Hz
MHz	megahertz	10^6 Hz
GHz	gigahertz	10^9 Hz

Table 1: Notational conventions

ABBREVIATIONS

ALU	Arithmetic Logic Unit
API	Application Program Interface
AppArmor	Application Armor
CGROUP	Control Groups
COW	Copy-on-Write
CPU	Central Processing Unit
CRIU	Checkpoint/Restore in Userspace
DNS	Domain Name System
EGID	Effective Group Identifier
EUID	Effective User Identifier
FPU	Floating-point Unit
GID	Group Identifier
HPC	High Performance Computing
HugeTLB	Huge page tables
IaaS	Infrastructure as a Service
IPC	Inter Process Communication
IRMAP	Inode Reverse MAP
iSCSI	Internet Small Computer Systems Interface
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
LXC	Linux Container
NFS	Network File System
NIS	Network Information Service
NUMA	Non Uniform Memory Access
OS	Operating System
PaaS	Platform as a service
PID	Process Identifier
POSIX	Portable Operating System Interface
PTE	Page Table Entry
PTMS	Pseudoterminal Master & Slave
PTY	Pseudoterminal Interface
QEMU	Quick Emulator
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RGID	Real Group Identifier
ROOTFS	Root File System
RPC	Remote Procedure Call
RUID	Real User Identifier
SaaS	Software as a service
SELinux	Security-Enhanced Linux
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TMPFS	Temporary File System
TTY	Teletypewriter
UDP	User Datagram Protocol
UID	User Identifier
URI	Uniform Resource Identifier
UTS	UNIX Timesharing System
vCPU	Virtual Central Processing Unit
VE	Virtual Environment
VM	Virtual Machine
VMA	Virtual Memory Address
VMM	Virtual Machine Monitor
VPS	Virtual Private Server
XML	Extensible Markup Language

Contents

1 Introduction **12**

1.1 Overview . . . 12
1.2 Problem identification and motivation . . . 14
1.3 Objectives . . . 17
1.4 Contributions . . . 18

2 Background **19**

2.1 Virtualisation Technologies . . . 19
2.1.1 Kernel Level Process Migration . . . 19
2.1.2 Virtual Machine Migration . . . 21
2.2 Container-Based Virtualisation . . . 24
2.2.1 Linux Namespaces . . . 24
2.2.2 Linux Control Groups . . . 25
2.2.3 Cgroups Version 1 versus Version 2 . . . 26
2.2.4 Cgroups controllers . . . 26
2.3 Alternative Technologies - Linux Containers . . . 27
2.3.1 Docker . . . 27
2.3.2 Docker Images and Layers . . . 28
2.3.3 LXC/LXD . . . 29
2.3.4 OpenVZ . . . 29
2.4 Domain Management with Libvirt . . . 30
2.5 Checkpoint and Restore . . . 30
2.6 Basics and Architecture of CRIU . . . 32
2.6.1 Pre-copy migration with CRIU . . . 32
2.6.2 Post-copy migration with CRIU . . . 32
2.7 Special File Systems . . . 34
2.7.1 Temporary file system (tmpfs) . . . 34
2.7.2 Proc file system (procfs) . . . 34
2.7.3 Overlay File system (overlayfs) . . . 35

3 Design **36**

3.1 Requirements . . . 36
3.1.1 Scenarios . . . 36
3.1.2 Functional Requirements . . . 36

3.1.3 Non-functional Requirements . . . 37
3.2 Libvirt . . . 37
3.2.1 Architecture & internals . . . 38
3.2.2 Libvirt-LXC Driver . . . 40
3.3 Container Monitor . . . 41
3.4 Pivot root . . . 41
3.5 Process Creation . . . 41
3.5.1 Zombie Processes . . . 41
3.5.2 Orphan Processes . . . 42
3.5.3 User and Group Identifiers . . . 42

4 Implementation 43
4.1 LXC Driver Architecture . . . 43
4.1.1 Integration of CRIU with Libvirt-LXC . . . 44
4.1.2 Configuration Parameters . . . 45
4.2 Container Checkpoint and Restore . . . 45
4.3 Container Migration . . . 46
4.3.1 Native live migration . . . 47
4.3.2 Pre-copy live migration . . . 47
4.3.3 Post-copy live migration . . . 50
4.3.4 Image-cache and Image-proxy live migration . . . 50
4.3.5 Migration Parameters . . . 51
4.4 Regression Testing . . . 51
4.5 Programming Style . . . 52
4.5.1 Naming conventions . . . 52
4.5.2 Bracket spacing . . . 53
4.5.3 Commas . . . 53
4.5.4 Semicolons . . . 53
4.5.5 Curly braces . . . 54

5 Evaluation 55
5.1 Experimental Design . . . 55
5.1.1 Test Environment . . . 55
5.1.2 Experimental Setup . . . 55
5.2 Results . . . 56
5.3 Analysis . . . 59
5.4 Interpretation of Results . . . 62

6 Discussion & Explanation of Difficulties Found 63
6.1 CRIU Limitations . . . 63
6.2 Limitations of Linux Containers . . . 64
6.3 Limitations of Pre-copy Live Migration . . . 64
6.4 Limitations of Post-copy Live Migration . . . 65

6.5 Limitations of image-cache & image-proxy 65
6.6 Independent and Dependent Variables 66

7 Summary and Conclusion **67**
7.1 Project Summary & Critical Evaluation 67
7.2 Future Work Suggestions 67
7.3 Conclusion & Personal Reflection 68

A User Manual **70**
A.1 Virsh command line tool 70
A.2 Creating a Container 71
A.3 Save & Restore Container State 74
A.4 Live Migrate Container 74

B Maintenance Manual **76**
B.1 Installation instructions 76
B.1.1 Installing Libvirt 76
B.1.2 Installing CRIU 76
B.2 Software dependencies 76
B.2.1 Libvirt dependencies 76
B.2.2 CRIU dependencies 78
B.3 Future adaptations and extensions 79

C Pre-copy Migration **80**

D Post-copy Migration **83**

E Page-server Migration **85**

F Image-cache & Image-proxy Migration **87**

Chapter 1

Introduction

1.1 Overview

Live migrating applications and operating system instances across distinct physical hosts enables a variety of benefits, such as dynamic load balancing, fault tolerance, and data access locality. It allows a clean separation between hardware and software, and makes easy low-level system maintenance. Migration can be used to help optimise several aspects of operations such as power efficiency. Virtual environments (VMs and containers) can be gathered on a single physical machine to allow unused hardware resources to be suspended or switched off.

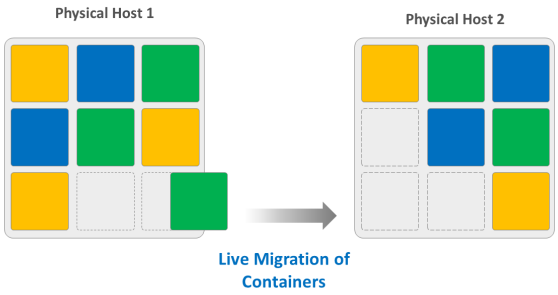


Figure 1.1: Live Migration of Containers

The algorithm that transfers a process from one machine to another has the highest influence on the performance of a process-migration facility. In the past thirty years, many process migration algorithms have been proposed in the literature, such as Pre-copy [104], Flushing [40], Post-copy [94], Freeze Free [96], Hybrid-copy [58, 49], Copy-on-reference [109, 108], Queued Pre-copy [80], Assisted Post-copy [80], Auto-convergence [21], Post-copy via Active Pushing [49], Post-copy via Pre-paging [49]. Some of them are discussed in detail in chapter 2 and chapter 4.

Server virtualisation has proven to be a valuable tool to increase resource efficiency and simplify data centre management as well as improve service reliability, fault tolerance and security.

Hardware virtualisation is based on the concept of Virtual Machines (VMs) that are controlled by a *Hypervisor* or *Virtual Machine Monitor (VMM)*. A VM was formally defined for a first time in [87] to be *an efficient, isolated duplicate of the real machine*. As a piece of software, a VMM has three essential characteristics:

1. The VMM provides an environment for programs that is essentially identical to the original

machine.

2. Programs that run in this environment show at worst only minor decreases in speed.

3. The VMM is in complete control of system resources.

Modern virtual machine monitors have the ability to run a large number of virtual machines on a single physical server. This enables consolidation of multiple under-utilised servers inside a single one, while preserving the property of failure and resource isolation between VM instances.

Another type of virtualisation is the operating system level virtualisation, also known as container-based virtualisation, lightweight virtualisation [74], BSD jails [54] or Solaris zones [38]. It has similar characteristics and is used to partition a single physical machine resource into multiple isolated user-space instances that share the same OS kernel. Such virtual environments are often called *containers*.

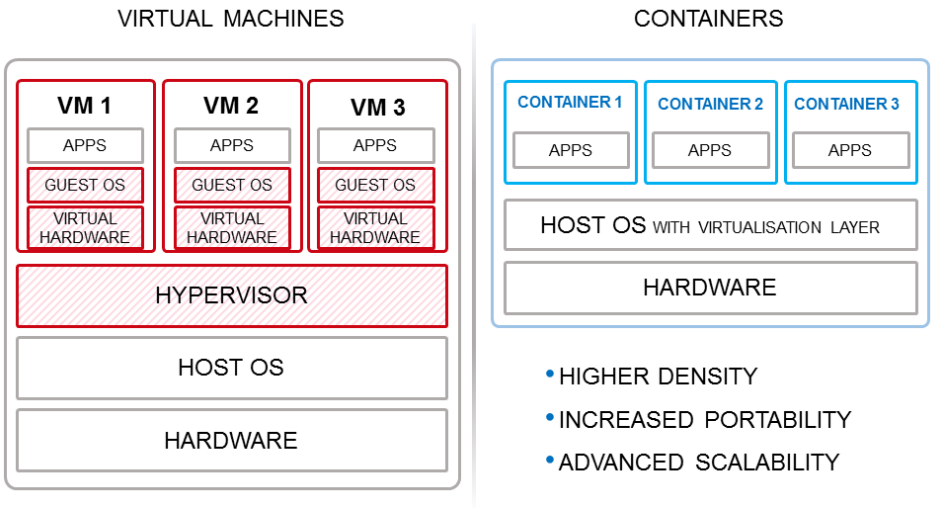


Figure 1.2: Virtual Machines VS Containers

One disadvantage of virtual machine hypervisors is the introduced performance overhead in terms of virtualised hardware devices and secondary kernel instance dedicated to each VM [74, 42]. In comparison with hypervisors, operating system level virtualisation (containers) provide a different level of abstraction in terms of isolation [99]. The fundamental difference between VMs and containers is that all container instances run on top of a single kernel instance (as shown in Figure 1.2). Due to the shared kernel container-based solutions have the advantage of higher density and lower performance overhead.

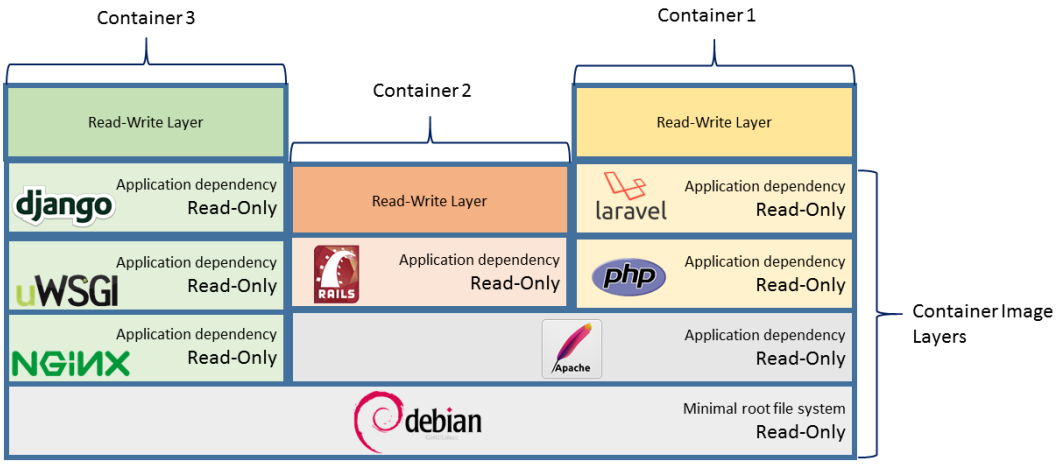


Figure 1.3: OverlayFS Layers of Container Images

Another difference is the way VMs and containers package an OS instance. A container image consists of a set of tar archives that store only the minimal set of files (root file system) needed for a containerised application to work [52]. In contrast to containers, virtual machines utilise disk images that contain the whole structure of a disk volume or of an entire data storage device.

The essential distinction between container images and VM images is the concept of layers. In container images, every tar archive represents a different layer and every layer contains only file changes with regards to a previous layer. A base layer (the first layer) of a an image stores the initial set of files and it is used as read-only by other above layers (See Figure 1.3).

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. Each container is characterised by its own Linux-specific mount namespace [1] When a container is running, the data from all layers is used as read-only and all file changes that are made inside the container are kept in an isolated read-write layer that is only visible to the mount namespace of the container. This architecture avoids data redundancy by allowing multiple container images to share identical layers without duplication of data.

From the perspective of containerised application, the virtual environment is exactly like a stand-alone server. A container can be rebooted independently and have root access. It also has its own users and groups, IP address(es), memory, processes, root file system, etc. From a kernel point of view, a container is a separate set of processes completely isolated from the other containers and the host system [73].

1.2 Problem identification and motivation

In the last decade, there has been a significant increase of web services that provide social media, video streaming, online banking, trading, shopping, etc [50]. Downtime for such services has big financial impact on the corporation responsible. For example, it is unacceptable for social media

[1] Linux maintains 7 so-called namespaces, of which a mount namespace is one of them, where "mount" points to the fact that the container file system is isolated (invisible) from the host file system, but is mounted under `/proc`.

services, such as Facebook [2], Twitter [3] or Snapchat [4], to become unavailable for maintenance, though this has been a common practice up until recently and still is for some online services (e.g. Blackboard [5]). Live migration is very important for online shopping services and downtime can have significant financial impact.

Live migration has an important role for cloud providers (SaaS, PaaS and IaaS) because it does not change any attributes or properties of the applications running inside a container or a VM. An example of such service provides is Google Compute Engine [6]. It utilises live migration to keep application instances running even when a host system event occurs, such as a software or hardware update. Common scenarios when live migration is necessary include regular infrastructure maintenance, upgrades, network and power grid maintenance in the data centres, failed hardware (e.g. memory, CPU, network interface cards, disks, power, etc), host OS and BIOS upgrades, security-related updates that require quick response, system configuration changes (e.g. changing the size of the host root partition).

In order to comply with given QoS (Quality of Service) requirements when providing services to clients, cloud and web service providers need to allocate enough resources that meet the demand. However, it is difficult to maintain these performance guarantees due to the fact that the workload is unpredictable. The performance of live migration depends on the down time and total migration time of the migrated application. Total migration time is the time required to transfer memory pages and CPU state from source to destination host. Downtime is the time between the suspension of migrated application at the source and its resumption at the target node [34]. Satisfying QoS requirements of different types of requests and minimising the amount of downtime of such services requires them to utilise high-availability clustered deployments on large scale.

Lightweight virtualisation allows complex applications to be decomposed into multiple small, independent processes (e.g. web-server, database server, load balancer, etc.), known as "micro-services". Each micro-service runs in its own isolated container that includes only necessary libraries and files for it to operate (See Figure 1.4). The microservices architecture enables the development of loosely coupled services that can be rapidly scaled and guarantee fault tolerance by efficiently managing replications of services, for example, utilising master-slave configuration.

[2] https://facebook.com/
[3] https://twitter.com/
[4] https://snapchat.com/
[5] https://blackboard.com/
[6] https://cloud.google.com/compute/docs/instances/live-migration

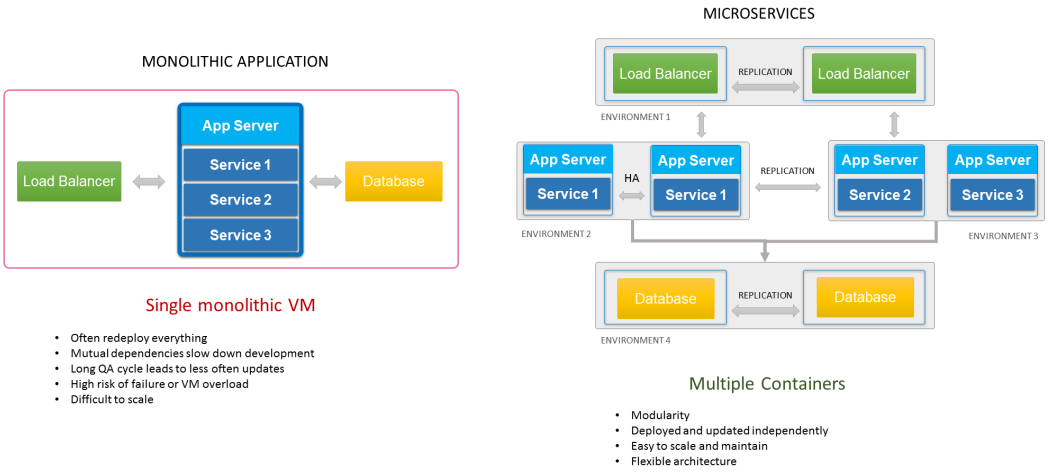


Figure 1.4: Monolithic VS Micro-services

Container-based virtualisation receives a significant amount of attention from projects such as Docker [7], Google Kubernetes [8], LXC/LXD [9], Atomic [10] and OpenShift [11], that are responding with innovation to make the lightweight virtualisation approach ready for production environments. In parallel to these large scale projects, a number of companies are developing and releasing container tools like the ones show in table **??**, and container focused operating systems such as CoreOS [12] and RancherOS [13].

Live migrating virtual environment instances across distinct physical hosts allows a clean separation between hardware and software, and provides means for fault tolerance, dynamic load balancing, and low-level system maintenance [22]. Migrating an entire OS and all of its applications as one unit avoids many of the complications faced by process-level migration approaches. In particular, it makes it easy to avoid the problem of "residual dependencies" [72] that requires the original host machine to remain available and network-accessible in order to service system calls or even memory access on behalf of the migrated process.

Live migration of VMs was first demonstrated by [22] and since then has become a standard feature of hypervisors. It is an active field of research. In recent versions of the Linux have been merged patches that make possible checkpoint/restore of precesses in user space. Several container-type virtualisation platforms have utilised this new functionality to make possible taking a stateful snapshot of a container as well as live migration. In conjunction with migration mechanisms, container runtime systems allow for the facilitation of load balancing strategies, an improvement of fault resiliencies, a simplified system administration, and bring advantages such as low start-up time and higher density.

[7] `https://www.docker.com/` - Open platform for developers and system administrators to build, ship, and run distributed applications.

[8] `https://kubernetes.io/` - Production-Grade Container Orchestration.

[9] `https://linuxcontainers.org/` - Environment for the development of Linux container technologies.

[10] `https://www.projectatomic.io/` - A lightweight container OS using the LDK (Linux, Docker, Kubernetes) stack.

[11] `https://www.openshift.com/` - Container Application Platform.

[12] `https://coreos.com/` - An open-source Linux-based OS providing infrastructure for clustered deployments.

[13] `https://rancher.com/rancher-os/` - Open-source Linux distribution designed to run Docker in production.

Project	URL	Description
Etcd	`https://coreos.com/etcd/`	Distributed reliable key-value store for the most critical data of a distributed system.
Prometheus	`https://prometheus.io/`	An open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach.
Fluentd	`https://www.fluentd.org/`	An open source data collector for unified logging layer.
CoreDNS	`https://coredns.io/`	DNS server that chains plugins.
OpenTracing	`http://opentracing.io/`	Consistent, expressive, vendor-neutral APIs for distributed tracing and context propagation.
CNI	`https://github.com/containernetworking/cni`	Networking for Linux containers.
Envoy	`https://www.envoyproxy.io/`	An open source edge and service proxy, designed for cloud-native applications.

Table 1.2: Open Source Container-Based Tools

1.3 Objectives

Live migration of containers enables:

- **Dynamic load distribution**, by migrating containers from overloaded nodes to less loaded ones.
- **Fault resilience**, by migrating containers from nodes that may have experienced a partial failure.
- **Improved system administration** by migrating containers from nodes that are about to be shut down or otherwise made unavailable.
- **Data access locality**, by migrating processes closer to the source of some data.

A Linux container is composed of isolated set of processes (a single process tree) that can be checkpointed and restored to the same or a different physical machine. The checkpoint and restore mechanisms can be regarded as generalisation of process migration. Restoring the running state of a container, from a checkpoint, on a different node, than the one the checkpoint was taken on, is effectively a migration of that container.

Live migration of a whole container is less complex than a migration of a single process. This is due to the fact that containers have their own root file system that keeps all files (e.g. programs, libraries, configurations, etc.), needed for applications inside the container to operate. This allows them to be easily synchronised (e.g. with rsync) to remote host, as well as to keep track of changes. In addition all processes running inside a container are grouped in an isolated IPC namespace that allows easy suspend and restore of inter-process communication between all applications inside the container.

1.4 Contributions

This thesis builds on previous work from Katerina Koukiou [103] and Rodrigo Bruno [15]. Katerina has implemented a proof-of-concept (PoC) of save and restore functionalities for the Libvirt-LXC driver. The previous work from Rodrigo introduced an extension for the CRIU tool that enables caching of image files (the precess snapshot) in memory rather than storing them on persistent storage. This essentially provides the image-cache and image-proxy mechanism that we used for live migration of Linux containers.

The contributions of this work extend Katerina's PoC to a working solution compatible with the current master branch of Libvirt, in a format suitable to be upstreamed. In addition, this project resolves and upstreamed the solutions for several issues related to the integration of CRIU with Libvirt. These contributions improve the restore functionality of CRIU that is essential for live container migration.

Chapter 2

Background

This section discusses previous work from the literature, explains the basic concepts and technologies used throughout the project, and technical challenges with regards to virtualisation, live migration and checkpoint-restart.

2.1 Virtualisation Technologies

Virtualisation technologies have predominant role in recent years, and the number of software solutions is increasing rapidly. One of the reasons for adopting and deploying these advanced technologies, and to build new paradigms in this field is to keep up with the growth of internet and the increased amount of web-services and exchanged data. A consequent need to increase the capability of data centres by means of server virtualisation.

Although virtualisation is still an active area of research, the first work dates as far back as the 1970s, when IBM introduced a method to logically divide system resources provided by mainframe computers between different applications or users [70]. Time sharing solutions were very important at that time as computer resources were not affordable even for large organisations. During 1980s and 1990s, expensive stand alone mainframes were replaced with clusters of workstations as a high-performance facility. The first appearance of such distributed computing environments introduced a new problem with the utilisation of the additional processing power. A lot of the performance capacity was not being used because machines were generally idle [80] and this introduced a demand for dynamic allocation and re-allocation of computational resources. Nowadays, the need of virtualisation emerges from the fact that the capacity in a single server is so large that it is almost impossible for most workloads to effectively use it.

2.1.1 Kernel Level Process Migration

A *process* is an operating system abstraction representing an instance of a running computer program. Process migration is the act of transferring a process between two computers during execution [57]. This is commonly used method to achieve dynamic load balancing. It differs from *static load balancing*, where the performance of a server is determined at the commencement of execution, and once the load is allocated to a server, it cannot be transferred to a different one [72].

In many distributed operating systems it is possible to transfer data, such as files, between various physical machines. Some of them additionally allow the movement of the process's execution site, a mechanism known as *process migration*. Process migration research started with the appearance of distributed processing among multiple processors and introduced the opportunity for sharing processing power and other resources, such as memory and communication channels.

Kernel level process migration techniques modify the operating system kernel to make process migration simpler and more efficient. Kernel modifications allow for migration to be achieved faster and to migrate different types of processes. Unfortunately, many implementations from the past have high overhead, long freeze times, and still cannot migrate all types of processes [57]. Several implementations presented in the literate, that enable process migration have significant impact on the technologies nowadays – Accent [92], Amoeba [76], Arcade [24], Charlotte [10], Chorus [26], Clouds [35], Demos/MP [89], Locus [106], Mach [8], Mosix [12], OSF/1 AD TNC [107], RHODOS [36], Sprite [40], V-System [104] and XOS [71]. All these implementations have similarities in the approach used for live migration, but they differ in the transfer mechanism used for virtual memory pages. A high-level description of the migration algorithms is illustrated in Figure 2.1.

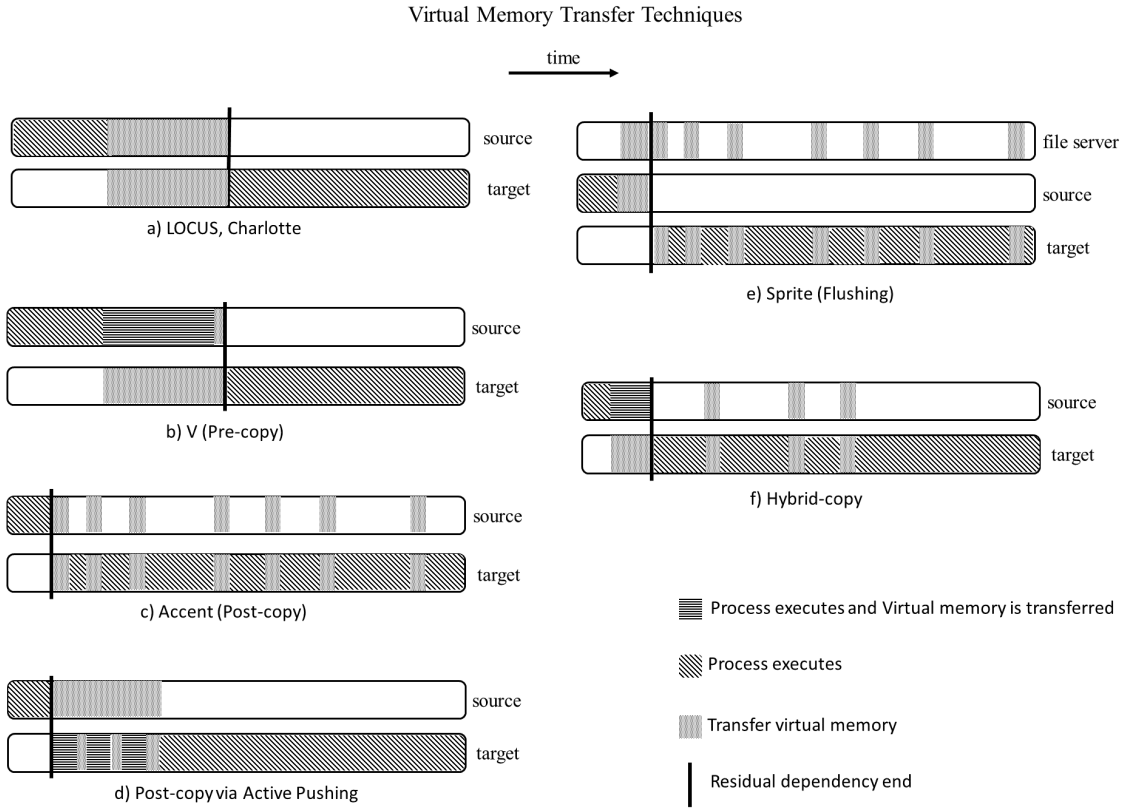


Figure 2.1: Different techniques for transferring virtual memory during process migration

Several of these implementations are discussed in detail in chapter 4, but they all have in common the property of being distributed operating systems. All of the above distributed operating systems utilise one of the process migration algorithms discussed in section 4.3. A few of them have a very important contribution - process migration in System-V was revolutionary because it did not transfer the whole address space at migration time; the Accent implementation was very important because it represented for first time the demand-paging of the address space (the Post-copy algorithm), and the Sprite implementation was also important as it introduced the distributed file system support for process migration. All implementations of the above operating systems features influenced the development of computer technologies used nowadays like: distributed file

systems (e.g. NFS, GlusterFS), remote execution (e.g. RPC, RMI), and particular aspects of OS architectures such as memory management and process scheduling.

2.1.2 Virtual Machine Migration

Research on virtual machines on scalable shared memory multiprocessors [16] presents a potential for migration of whole virtual machine between processors of a multiprocessor, that abstracts away most of the complexities of operating systems. This approach is reducing the migratable state only to memory and to state contained in a virtual machine monitor [46]. Therefore, migration is easier to implement with existing virtual machine. The effectiveness of this approach has been demonstrated in [22].

The algorithm that transfers the process from one machine to another has the highest influence on the performance of a process-migration facility. In the past years, many process-migration algorithms have been proposed in the literature, such as Pre-Copy [104], Post-Copy [94], Flushing [40], Freeze Free [96]. Although there are many different migration implementations and designs, most of them can be summarized in the following steps.

1. **A migration request is issued to a remote node.** After negotiation, migration has been accepted.
2. **A process is detached from its source node** by suspending its execution, declaring it to be in a migrating state, and temporarily redirecting communication as described in the following step.
3. **Communication is temporarily redirected** by queuing up arriving messages directed to the migrated process, and by delivering them to the process after migration. This step continues in parallel with steps 4, 5 and 6, as long as there are additional incoming messages. Once the communication channels are enabled after migration (as a result from step 7), the migrated process is known to the external world.
4. **The process state is extracted,** including memory contents; processor state (register contents); communication state (e.g., opened files and message channels); and relevant kernel context. The communication state and kernel context are OS-dependent. Some of the local OS internal state is not transferable. The process state is typically retained on the source node until the end of migration, and in some systems it remains there even after migration completes. Processor dependencies, such as register and stack contents, have to be eliminated in the case of heterogeneous migration.
5. **A destination process instance is created into which the transferred state will be imported.** A destination instance is not activated until a sufficient amount of state has been transferred from the source process instance. After that, the destination instance will be promoted into a regular process.
6. **State is transferred and imported into a new instance on the remote node.** Not all of the state needs to be transferred; some of the state could be lazily brought over after migration is completed.
7. **Some means of forwarding references to the migrated process must be maintained.** This is required in order to communicate with the process or to control it. It can be achieved

by registering the current location at the *home* node (e.g. in Sprite [40]) by searching for the migrated process (in the V Kernel [104], at the communication protocol level), or by forwarding messages across all visited nodes (e.g. in Charlotte [10]). This step also enables migrated communication channels at the destination and it ends step 3 as communication is permanently redirected.

8. **The new instance is resumed** when sufficient state has been transferred and imported. With this step, process migration completes. Once all of the state has been transferred from the original instance, it may be deleted on the source node.

Given the relative complexity of implementation, and the expense incurred when process migration is invoked, researchers often choose to implement alternative mechanisms.

- **Remote execution** is the most frequently used alternative to process migration. Remote execution can be as simple as the invocation of some code on a remote node, or it can involve transferring the code to the remote node and inheriting some of the process environment, such as variables and opened files. Remote execution is usually faster than migration because it does not incur the cost of transferring a potentially large process state (such as the address space, which is created anew in the case of remote execution).

  Remote execution has disadvantages as well. It allows creation of the remote instance only at the time of process creation, as opposed to process migration which allows moving the process at an arbitrary time. Allowing a process to run on the source node for some period of time is advantageous in some respects. This way, short-lived processes that are not worth migrating are naturally filtered out. Also, the longer a process runs, the more information about its behaviour is available, such as whether and with whom it communicates. Based on this additional information, scheduling policies can make more appropriate decisions.

- **Cloning processes** is useful in cases where the child process inherits state from the parent process. Cloning is typically achieved using a remote fork mechanism. A remote *fork*, followed by the termination of the parent, resembles process migration. The complexity of cloning processes is similar to migration, because the same amount of the process state is inherited (e.g. open files and address space). In the case of migration, the parent is terminated. In the case of cloning, both parent and child may continue to access the same state, introducing distributed shared state, which is typically complex and costly to maintain.

- **Mobile agents** are becoming increasingly popular. The mobility of agents on the Web emphasizes safety and security issues more than complexity, performance, transparency and heterogeneity. Mobile agents are implemented on top of safe languages, such as Java [11], Telescript [39] and Tcl/Tk [82]. Compared to process migration, mobile agents have reduced implementation complexity because they do not have to support OS semantics. Performance requirements are relaxed due to the wide-area network communication cost, which is the dominant factor. Heterogeneity is abstracted away at the language level. The early results and opportunities for deployment, as well as the wide interest in the area of mobile agents, indicate a promising future for this form of mobility. However, the issues of security, social acceptance, and commercialisable applications have been significantly increased and they represent the main focus of research in the mobile agent community.

There are several issues with process migration that prevented widespread use despite the ongoing research efforts. One reason for this is the complexity of adding transparent migration to systems originally designed to run stand-alone. The complexity of implementation and dependency on an operating system are among the biggest obstacles, especially for fully-transparent migration implementations. Migration can be applied on different level, as part of the operating system kernel, in user space, as part of the system environment, or as part of the application. Implementations on different level result in different performance, complexity, transparency and reusability.

User-level migration typically yields simpler implementations, but suffers too much from reduced performance and transparency to be of general use for load distribution. [65] Migration implemented as part of an application can have poor reusability if modifications are required to the application, as was demonstrated in [44] and [98]. Low level migration is more complex to implement, but has better performance, transparency and reusability.

Another reason is that there has not been a compelling commercial argument for operating system vendors to support process migration. Checkpoint-restart approaches offer a compromise here, since they can run on more loosely-coupled systems by restricting the types of processes that can migrate. Today the term *virtualisation* is widely applied to a number of concepts including:

- **Hardware virtualisation** refers to the creation of a VM that acts like a physical computer with an independent operating system instance. Such virtual machine is commonly referred as a *guest*, and the actual physical machine is referred as a *host*. The software that facilitates the creation and management of the VMs is called a *hypervisor* or a *virtual machine monitor*. There are three basic types of hardware virtualisation regarding the level of awareness of the guest operating system:
  - **Full virtualisation**: The guest operating system is unaware that it is in a virtualised environment and therefore hardware is virtualised by the host operating system so that the guest can issue commands to what it thinks is actual hardware, but really are just simulated hardware devices created by the host.
  - **Partial virtualisation**: Some but not all of the target environment attributes are simulated. As a result, some guest programs may need modifications to run in such virtual environments.
  - **Para-virtualisation**: The guest operating system is aware that it is a guest and accordingly has drivers that, instead of issuing hardware commands, simply issues commands directly to the host operating system. This technique result in performance enhancement for operations performed by the guest OS.
- **Hardware-assisted virtualisation** improves overall efficiency of VMs. It involves CPUs that provide support for virtualisation in hardware, and other hardware components that help improve the performance of a guest environment.
- **Operating system-level virtualisation** is a server virtualisation method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one. Such instances are often called containers.

- **Application-level virtualisation** is a software technology that encapsulates computer programs from the underlying operating system on which they execute. Such applications directly interface the original operating system and all the resources managed by it.
- **Network virtualisation** is the process of combining hardware and software network resources into a single software-based administrative entity, known as virtual network.
- **Storage virtualisation** is the process of abstracting logical from physical storage.
- **Virtual memory** is a memory management technique that provides the illusion of contiguous working memory by abstraction the underlying physical memory implementation.

2.2 Container-Based Virtualisation

In this section, we discuss in more detail the operating system level virtualisation and the concept of containers.

This virtualisation approach creates an impression of multiple operating systems by using isolation and control mechanisms for the separation of user-space instances, i.e., containers. In contrast to hypervisor-based virtualisation, the abstraction is provided at the system call level, i.e., a single kernel is shared among all containers, reducing the overhead of multiple kernel instances running at the same time. Therefore, containers often could be used as an lightweight alternative to VMs, especially for HPC workloads.

2.2.1 Linux Namespaces

In the Linux kernel, a *namespace* is a concept that is used to partition kernel resources as an abstraction for a set of processes. This allows a global system resource (e.g. PIDs, IPCs, UIDs/GIDs, network interfaces, mount points etc.) to appear as an isolated instance for one set of processes that are grouped within a namespace [77]. Namespaces enable isolation and partition of specific system resources. This kernel feature is a fundamental security primitive used to implement Linux containers.

A namespace provides means for isolation of a group of processes (running within a container) from other processes running on the host OS, without any overhead from virtualisation.

The Linux kernel provides the following namespaces:

- **Cgroup** namespace [20] virtualise the view of a process's cgroups. Each cgroup namespace has its own set of cgroup root directories. These root directories are the base points for the relative locations displayed in the corresponding records in the `/proc/PID/cgroup` file.
- **IPC** namespace isolates the Inter-Process Communication resources, namely, System V IPC objects [101] and POSIX message queues [88]. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue file system. Objects created in an IPC namespace are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces [77]. This allows all process communication inside a container to be paused and restored during live migration.
- **Network** namespaces [78] provides isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the /proc/net directory (which is a symbolic link to /proc/PID/net), the /sys/class/net directory, various files under /proc/sys/net, port numbers (sockets), and so on.

- **Mount** namespace [75] provides isolation of the list of mount points seen by a processes in each namespace instance. Thus, the processes in each of the mount namespace instances will see distinct single directory hierarchies.
- **PID** namespace makes possible the separation of different process groups. It allows to isolate the process ID number space, meaning that processes in different PID namespaces can have the same PID [85]. This is important to facilitates migration of containers across nodes since the IDs of the processes running therein do not have to be adapted on the target subsystems.
- **User** namespaces [105] isolate security-related identifiers and attributes, in particular, UIDs and GIDs [28], the root directory, keys [55], and capabilities [18]. A process's UIDs/GIDs can be different inside and outside a user namespace. In particular, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.
- **UTS** [77] namespaces provides isolation of two system identifiers: the host name and the NIS domain name.

2.2.2 Linux Control Groups

Generic process control groups were implemented and merged into the upstream Linux kernel in 2006 [69]. This introduced the *cpusets* mechanism [83] and moved concept of containerization forward. This patch set was the minimal intrusive changes with little impact on performance, code quality, complexity, and future compatibility. The result was generic process containers. The name *control groups*, or *cgroups* was chosen to reflect the fact that "this code is an important part of a container solution, but it's far from the whole thing" [25]. CGroups allow a set of processes to be grouped together, and ensure that each group receives an allocated share of resources (i.e. memory, CPU, disk I/O, etc).

In contrast to namespaces, control groups provide a mechanism for aggregating and partitioning sets of tasks, and all their future children, into hierarchical groups with specialised behaviour [84] . This becomes especially important when scheduling multiple Linux containers concurrently on the same node.

A *cgroup* associates a set of tasks with a set of parameters for one or more subsystems. A *subsystem* is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes.

A *hierarchy* is a set of cgroups arranged in a tree, such that every task in the system is in exactly one of the cgroups in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each cgroup in the hierarchy. Each hierarchy has an instance of the cgroup virtual file system associated with it.

For instance, cpusets use the generic cgroup subsystem to provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks. A "memory node" refers to an on-line node that contains memory. Cpusets constrain the CPU and Memory placement of tasks to only the

resources within a task's current cpuset. They form a nested hierarchy visible in a virtual file system. These are the essential hooks, beyond what is already present, required to manage dynamic job placement on large systems.

2.2.3 Cgroups Version 1 versus Version 2

The initial release of the cgroups implementation was in Linux 2.6.24. Over time, various cgroup controllers have been added to allow the management of various types of resources. However, the development of these controllers was largely uncoordinated, with the result that many inconsistencies arose between controllers and management of the cgroup hierarchies became rather complex [47]. Because of the problems with the initial cgroups implementation (cgroups version 1), starting in Linux 3.10, work began on a new, orthogonal implementation to remedy these problems. Initially marked experimental, and hidden behind the `-o __DEVEL__sane_behavior` mount option, the new version (cgroups version 2) was eventually made official with the release of Linux 4.5.

2.2.4 Cgroups controllers

Each of the cgroups version 1 controllers is governed by a kernel configuration option. Additionally, the availability of the cgroups feature is governed by the `CONFIG_CGROUPS` kernel configuration option.

- **cpu** - is used to can be guaranteed a minimum number of "CPU shares" [19] for a cgroup when a system is busy. This does not limit a cgroup's CPU usage if the CPUs are not busy.
- **cpuacct** - CPU Accounting Controller account CPU usage for groups of tasks [27].
- **cpuset** - controller provides a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks [83].
- **memory** - controller isolates the memory behaviour of a group of tasks from the rest of the system [68].
- **devices** - This controller associates a device access white list with each cgroup. It tracks and enforces *open* and *mknod* restrictions on device files [37].
- **freezer** - This cgroup is useful to batch job management system which start and stop sets of tasks in order to schedule the resources of a machine according to the desires of a system administrator [45]. This sort of program is often used on HPC clusters to schedule access to the cluster as a whole. The cgroup freezer uses cgroups to describe the set of tasks to be started/stopped by the batch job management system. It also provides a means to start and stop the tasks composing the job.

  The cgroup freezer will also be useful for checkpointing running groups of tasks. The freezer allows the checkpoint code to obtain a consistent image of the tasks by attempting to force the tasks in a cgroup into a quiescent state. Once the tasks are quiescent another task can walk `/proc` or invoke a kernel interface to gather information about the quiesced tasks. Checkpointed tasks can be restarted later should a recoverable error occur. This also allows the checkpointed tasks to be migrated between nodes in a cluster by copying the gathered information to another node and restarting the tasks there.

- **blkio** - Block IO Controller [13] controls and limits access to specified block devices by applying IO control in the form of throttling and upper limits against leaf nodes and intermediate nodes in the storage hierarchy.
- **perf_event** - This controller allows performance monitoring of the set of processes grouped in a cgroup.
- **net_cls** - Network classifier cgroup provides an interface to tag network packets with a class identifier. The Traffic Controller (tc) can be used to assign different priorities to packets from different cgroups. Also, Netfilter (iptables) can use this tag to perform actions on such packets.
- **net_prio** - Network priority cgroup provides an interface to allow an administrator to dynamically set the priority of network traffic generated by various applications.
- **hugetlb** - HugeTLB controller allows to limit the usage of large pages per control group and enforces the controller limit during page fault.
- **pids** - Process number controller is used to allow a cgroup hierarchy to stop any new tasks from being fork()'d or clone()'d after a certain limit is reached. (i.e. limits the maximum number of processes allowed in the cgroup)

2.3 Alternative Technologies - Linux Containers

Linux containers are operating system level virtualisation solution that utilise control groups (cgroups) and namespaces provided by the Linux kernel. This technology is implemented in user-space and requires at least Linux Kernel 3.8 although certain features depend on more recent versions.

A container instance consist of a set of processes to which appropriate namespaces are being applied to achieve isolation from the host OS and other containers. Resource limitations are realised by means of cgroups.

There has been many implementations that make use of kernel features to implement container-based isolation. The most popular projects are Docker [1], LXC/LXD [2], OpenVZ [3], Libvirt-LXC [4], Linux-VServer [5].

2.3.1 Docker

Docker is an open source software that allows automating deployment of applications inside Linux containers. It provides means to package an application with all of its dependencies into container image. It also allows you to run a container, configure network interfaces and firewall rules associated with containers, attach volumes and create/restore snapshots. Docker has been designed with the idea of using remote registries to obtain container images. The most popular public registry is Docker Hub [6].

[1] `https://docker.com/`
[2] `https://linuxcontainers.org/`
[3] `https://openvz.org/`
[4] `https://libvirt.org/drvlxc.html`
[5] `http://linux-vserver.org`
[6] `https://hub.docker.com/`

Docker is implemented in the Go Programming Language and consist of many components. In April 2017 [7], Docker (the company) announced the decision to differentiate Docker (the commercial software products Docker CE and Docker EE) from Docker (the open source project). The reason for this was to simplify the process of combining components into something usable. This change made it easy for the developer teams to collaborate not only on the software components, but also on assemblies of components. This is similar to the car industry where assemblies of components are reused to build completely different cars. With this change was introduced the *Moby* project [8]. Moby is comprised of a library backend components (e.g., a low-level builder, logging facility, volume management, networking, image management, etc.), a framework for assembling the components into a standalone container platform, and tooling to build, test and deploy artifacts for these assemblies. It also includes a reference assembly, called *Moby Origin*, which is the open base for the Docker container platform.

2.3.2 Docker Images and Layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile [51] that is used to build a container. Each layer except the very last one is used as read-only (See Figure 2.2). For example the following Dockerfile:

```
FROM centos:7
COPY . /app
RUN make /app
CMD python /app/app.py
```

This Dockerfile contains four commands, each of which creates a separate layer. The `FROM` statement starts out by creating a layer from the `centos:7` image. The `COPY` command adds files from the current working directory. The `RUN` command builds your application using the `make` command. Finally, the last layer specifies what command to run when the container is started.

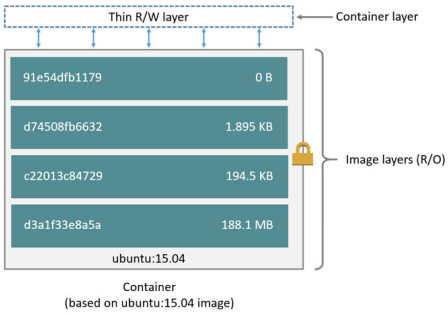


Figure 2.2: Layers of container image [51]

Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, a new writable layer is added on top of the underlying layers (See Figure 2.2). This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

The major difference between a container and an image is the top writable layer (See Figure 2.3). All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. Because each container has its own writable container layer, and all changes are

[7] `https://blog.docker.com/2017/04/introducing-the-moby-project/`
[8] `https://mobyproject.org/`

stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state.

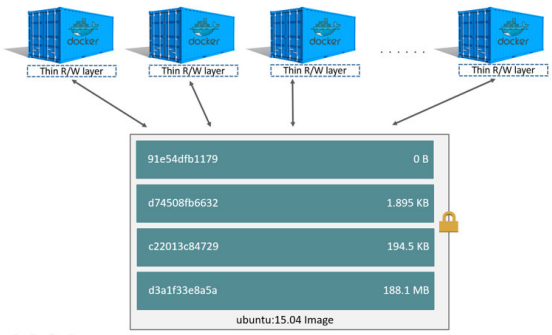


Figure 2.3: Sharing layers of container images [51]

The copy-on-write strategy is used to share and copy files with maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers.

2.3.3 LXC/LXD

LXC is an open-source software that provides operating system level virtualisation by utilising kernel namespaces, cgroups and other Kernel features. The main focus of the LXC project are system containers. These are containers which offer an environment as close as possible as what a VM will provide but without the overhead that comes with running a separate kernel and simulating all the hardware.

LXC supports unprivileged containers. These are containers are containers that have enabled user namespace and applied set of UID/GID mappings to all running processes. In essence, user namespaces isolation is achieved by establishing a mapping between a range of UIDs and GIDs on the host to a different (unprivileged) range of UIDs and GIDs in the container. The kernel will translate this mapping in a way that inside the container all UIDs and GIDs appear as you would expect from the host whereas on the host these UIDs and GIDs are in fact unprivileged. For example, a process running as UID/GID 0 inside a container might appear as UID/GID 1000 on the host.

LXC is implemented in C and provides a library as well as API bindings for Ruby [9], Python [10], Lua [11] and Go [12].

LXD is being built on top of LXC to provide better user experience. Under the hood, LXD uses LXC through the C library `liblxc` and its Go bindings to create and manage the Linux containers. The core of LXD is a privileged daemon that exposes a REST API over a local unix socket as well as over the network (if enabled).

2.3.4 OpenVZ

OpenVZ is an open-source software for container-based virtualisation and server automation. OpenVZ utilises a patched version of the Linux kernel currently based on version 2.6.32 [13]. The next major release is prepared to upgrade to version 3.10.

[9] `https://github.com/lxc/ruby-lxc`
[10] `https://github.com/lxc/python3-lxc`
[11] `https://github.com/lxc/lua-lxc`
[12] `https://github.com/lxc/go-lxc`
[13] `https://openvz.org/Download/kernel`

OpenVZ creates multiple secure, isolated containers (otherwise known as VEs or VPSs) on a single physical server. This enables better server utilization and ensures that applications run in independent, isolated environments.

OpenVZ was initially developed by the company SWsoft, which was later renamed to Parallels, and nowadays called Virtuozzo. OpenVZ is free and open implementation of Virtuozzo containers for Linux and currently is used by more then 100 hosting providers [14] around the world.

2.4 Domain Management with Libvirt

Libvirt is an open source domain-management library written in C with main goal to provide a common and stable layer sufficient to securely manage domains on a node, possibly remote. Originally it was a wrapper around Xen, now it supports a variety of virtualisation solutions such as OpenVZ, QEMU, VirtualBox, VMware ESX, VMware, Microsoft Hyper-V, IBM PowerVM (phyp), Virtuozzo, Bhyve and LXC - Linux Containers. Furthermore, it provides a long-term stable API and a unified domain configuration via XML files. This guarantees that clients interacting with Libvirt will not be affected by changes (e.g. upgrade) of the virtualisation layer's API.

Internally, libvirt is divided into two layers – domain-dependent and domain-independent (See Figure 3.1). The former is implemented by means of Libvirt *drivers* for each virtualisation solution (hypervisor). The domain-independent layer separates these drivers from user interfaces. On the one hand, this eases the driver development as common functionality is available on the domain-independent layer. On the other hand, users are provided with a consistent interface for different virtualisation solutions. In addition to the domain drivers, libvirt provides the special *remote* driver. This driver enables remote management by redirecting calls to instances of the *libvirtd* daemon that are running on remote server.

2.5 Checkpoint and Restore

Application checkpoint-restore is the ability to save the state of a running application to secondary storage so that it can later resume its execution from the time at which it was checkpointed.

Checkpoint-restart can provide many potential benefits [56], including fault recovery by rolling back an application to a previous checkpoint, better application response time by restarting applications from checkpoints instead of from scratch, and improved system utilization by stopping long running computationally intensive jobs during execution and restarting them when load decreases.

An application can be migrated by checkpointing it on one machine and restarting it on another providing further benefits, including fault resilience by migrating applications off of faulty hosts, dynamic load balancing by migrating applications to less loaded hosts, and improved service availability and administration by migrating applications before host maintenance so that applications can continue to run with minimal downtime.

Many important applications consist of multiple cooperating processes. To checkpoint-restart such applications, not only must application state associated with each process be saved and restored, but the state saved and restored must be globally consistent and preserve process dependencies. Operating system process state including shared resources and various identifiers that

[14] `https://openvz.org/Hosting_providers`

define process relationships such as group and session identifiers must be saved and restored correctly. Furthermore, for checkpoint-restart to be useful in practice, it is crucial that it transparently support the large existing installed base of applications running on commodity operating systems.

Transparent checkpoint-restart of unmodified applications may be composed of multiple processes on commodity operating systems. The key idea introduced in [81] is to provide a thin virtualisation layer on top of the operating system that encapsulates a group of processes in a virtualised execution environment and decouples them from the operating system. This layer presents a host-independent virtualised view of the system so that applications can make full use of operating system services and still be checkpointed then restarted at a later time on the same machine or a different one. While this design does not require patching of the kernel, it has a number of important engineering issues in building a robust checkpoint-restart system. In particular, a key issue is how to transparently checkpoint multiple processes such that they can be restarted in a globally consistent state.

This consistency issue is addressed in [56] by combining a kernel-level checkpoint mechanism with a hybrid user-level and kernel-level restart mechanism to leverage existing operating system interfaces and functionality for checkpoint-restart. This mechanism accounts for process relationships that are correctly saved and restored in a globally consistent manner and is available for commodity operating systems. This algorithm is crucial for enabling transparent checkpoint-restart of interactive graphical applications and corrects job control.

Checkpoint/restore originated in High Performance Computing [7]. It was particularly valuable in HPC environments where a single application might be distributed to hundreds or thousand of cores. In HPC the failure of a single component can lead to data loss, and then the CPU cycles of those hundreds or thousand of cores are wasted.

Different approaches are used to avoid data loss caused by failures with checkpoint-restore.

- The application can checkpoint/restore itself to store its current state.
- The application can be checkpointed/restored in a “semi-transparent” mode by intercepting system calls.
- The application can be checkpointed/restored in fully transparent mode at the operating system level.

The advantage of a fully transparent operating-system-level checkpoint/restore is that you do not have prerequisites before you can checkpoint and restore. There is no requirement for special libraries to be linked against the application or specially prepared environments to intercept system calls. However, this approach requires a more complex tool to checkpoint and restore.

Many different checkpoint/restore implementations are available, but for many years they have not been easily available in most Linux distributions. Most Linux implementations have been too limited in their functionality or only useful to a limited audience.

With the rising interest in Linux container technology, checkpoint/restore has begun attracting more attention. Checkpoint and restore of a process can be used as a means of fault tolerance or dynamic load balancing by migrating a running process from one system to another.

Migrating a running process is nothing more than checkpointing a process, transferring it to

the destination system, and restoring the process to its original state. Checkpoint/restore technology can restore a whole process group. As a result, checkpoint/restore could become the perfect base technology for container migration.

2.6 Basics and Architecture of CRIU

Early implementations of checkpoint/restore did not focus on upstream inclusion. As a result, there was no agreement in the Linux kernel community on the design. This led to the adoption of solutions that were not officially accepted by the Linux community.

An in-kernel checkpoint/restore implementation was developed in cooperation with the Linux community. The in-kernel checkpoint/restore approach was getting too complex to be integrated into the Linux kernel and was therefore not further developed and abandoned.

To solve the problems of these earlier implementations, CRIU takes another approach. It implements as much functionality as possible in the user space and uses existing interfaces to implement checkpoint/restore successfully.

2.6.1 Pre-copy migration with CRIU

One of the most important kernel interfaces for CRIU is the *ptrace* interface. CRIU relies on being able to seize the process via *ptrace*. Then, it injects parasite code to dump the memory pages of the process into image files from within the process's address space.

For each checkpointed part of the process, separate image files are created. Information about memory pages, for example, is collected from `/proc/PID/smaps`, `/proc/PID/mapfiles/` and from `/proc/PID/pagemap`.

The memory pages image files require the most storage space, especially compared to the remaining image files. The remaining image files contain additional information about the checkpointed process, such as opened files, credentials, registers, task state, and so on. To checkpoint a process tree (a process and all its child processes), CRIU checkpoints each connected child process.

2.6.2 Post-copy migration with CRIU

Post-copy migration minimises the application downtime and has the advantage of sending each memory page overall network at most once. This results in reduced overall bandwidth used for application migration [108]. The userfault technology recently introduced to the Linux kernel allows post-copy migration of virtual machines. However, this technology is missing essential features required for post-copy migration of Linux containers.

To restore a process, CRIU uses the information gathered during checkpointing. A process can be restored only if it has the same PID (process ID) it had when it was originally checkpointed. If another process is using this PID, the restore will fail.

One of the reasons that the process must be restored with the same PID is that parent-child process trees have to be restored exactly as they were. It is not possible to re-parent a process. To restore a process with the same PID, a newly introduced kernel interface is used to influence which PID the kernel gives to the next process.

If the process just created with `clone()` has the correct PID, CRIU transforms it into the same state the process was in before being checkpointed. Files are opened and positioned as they were before, memory is restored to the same state, and all other remaining information from the

image files is used to restore the process. Once the state is restored, the remaining parts of the restorer are removed. Then, the restored process resumes control and continues from the point at which it was previously checkpointed.

One general limitation is that CRIU can only checkpoint and restore processes using inter-process communication (IPC) if the processes are running inside of an IPC-namespace.

Existing parent-child relations in process trees must be kept intact. This means that CRIU always checkpoints and restores a parent process and all its child processes. It is not possible to checkpoint and restart a parent process on its own.

This limitation is related to the requirement that the PID must stay the same. A CRIU restore process fails if the intended PID is already in use.

For a successful migration the used libraries must be the exact same version on both the source and destination systems. This limitation exists because the process already has all required libraries loaded and expects that the functions provided by those libraries are at the same address as they were during start up.

One of the main use cases of CRIU is to migrate a Linux container. Depending on the applied container technology, checkpointing and restoring with the help of CRIU might already be included.

Previous work on live migration of Linux Containers has been implemented in a way to store duped state of containers on disk before it is send over to the destination node. Disadvantage of this approach is that all images files are written to disk twice (once when dumping the container on the source host, and once when receiving the images on the destination host), and being read from disk twice (once when sending the images to destination host, and once when performing the restore operation).

One proposed solution is *Disk-less migration*, which simply mounts a temporary file system on the target path where image files will be stored [32]. This guarantees that all files will be stored in memory and does not require modification of the CRIU tool.

On February 2017, a patch series was merged into the development branch of CRIU that decouples the process of saving/reading image files from dumping/restoring process tree. The communication between these two components is achieved via Unix sockets. This simplifies the process of live migration between two nodes by combing the dumped memory pages with the rest of the process state. In addition, it is more efficient because it keeps all images in memory buffer which is directly send over to the destination node. Live migration is achieved by first creating, on the source node, a *cache* socket with the `criu --image-cache` command that takes an argument `--port` and listens for incoming connections. Then, on the destination node, start a proxy socket that is used to forward data from the local machine to the remote host. Such socket is created with the `criu --image-proxy` command that takes arguments `--address` and `--port` for the destination host. The only change necessary to the dump/restore command of CRIU is to add the `--remote` argument.

Table 2.1: Commits introducing image/cache proxy

Commit Hash	Description

c45eb121e5ce3	Prepare for remote snapshots by adding the O_FORCE_LOCAL flag to force images not to stay as local files in the file system.
91b689a3a4695	Introduce the read_into_buffer helper that is required for page-cache and page-proxy set.
9469f2c022e5d	Prepare socket packets into buffer. In order to use sockets for image files transfer `lseek` calls needs to be removed and the packet data is read at the same time. This has the disadvantage to consume more memory between the image read and the packets restore because images are kept in memory.
86b9170ff3cd5	Read memory pages by chunks for the case when we get data from image-cache/proxy socket.
5afbbe5ce2b0e	Splice data into images by chunks in case when the target descriptor is cache/proxy socket.
2fb84926465a9	This patch introduces the `--remote` option and the necessary code changes to support it. This leaves user the option to decide if the checkpoint data is to be stored on disk or sent through the network (through the image-proxy). The latter forwards the data to the destination node where image-cache receives it.
0848223500bc3	The current patch brings the implementation of the image proxy and image cache. These components are necessary to perform in-memory live migration of processes using CRIU. The image proxy receives images from CRIU dump/pre-dump (through UNIX sockets) and forwards them to the image cache (through a TCP socket). The image cache caches image in memory and sends them to CRIU restore (through UNIX sockets) when requested.
81ae3efbf2fd6	When the input/outut files are cache/proxy sockets do not call `sendfile`.
15ee55f404018	Tests for image-proxy and image-cache.

2.7 Special File Systems

2.7.1 Temporary file system (tmpfs)

Temporary file system (tmpfs) is a memory that resides in RAM and/or swap, depending on current memory usage [95]. Creating a mount point as tmpfs enables directory file structure to be stored and accessed with higher speed. It is important to note that tmpfs is not persistent storage (i.e. the data will not be preserved on reboot). This file system type is extremely beneficial for applications with intensive I/O operations that do not need to be kept on secondary storage (e.g. caching). This file system is used to achieve disk-less [15] live migration with CRIU.

2.7.2 Proc file system (procfs)

The proc file system (procfs) is a special file system in Unix-like operating systems that presents information about processes in a hierarchical file-like structure, providing a more convenient and standardised method for dynamically accessing process data that held in the kernel than traditional tracing methods or direct access to kernel memory [102].

[15] `https://criu.org/Disk-less_migration`

Commonly this file system is mapped to a mount point named `/proc` by the operating system at boot time. The proc file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime.

The Linux kernel assigns each process a symbolic link per namespace kind in `/proc/PID/ns/`. The inode number pointed to by this symlink is the same for each process in this namespace. This uniquely identifies each namespace by the inode number pointed to by one of its symlinks. Namespaces can be referenced by a process belonging to that namespace, or using an open file-descriptor to the namespace's file `/proc/PID/ns/<ns-kind>`, or by creating a bind mount of the namespace's file `/proc/PID/ns/<ns-kind>`.

2.7.3 Overlay File system (overlayfs)

The overlay file system functionality in Linux (sometimes referred to as union file system) enables one file system to be mounted on top of the other [14]. The result will inevitably fail to look exactly like a normal file system for various technical reasons. In many cases, an object (a file) accessed in the union will be indistinguishable from accessing the corresponding object from the original file system.

An overlayfs combines two file systems - an "upper" file system and a "lower" file system. When a name (file) exists in both file systems, the object in the 'upper' file system is visible while the object in the "lower" file system is either hidden or, in the case of directories, merged with the "upper" object (folder). The lower file system can be any type of file system supported by Linux and does not need to be writable. The lower file system can even be another overlayfs. Only the lists of names from directories are merged. Other content such as metadata and extended attributes are reported for the upper directory only. These attributes of the lower directory are hidden.

The overlay file system is very important for Docker and OCI compliant containers. Such containers are based on images that consist of one or more layers. All layers are mounted in the right order using the overlay file system to merge all file changes stored in the layers of a container image.

Chapter 3

Design

This chapter includes appropriately analysed and defined requirements of the project, justification for design decisions and explores alternative checkpoint-restore solutions in comparison with CRIU. This chapter also includes explanation of the design decisions made to extent the architecture of Libvirt with support for save, restore and live migration of the LXC driver.

3.1 Requirements

This section establishes the functional and non-functional requirements of the project.

3.1.1 Scenarios

The scenarios below are used to frame the functionality introduced in the Libvirt-LXC driver. They provide basis for the functional and non-functional requirements, as well as the experiments carried out to evaluate the implementation.

1. Live migrating a memory intensive process running inside a container that does not interact with the host OS or other containers.

2. Live migrating a running container with set limits for the amount of CPU and Memory available to that container.

3. Live migrating a container that has a running process with large amount of cached data kept in memory.

4. Live migrating a server with open TCP connection.

3.1.2 Functional Requirements

While the checkpoint-restore tool is the main component of the system, additional checks for hardware comparability of source and destination nodes are required to be able to prevent failure in during live migration. The following requirements are for both the fault detection and functionalities used to achieve live migration.

(FR1) The user must be able to establish connection between local and remote libvirt daemon instance using the LXC driver.

In order to achieve live migration a user first have to be able to access remote host over the network, to which a container instance will be migrated.

(FR2) The user must be able to define, undefine, and edit the configuration of remote container instances via the *virsh* command line utility.

Managing domains in Libvirt could be done via an API interface used by a client program. The *virsh* utility is provided with Libvirt and implements a command-line tool for managing local and remote Libvirt instances.

(FR3) The user must be able to start and shutdown a local and remote instance of a Linux container using the Libvirt-LXC driver.

This is an essential requirement to control the life cycle of container instance on local and remote hosts. This functionality is very important to be able to successfully manage large scale cloud deployments of container instances with Libvirt.

(FR4) The user must be able to save the running state of container instance as a set of files on disk with or without interrupting the state of container.

The user must be able to easily backup a running instance of LXC container by saving its running state on persistent storage. This is an essential feature of every enterprise level hypervisor that allows to effectively keep copy of the state of domain at given time.

(FR5) The user must be able to effectively restore the running state of LXC domain from saved backup.

This requirement is essential for both the fault tolerance of domains based on the Libvirt-LXC driver and the live migration feature. Effective container restore from snapshot is necessary feature to achieve domain migration.

3.1.3 Non-functional Requirements

Non-functional requirements reflect on the expectation the usability and scalability of the system.

(NFR1) The live migration of container must be able to finish in finite amount of time despite of the intensity and type of the processes running inside a container.

While there are cases when memory intensive process running inside a container might have significant impact on the live migration process (e.g. in case of pre-copy migration), the migration must finish in a finite amount of time.

(NFR2) Domain configurations must be preserved on destination host after live migration.

This is an essential requirement to achieve effective domain migration with Libvirt. Each domain has associated configuration stored in XML format. This configuration must be made available to the destination host applied on restore.

(NFR3) Live migration must work with recent versions of the Linux kernel and compatible with the latest version of CRIU.

Checkpoint and restore functionalities utilise many system calls that are available only in recent versions of the Linux kernel. It is important to ensure that live migration also works with these kernel versions.

3.2 Libvirt

This section describes the internal architecture of Libvirt and Libvirt-LXC, and important aspects of Linux and Linux processes that needs to be taken under consideration when creating or managing containers.

3.2.1 Architecture & internals

Libvirt is collection of software mostly written in C with main goal to manage virtualisation capabilities of recent versions of Linux and other OSes. It provides an abstraction layer for various hypervisor management APIs and generally virtualisation technologies like virtual networking and storage. Libvirt is intended to be a building block for higher level management tools and for applications focusing on virtualisation of a single node (the only exception being domain migration between node capabilities which involves more than one node). The libvirt project have three main components - an API library, a daemon (*libvirtd*), and a command-line interface (*virsh*). [59] The API library supports C and C++ directly, and has bindings available for other languages such as C#, GO, Java, OCaml, Perl, PHP, Python, Ruby, D-Bus.

Libvirt goal [60] is to provide all APIs needed for various management operations to be performed on the guest OSes (e.g. provision, create, modify, monitor, control, migrate and stop the domains) within the limits of the hypervisor for those operations. Multiple nodes may be accessed with libvirt simultaneously, but the APIs are limited to single node operations. Libvirt also offers node resource operations needed for management and provisioning of guests such as network interface setup, firewall rules, storage management, and general provisioning APIs.

The code-base of libvirt is conceptually divided into several hypervisor specific parts also called *drivers*, and common utilities. (See figure 3.1)

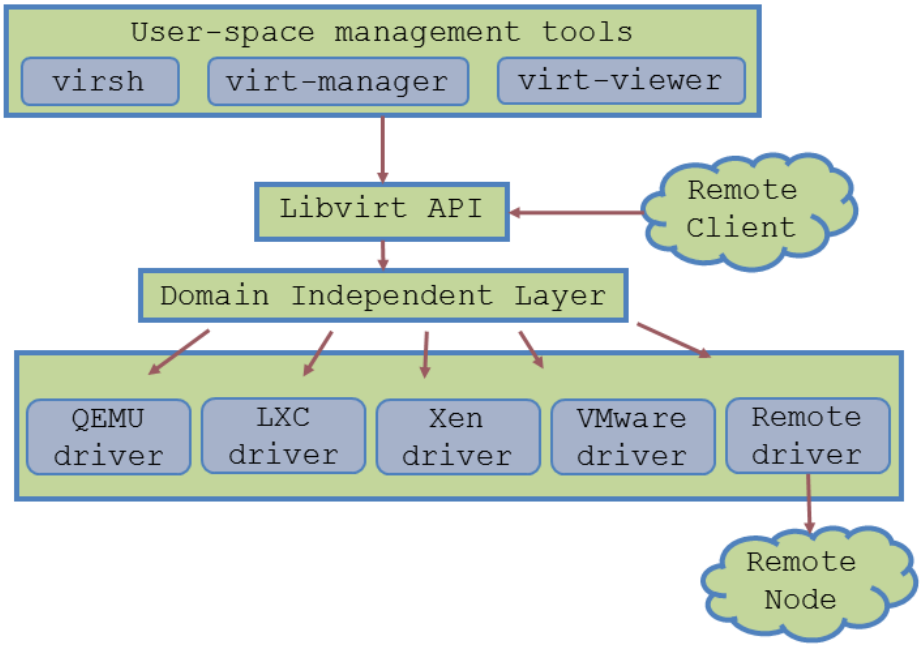


Figure 3.1: Overview of libvirt

Many hypervisors (e.g. KVM, QEMU) do not offer remote management facilities. Libvirt attempts to circumvent this by implementing a remote driver and a daemon handling remote requests. Client requests are tunnelled through the remote driver to a destination server, where the specific hypervisor is running. The daemon on the server node receives requested commands and locally calls the corresponding driver. (See figure 3.2)

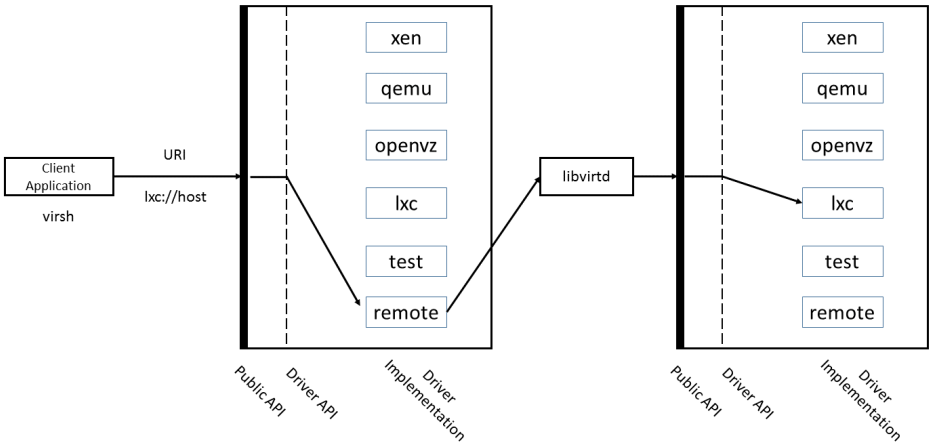


Figure 3.2: Remote driver in libvirt

Introducing a middle-ware architecture consisting of the remote driver and the libvirtd daemon offers great flexibility. Communication between client and hypervisor can be, for example, compressed and/or encrypted in a way that is totally transparent to the hypervisor. However, a drawback of this architecture is that the physical server, on which the remotely managed hypervisor resides, has to run the libvirt daemon.

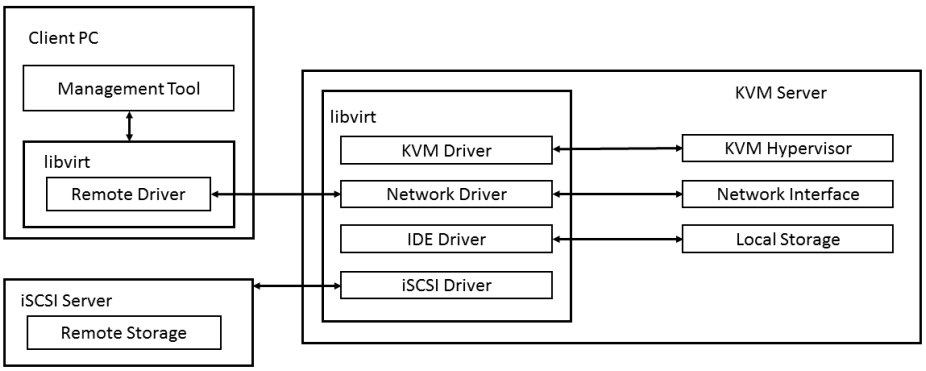


Figure 3.3: Driver based libvirt architecture

Each driver is responsible for loading specific functions, in order to be used by the public API of libvirt. For the majority of supported hypervisors, access to libvirt drivers is performed as follows. First the client application connects to a so called *remote driver*, providing a URI as defined in RFC 2396 [79]. Then the remote driver forwards the commands to a libvirt daemon via RPC calls. The libvirt daemon in turn polls the requested drivers asking them to perform the requested operation. When libvirtd gets a reply, it passes the results back to the client application which in turn decides what to do with them. (See figure 3.3)

Remote URIs in libvirt offer a rich syntax and many features [63]. The have a general form shown below, where `[...]` is an optional part, and either the `transport` or the `hostname` must be provided in order to distinguish this from a local URI. [61]

```
driver[+transport]://[username@][hostname][:port]/[path][?parameters]
```

3.2.2 Libvirt-LXC Driver

In terms of technical implementation, every Linux Container created with libvirt toolkit has a controller process whose binary is provided in the container's XML configuration file and is usually in form of `/usr/libexec/libvirt_lxc`. The main objective of libvirt-lxc is to setup a container by mounting and pivot_root()-ing the corresponding root file system, clone()-ing the initialisation process, set appropriate control groups, configure network interfaces and provide console access to the container.

The libvirt-lxc driver calls the *clone()* system call in order to create a new isolated process, from which a container will be generated. Namespace isolation is achieved via suitable flags passed to that system call. The only default flags used are CLONE_NEWPID and CLONE_NEWNS. If idmap [62] has been set the CLONE_NEWUSER is appended as well. In addition, a container can be configured to not inherit the Net namespace (CLONE_NEWNET), IPC namespace (CLONE_NEWIPC) and UTS namespace (CLONE_NEWUTS).

For the container to have stdin, stdout and stderr, libvirt-lxc utilises pseudoterminal interfaces (PTYs). A PTY is a pair of virtual character devices that provide a bidirectional communication channel. One end of the channel is called the master; the other end is called the slave. The slave end of pseudoterminal provides an interface that behaves exactly like a normal unix terminal interface (TTY). Inside the container is supplied the slave end of a PTY, whilst the container's controller process (libvirt-lxc) is supplied with the master end. This enables interaction with the container via a virtual console. For Linux Containers, the slave ends live inside the container in form of `/dev/pts/*` files and the master end exists in the host OS in `/dev/ptmx` device. (See figure 3.4)

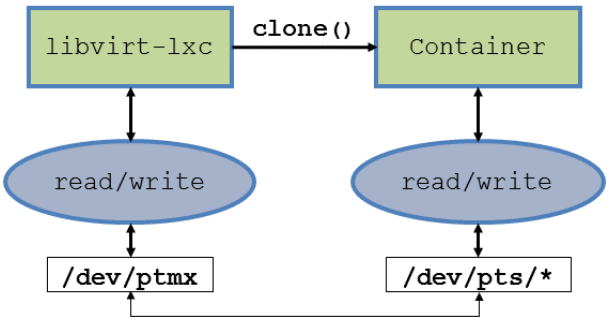


Figure 3.4: Pseudoterminal device between libvirt-lxc and container

Once the container setup is complete, libvirt-lxc executes container's init process, specified in the XML configuration file. Then libvirt-lxc becomes a parent process of the container, acting as a controller or monitor. It will receive a SIGCHLD signal when the last process in the container terminates. On the other hand, if the controller process is killed, the container dies a well. The way the usage of the host's resources is restricted in the container's environment is cgroups. Among others, with cgroups the container has limited memory usage, CPU usage and Block IO, all of which are configurable through container's XML. Lastly, in order that the container gets different root file system than the process who has cloned him, *pivot_root* system call [86] is used.

3.3 Container Monitor

An instance of a process, often called *container monitor*, is associated with each container. Such process remains in the host namespaces and from a kernel perspective it becomes a parent of the process tree inside the container, however it is invisible from container perspective. A container monitor is responsible for monitoring the state of container and initial setup - it prepares root file system, creates network interfaces, setup environment variables, file descriptors, pseudo-terminals, SELinux/AppArmor contexts, and spawn the initial process tree inside a container.

In LXC project this process is called *lxc-monitor* [67], in Libvirt this is the *libvirt-lxc* process, implemented in `src/lxc/lxc_container.c`.

3.4 Pivot root

As part of the root file system preparation the `pivot_root()` system call [86] is used to change the root file system. This syscall moves the rootfs of the calling process (i.e. usually the container monitor) to a directory underneath, that is, adding a non-zero number of `/..` to the path of the old root must yield the same directory as new root. Calling the `pivot_root()` may or may not affect its current working directory. It is therefore recommended to call `chroot("/")` immediately after `pivot_root()`.

3.5 Process Creation

In Linux, a process is an instance of a running program. Every process is created when another process performs a `fork()` [43] or a `clone()` [23] system call [100]. The only exception is the very first process that the kernel creates after system boot (which has PID=0).

The process that invoked the system call is called *parent* process and the newly created process is the *child* process.

This is important to understand how Linux container are created, monitored and managed.

3.5.1 Zombie Processes

The Linux Kernel maintains a table that associates every process, by means of its process identifier (PID) to the data necessary for its functioning, such as memory allocation, arguments, environment variables, resource usage counters, user-id, group-id, etc. When a process terminates its execution, either by calling `exit()` or return from the main function, or by receiving a signal that causes it to terminate abruptly, the operating system releases most of the resources and information related to that process, but still keeps the data about resource utilization and the termination status code, because a parent process might be interested in knowing if that child executed successfully and the amount of system resources it consumed during its execution.

The Linux Kernel assumes that a parent process is always interested in such information, and therefore sends the parent a signal `SIGCHLD` to indicate the child's termination. A parent process collect child's information by calling a function of the `wait` family (e.g. `wait`, `waitpid`, `waitid`, `wait4`, etc.). Immediately after such collection is made, the system releases those last bits of information about the child process and removes its pid from the process table. However, if the parent process does not collect the child's data, the Kernel keeps the child's pid and termination data in the process table indefinitely. Such a terminated process whose data has not been collected is called a *zombie process*.

Zombie processes might pose problems on systems with limited resources or that have limited-size process tables, as the creation of new, active processes might be prevented by the lack of resources still used by long lasting zombies. It is, therefore, a good programming practice in any program that might spawn child processes to have code to prevent the formation of long lasting zombies from its original children. The most obvious approach is to have code that calls wait or one of its relatives somewhere after having created a new process. If the program is expected to create many child processes that may execute asynchronously and terminate in an unpredictable order, it is generally good to create a handler for the `SIGCHLD` signal, calling one of the wait-family function in a loop, until no uncollected child data remains. It is possible for the parent process to completely ignore the termination of its children and still not create zombies, but this requires the explicit definition of a handler for `SIGCHLD` through a call to `sigaction()` [97] with flag `SA_NOCLDWAIT`.

3.5.2 Orphan Processes

Orphan processes are the opposite of zombie processes. They refer to the case when a parent process terminates before it's child processes. Such child processes are called "orphaned". Unlike the parent process, child processes are not notified when their parent exits. Instead, the operating system redefines the "parent PID" field in the child process's data to be the process that has PID value of 1 (one), and whose name is traditionally "init". Thus, it is said that init "adopts" every orphan process on the system.

Since Linux 3.4 onwards was added the ability for processes to set themselves up as "subreapers", thus a processes can issue the `prctl()` system call [90] with the `PR_SET_CHILD_SUBREAPER` option. As a result the child processes of a terminating process will be adopted by this process's immediate parent process.

3.5.3 User and Group Identifiers

In Unix-like operating systems every process is associated with user and group identifiers (UID and GID) [48]. Containers make use of the user namespace implemented in the Linux kernel to have their own process identifier table, allowing unprivileged processes on the host to run with UID/GID 0 inside the container. Although, such processes seem to be running as superuser from container's perspective, from the kernel's point of view they remain unprivileged processes.

The user and group ID can further be classified into *real* and *effective*.

Real user and real group identifiers

These identifiers give information about the user and group to which a process belongs. A process inherits these identifiers from its parent process.

Effective user ID, effective group ID and supplementary group ID

These three IDs are used to determine the permission that a process has. Usually the effective UID is same as real UID but in case its different then it means that process is running with different privileges then what it has by default (i.e. inherited from its parent). If a process is running with effective user ID '0', this means that this process has special privileges. Such processes are known as *privileged processes* as they are running as superuser. These processes bypass all the permission checks that kernel has in place for all the unprivileged processes.

Chapter 4

Implementation

This section describes the integration of CRIU tool with the Libvirt-LXC driver to add support save/restore and live migration of Linux containers. This section also includes justification for all of the implementation and how the code has been made efficient, reusable and modular.

4.1 LXC Driver Architecture

The source code of `libvirt-lxc` driver is placed in the `src/lxc/` subdirectory. This driver shares the architecture of other Libvirt drivers, such as QEMU and XEN. The current implementation of the driver is composed of the following files.

Current Libvirt-LXC Driver Source Files	
`libvirtd_lxc.aug`	Augeas [1] support for configuration file. Augeas is a configuration editing tool. It parses configuration files in their native formats and transforms them into a tree.
`lxc_cgroup.{c,h}`	LXC cgroup helpers for reading and setting resource limits.
`lxc.conf`	Master configuration file for the LXC driver. All settings described here are optional - if omitted, sensible defaults are used.
`lxc_conf.{c,h}`	Configuration functions for managing Linux containers.
`lxc_container.{c,h}`	Responsible for performing container setup tasks.
`lxc_controller.c`	Linux container process controller.
`lxc_domain.{c,h}`	Helper function for performing domain related operations.
`lxc_driver.{c,h}`	Helpers for LXC driver functions.
`lxc_fuse.{c,h}`	Fuse file system support for Libvirt-LXC
`lxc_hostdev.{c,h}`	Functions for preparation and setup of host devices.
`lxc_monitor.{c,h}`	Client for LXC controller monitor.
`lxc_monitor_protocol.x`	Definition of protocol for communication between the LXC driver in libvirtd, and the LXC controller in the libvirt_lxc helper program.
`lxc_native.{c,h}`	Function for LXC native configuration import. They implement `domxml-from-native` functionality for the LXC driver that can be used to convert LXC containers to Libvirt-LXC.

`lxc_process.{c,h}`	Responsible for life-cycle management of `libvirt-lxc` process.
`Makefile.inc.am`	A programmer-defined file and is used by `automake` [3] and `autoconf` [2] to generate Makefile.
`test_libvirtd_lxc.aug.in`	Configuration for auto-generation of Augeas test cases. When a new configuration file parameters are added, the corresponding additions to the Augeas lenses will be automatically generated.

My contribution to the current implementation build on top of Katerina's work [103] where was introduced a proof-of-concept for *save* and *restore* functionalities. This implementation was extended to a working solution that allows for a container to be checkpointed, restored and migrated to remote host running Libvirt daemon instance.

Source Files Implementing Live-Migration for the Libvirt-LXC Driver	
`lxc_migration.{c,h}`	Helper functions for handling LXC migration.
`lxc_criu.{c,h}`	Helper functions for checkpoint/restore of Linux containers. Wrapper around the CRIU binary used for container live migration.
`lxc_driver.{c,h}`	Extended with functions that enable save, restore and migration support.

4.1.1 Integration of CRIU with Libvirt-LXC

An important implementation decision was weather to use the `libcriu` library [1] that provides C API for CRIU, to spawn the binary executable directly, or utilise directly the RPC [2] protocol which uses Google Protocol Buffers to encode its calls. The difference between these three approaches is the following. On first thought, Libvirt and CRIU are written in C and integrating `libcriu` library seems to be the most effective approach. To achieve this couple of patches [3] [4] were send to CRIU's mailing list and a few problems arose. The library does not contain all necessary features in order to achieve checkpoint/restore of Libvirt-LXC container. According to the API compliance [5] the following features are not available in the C library - `leave-stopped`, `restore-sibling`, `log-pid`, `page-server`, `address`, `port`, `pidfile`, `ms`, `feature`, `libdir`, `inherit-fd`, `skip-mnt`, `enable-fs`. From all these missing APIs, the most important ones are: `leave-stopped` - allows to terminate a container after it has been checkpointed; `restore-sibling` - allows the container's root task to be restored as a sibling. This

[1] `https://criu.org/C_API`
[2] `https://criu.org/RPC`
[3] `https://lists.openvz.org/pipermail/criu/2018-February/040352.html`
[4] `https://lists.openvz.org/pipermail/criu/2018-February/040421.html`
[5] `https://criu.org/API_compliance`

feature is necessary for the restored container to become child of the `libvirt-lxc` monitor process. Although this could also be achieved with the `exec-cmd` option that allows to attach the restored container to custom binary, this approach leads to unnecessary complications. This problem is one of the reasons avoid the RPC API and `libcriu`. Another important feature that is missing in the C library is `inherit-fd`, this feature allows to restore ttys associated with container on restore. Without this it is not possible to restore containers with associated ttys. This is very important, as OS containers by default have at least one tty associated with them, that allows direct access. Implementing effective live migration requires to be able to start a page server. This feature is missing from both the C library and the RPC protocol of CRIU. I send a patch [6]3 to the CRIU mailing list (on 7-th of February 2018), that is introducing the page-server features in RPC. However this was not merged yet, and no feedback from the community was provided. Another important feature that is missing, is the ability to use image-cache and image-proxy and the --remote option. I send a patch series [7]4 that extends both the RPC protocol and the C API with the ability to start image-cache and image-proxy, as well as to use the `remote` option with dump and restore. None of these patches was merged and no feedback from the community was received. This was the reason to reconsider the way CRIU was integrated with Libvirt-LXC and change the implementation from using `libcriu` library to use the binary directly. Another advantage of this is that users will not need to recompile libvirt when they want to upgrade the CRIU version.

4.1.2 Configuration Parameters

An important problem that arose during the integration of the CRIU binary was how to find the path where the CRIU binary is installed. In Katerina's work this was implemented by extending the automake/autoconf files for Libvirt to search for the CRIU binary within `/usr/sbin` and `/usr/local/sbin`. This approach has the advantage of finding the CRIU binary at compile time, and set a macro that will populate the path in the Libvirt's code. This approach however, might not be suitable for package maintainers. This will require them to have CRIU installed at compile time in the exact same path where the users will have it. This is becomes a problem, for example, when a user has installed CRIU in `/usr/local/sbin` and the package maintainer has configured Libvirt with `/usr/sbin` path. Moreover, if a Libvirt package has been configured with CRIU this will introduce a necessary dependency for all users, even for the ones that do not use Libvir-LXC.

RunC solves this problem by looking for `criu` in the `PATH` environment variable and allowing the user to specify a different path to binary using the `--criu` argument. Similar approach is used in the implementation of LXC allows to dynamically finding the CRIU binary at run time in the `PATH` environment variable.

My implementation replaces the initial automake/autoconf patch from Katerina with the `virFindFileInPath` function. This follows the same idea implemented in RunC and LXC to find the CRIU binary at runtime based on the PATH environment variable.

4.2 Container Checkpoint and Restore

The CRIU tool provides a functionality allowing to save the running state of a process into set of files. CRIU actually always operates on a process tree (i.e. one process and all of its child processes). It uses both existing and extended interfaces to the Linux kernel to collect information about the process [9]. This is possible by the `ptrace` system call [91], which allows CRIU to take

control over a process tree and pause it execution. Then in the paused process is injected code via `ptrace`. Such code is called *parasite-code* and it runs from within the process's address space. This process allows access to the memory content of the process and once all memory pages and all additional information have been collected (and possibly written to a disk) the process can either continue running or it can be aborted. Letting the process continue to run is something to expect in a fault tolerance scenario - to migrate a process to another system the process would be aborted.

To restore a process CRIU transforms the CRIU process doing the restore into the process to be restored. This is one of the places where CRIU uses a newly introduced Linux kernel interface. With CRIU, a process can only be restored with the same process identifier (PID) the process had during checkpoint. To influence which PID the restored process gets, CRIU writes the PID of the to be restored process minus one to `/proc/sys/kernel/ns_last_pid`. CRIU then verifies if the newly created process actually has the desired PID. If not – the process restoration is aborted. Other steps performed during restore include file descriptors, which are re-opened with the same identifier as in the original process and then the file descriptors are repositioned to the same location. All extracted (dumped) memory pages are loaded from the checkpoint directory to the currently being restored process and mapped to the same location as in the original process.

4.3 Container Migration

Migration refers to the process of moving a running application, a VM instance or container instance, between two servers. For the purpose of this thesis, we are going to deal only with container migration. However the basic ideas apply to application migration and VM migration as well. The trivial form of migration is known as cold (or offline) migration, which is performed as follows.

- Stop the container.
- Copy its file system to another server.
- Start container.

Cold migration has significant drawback that requires the container to be completely stopped before transfer. This condition might not be acceptable, if for example the running applications will lose their progress when terminated, or remote clients will lose connectivity. One solution to this issue is *live migration*. Live migration refers to the process of moving a running Linux container between different physical machines without disconnecting a remote clients or loosing the current state of applications running inside the container. Although, during live migration the running applications experience down-time to certain extend, the network connectivity between clients and the server will not be broken when the down-time is less than the time-out of the communication protocol. For example in the TCP protocol whenever a packet is sent, the sender sets a timer that is a conservative estimate of when that packet will be acknowledged. If the sender does not receive an acknowledge response by then, it transmits that packet again. The timer is reset every time the sender receives an acknowledgement. This means that even when an application serving client over the TCP protocol the client will not notice the that the server has been live migrated. In case of the UDP protocol, packet loss is acceptable and will be ignored by the clients.

Below are presented four implementations of live migration for Linux containers using CRIU tool. The differ in the way memory state is transferred between source and destination host, which affects the efficiency of the algorithm.

4.3.1 Native live migration

The basic form of live migration of containers consist of three steps - checkpoint a running container to set of image files; transfer the checkpoint to a remote host; restore the running state of the container. This was been implemented in the Libvirt-LXC driver by expanding the `lxc_container.c` source file with `lxcContainerMountFSDevPTSRestore` and `lxcContainerChildRestore` methods, and adding `restorefd` argument to `lxcContainerStart` function.

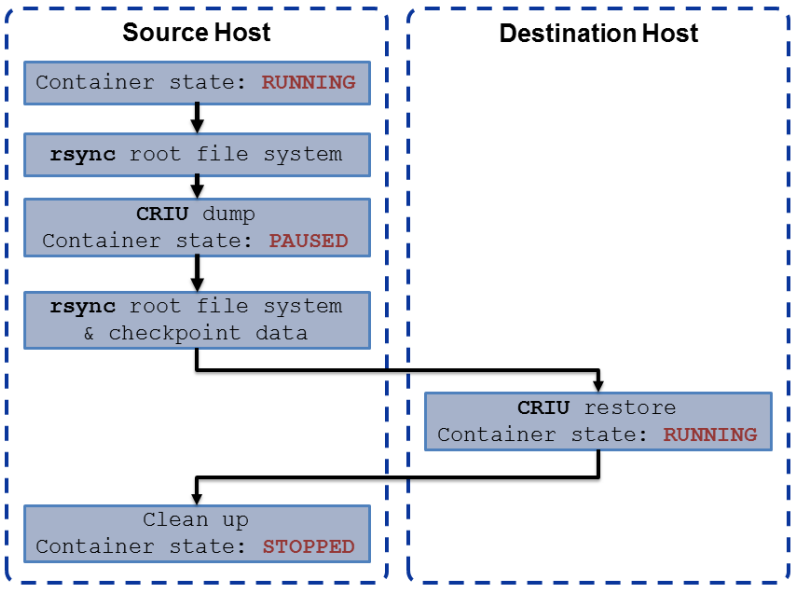


Figure 4.1: Native Container Migration

4.3.2 Pre-copy live migration

The pre-copy technique is used to decrease process downtime during live migration. The major advantage over simple native live migration is that in pre-copy technique the memory is copied from source to destination node in multiple iterations. During each iteration only memory pages that were changed since the last run are re-transferred to destination host. This technique avoids resending unmodified memory pages over the network when the migrated process is not running. This results in reduced down time for the migrated application.

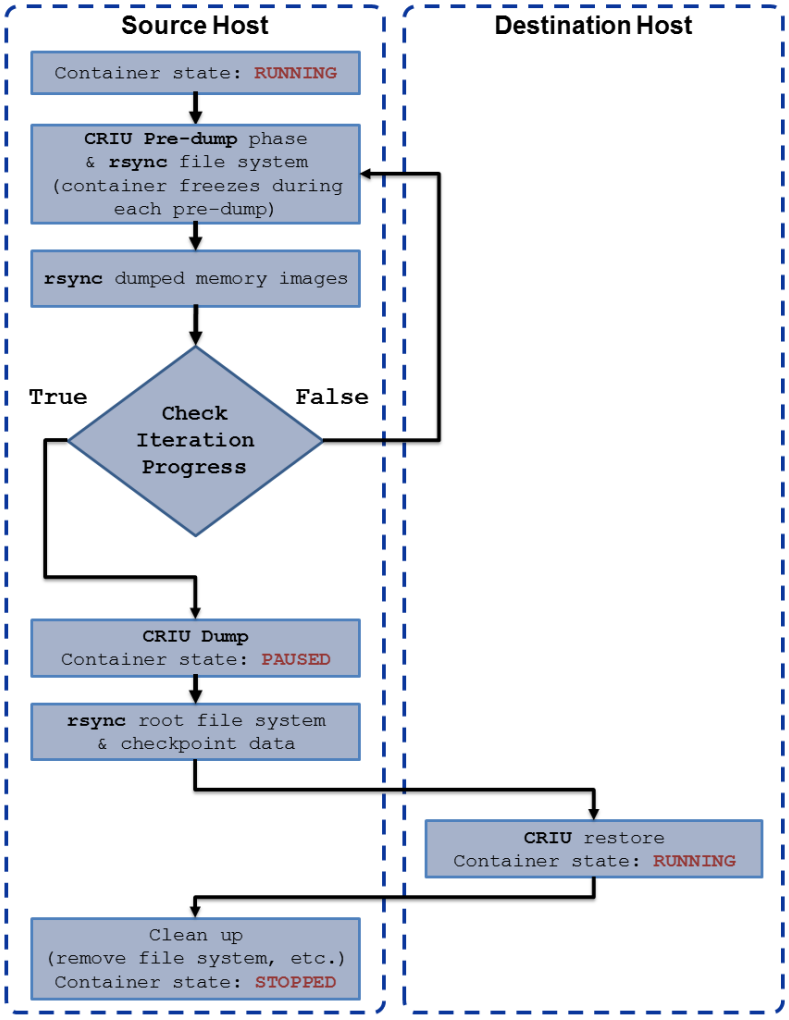


Figure 4.2: Pre-copy Container Migration

However, this approach is only effective when the dirty page rate is smaller than the network speed. CRIU supports incremental pre-dump phase by using the concept soft-dirty bit implemented as a kernel feature that allows memory changes tracking [41, 33, 66]. The soft-dirty is a bit on a Page Table Entry (PTE) that helps to track pages modified by a task. This *memory tracking* is utilised by writing 4 to `/proc/[pid]/clear_refs`. The kernel then will clear the soft-dirty bit for all pages associated with a process. This is used in conjunction with `/proc/[pid]/pagemap` by the check-point restore system to which pages of a process have been dirtied since the file `/proc/[pid]/clear_refs` was written to. The file `/proc/[pid]/pagemap` shows the mapping of each of the process's virtual pages into physical page frames or swap area. This process could be summarised in the following steps.

1. Clear soft-dirty bits from the task's PTEs by writing the number 4 into the `/proc/PID/clear_refs` file of the task.

2. Read soft-dirty bits from `/proc/PID/pagemap`. The soft-dirty bit of 64-bit qword is 55. If set, the respective PTE was written to since step 1.

Pre-copy optimisation for live migration of Libvirt-LXC containers is based on CRIU. There are a few important parameters used in the implementation.

The first one is the maximum number of pre-dump iterations. This value sets a hard limit of 8 pre-copy dumps. When reached the iteration loop breaks and the migration proceeds with final dump.

Second important parameter is the minimum count of dumped pages needed to continue iteration. This parameter is set with default value of 64 pages. In case when the number of pages is less, we proceed with final dump. The third parameter is the minimum count of transferred file system bytes needed to continue iteration. This parameter has default value of 0x100000 (1 MiB).

$$1 \text{ MiB} = 1024 * 1024 = 1048567 = 0x100000 \text{ bytes}$$

The fourth parameter used to control Pre-copy live migration is the maximum acceptable iteration grow rate percentage, with default value of 10%. This value is a threshold for the maximum increase of changes that needs to be transferred for current pre-dump with regards to the previous iteration. This threshold is used for both the root file system synchronisation and the transfer of memory pages. The algorithm calculates the grow rate G using the current and previous amount of transferred bytes T_{curr} and T_{prev} respectively.

$$\Delta := T_{curr} - T_{prev}$$

$$G := \Delta * \frac{100}{T_{prev}}$$

The fifth parameter is a threshold percentage of memory pages that needs to be pre-copied for the pre-copy migration to stop. This default value is set to 70%. The total amount of memory pages $F_T(x)$ is a function of the number of memory pages $P_W(x)$, that have been modified by the running process since the last reset of memory tracking, and the amount of pages that were not changes $P_S(x)$, and therefore needs to be skipped during pre-dump. Where x is the unique process identifier.

$$F_T(x) := P_W(x) + P_S(x)$$

The percentage of skipped memory pages during pre-dump operation $F_{PS}(x)$ is a function of the amount of modified memory pages $P_W(x)$ over the total amount of memory pages $F_T(x)$.

$$F_{PS}(x) := 100 - \frac{100 * P_W(x)}{F_T(x)}$$

The following algorithm is used to implement the logic for skipped memory pages.

After each pre-copy iteration is extracted information about the number of dumped memory pages by decoding the `stats-dump` file created by CRIU.

The above implementation of Pre-copy migration is based on the protocol used in P.Haul [8], as well as the Adrian's pull request [9] for LXD from 4-th of December 2017 and Katerina's migration scripts [10].

[8] `https://github.com/checkpoint-restore/p.haul`

[9] `https://github.com/lxc/lxd/pull/4072` - Add pre-copy migration support to LXD

[10] `https://github.com/KKoukiou/lxc-migration` - Scripts for LXC container Pre-copy migration using the CRIU tool

4.3.3 Post-copy live migration

The implementation of Post-copy migration is implemented by appending the arguments `--lazy-pages`, `--address`, `--port` and `--daemon` to the dump command. The address value passed to this command is `0.0.0.0` and the value for is available port number from the range live migration ports.

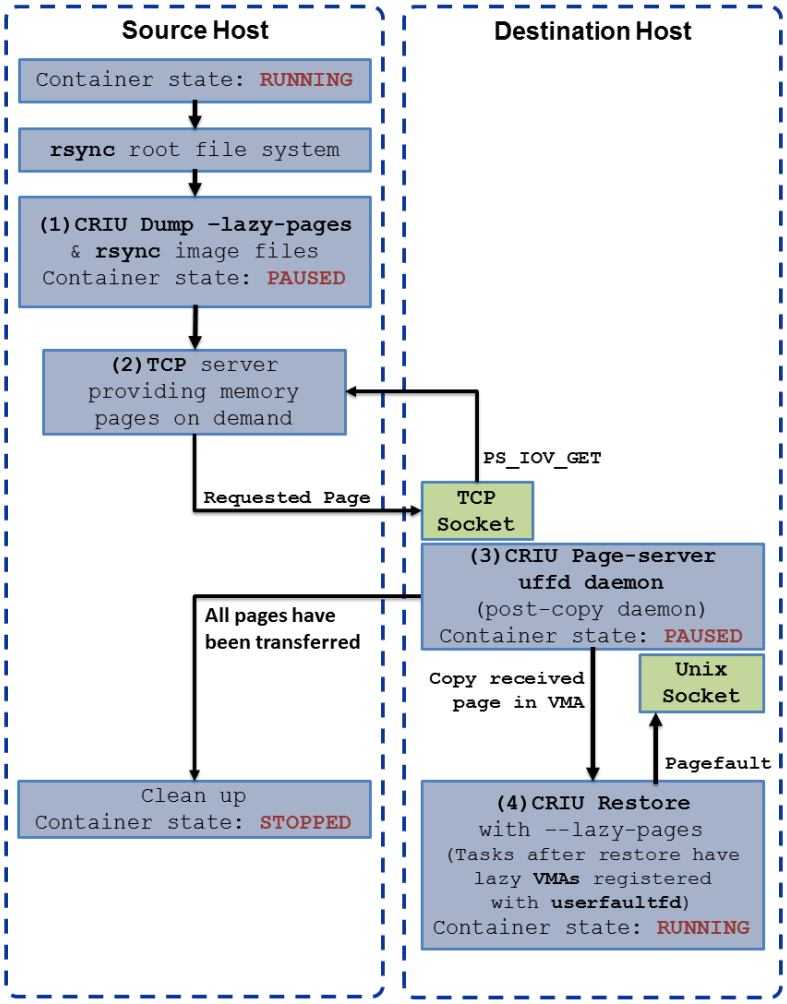


Figure 4.3: Post-copy Container Migration

On the destination host is started a daemon with `lazy-pages` option with arguments `--page-server`, `--address` and `--port`. The address value corresponds to the IP address of the remote (source) host, and the value for port matches the one used for checkpoint.

After the page-server daemon was started, Libvirt-LXC executes the restore command with additional `--lazy-pages` argument.

4.3.4 Image-cache and Image-proxy live migration

Minimising the down time during live migration is extremely important. The ultimate goal is to achieve down-time lower than a the TCP retransmission timeout [53]. This will ensure that clients will keep their connections open even during live migration.

One way to improve the efficiency is to avoid writing the process image files to secondary storage. Keeping all memory images in cache buffer, in memory allows the container state to be directly transferred to destination.

In [15] were proposed for first time two new auxiliary components to CRIU: Image Cache and Image Proxy. This former component runs on destination side and caches process snapshots in memory until the restore is finished. The latter runs at the source site and forwards process snapshots to the destination site.

As show in figure 4.4 the benefits from using such components are twofold [15]. First, all snapshots are kept in memory on both sides, which is much faster than writing and reading from disk (even for SSDs). Second, since the Image Proxy pro-actively forwards the snapshot to the Image Cache, restore process can start prior the dump has finished and the creation the snapshot and while the snapshot is still being transferred.

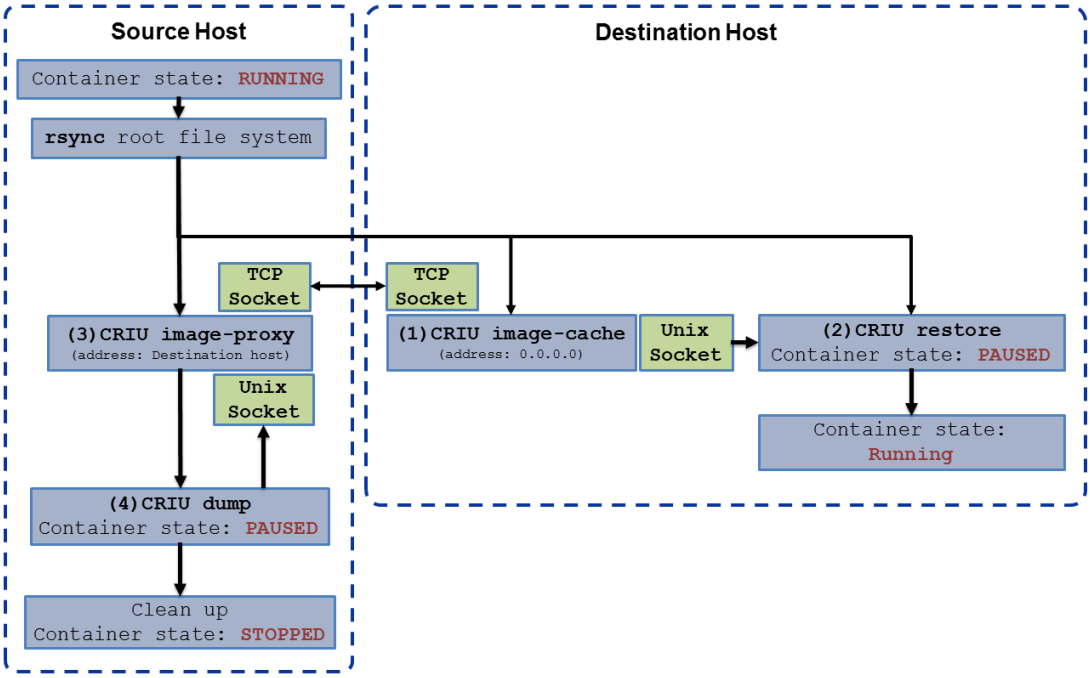


Figure 4.4: Post-copy Container Migration

The efficiency of this method has been demonstrated in [15] with comparison to other migration approaches including normal CRUI checkpoint/restore using NFS to transfer snapshots from source to destination site.

In chapter 5 we build on top of previous work and utilise this novel migration approach for live container migration. We also evaluate and compare live container migration using Pre-copy, Post-copy, only page-server with the Image-{cache,proxy} solution.

4.3.5 Migration Parameters

Summary of the parameters introduced with this implemented are described in Table 4.3.

Currently all parameters are used with their default values, and all constants are defined in the `migration.h` header file. In future work they will be exposed as command line and/or configuration options that can be specified by the user.

4.4 Regression Testing

In addition to extending the functionality of the Libvirt-LXC driver were introduced a set of regression tests by extending the current test suite of Libvirt. These set of tests will ensure that all

LXC_MIGRATION_PORT_MIN	Lowest port that can be used for Live migration between hosts.
LXC_MIGRATION_PORT_MAX	Highest port that can be used for Live migration between hosts.
LXC_MIGRATION_THRESHOLD	Percentage of skipped memory pages above which iterations of pre-copy migration will complete.
LXC_MIGRATION_MAX_ITERATIONS	Maximum numbers of pre-dump iterations during Pre-copy migration.
LXC_MIGRATION_MIN_PAGE_COUNT	Minimum count of dumped pages needed to continue iteration.
LXC_MIGRATION_MIN_FS_XFER_BYTES	Minimum number of transferred file system bytes needed to continue iteration.
LXC_MIGRATION_MAX_GROW_RATE	Maximum acceptable iteration grow rate.

Table 4.3: Introduced Libvirt-LXC Parameters

developed and tested code will continue to perform the same way after future modifications. The test suite can be launched with the `make check` command. It covers the following scenarios:

- Migrate an LXC domain with specified XML configuration and ensure that the received definition (configuration) matches the original.
- Migrate root file system and ensure that the new copy is identical to the original one by comparing a checksum for all files.

In future work this test suite will be extended with more comprehensive test cases used to address cases, such as a difference in the hardware configuration of the source and destination servers.

4.5 Programming Style

The libvirt project has well defined programming style [64] that has been followed in this work to ensure that the code is consistent with the rest of the code base. From all contributor guidelines there are several important that are described in this section.

The C source code in Libvirt generally adheres to some basic code-formatting conventions – use four spaces, not TABs for each indentation level.

Libvirt requires a C99 compiler for various reasons. However, most of the code base prefers to stick to C89 syntax unless there is a compelling reason otherwise. For example, it is preferable to use /* */ comments rather than //. Also, when declaring local variables, the prevailing style has been to declare them at the beginning of a scope, rather than immediately before use.

4.5.1 Naming conventions

The libvirt code has a number of different naming conventions that are evident due to various changes in thinking over the course of the project's lifetime. When working on existing files, while it is desirable to apply these conventions, keeping a consistent style with existing code in that particular file is generally more important. The overall guiding principal is that every *file*, *enum*, *struct*, *function*, *macro* and *typedef* name must have a "vir" or "VIR" prefix. All local

scope variable names are exempt, and global variables are exempt, unless exported in a header file.

File naming varies depending on the subdirectory. The preferred style is to have a "vir" prefix, followed by a name which matches the name of the functions/objects inside the file. For example, a file containing an object "virHashtable" is stored in files "virhashtable.c" and "virhashtable.h". Sometimes, methods which would otherwise be declared "static" need to be exported for use by a test suite. For this purpose a second header file should be added with a suffix of "priv" (e.g. "virhashtablepriv.h"). Use of underscores in file names is discouraged when using the "vir" prefix style. The "vir" prefix naming applies to `src/util`, `src/rpc` and `tests/` directories. Most other directories do not follow this convention.

All `enums` should have a "vir" prefix in their typedef name, and each following word should have its first letter in uppercase. The enum name should match the typedef name with a leading underscore. The `enum` member names should be in all uppercase, and use an underscore to separate each word.

All `structs` should have a "vir" prefix in their typedef name, and each following word should have its first letter in uppercase. The `struct` name should be the same as the `typedef` name with a leading underscore. A second `typedef` should be given for a pointer to the `struct` with a "Ptr" suffix.

All functions should have a "vir" prefix in their name, followed by one or more words with first letter of each word capitalized. Underscores should not be used in function names. If the function is operating on an object, then the function name prefix should match the object `typedef` name, otherwise it should match the file name. This is followed by a verb (action) name, and an optional subject name.

All macros should have a "VIR" prefix in their name, followed by one or more uppercase words separated by underscores. The macro argument names should be in lowercase.

4.5.2 Bracket spacing

The keywords `if`, `for`, `while`, and `switch` must have a single space following them before the opening bracket. Function implementations and function calls must not have any whitespace between the function name and the opening bracket. Function type definitions must not have any whitespace between the closing bracket of the function name and opening bracket of the argument list. There must not be any whitespace immediately following any opening bracket, or immediately prior to any closing bracket.

4.5.3 Commas

Commas should always be followed by a space or end of line, and never have leading space.

When declaring an `enum` or using a `struct` initialiser that occupies more than one line, use a trailing comma. This helps future extensions of that list to add only a single line, rather than modify an existing line to add the intermediate comma.

4.5.4 Semicolons

Semicolons should never have a space beforehand. Inside the condition of a for loop, there should always be a space or line break after each semicolon, except for the special case of an infinite loop.

4.5.5 Curly braces

Omit the curly braces around an `if`, `while`, `for` etc. body only when both that body and the condition itself occupy a single line. In every other case braces are required.

However, there is one exception in the other direction, when even a one-line block should have braces. That occurs when that one-line, brace-less block is an `if` or `else` block, and the counterpart block does use braces.

Keeping braces consistent and putting the short block first is preferred, especially when the multi-line body is more than a few lines long, because it is easier to read and grasp the semantics of an if-then-else block when the simpler block occurs first, rather than after the more involved block.

Chapter 5

Evaluation

5.1 Experimental Design

5.1.1 Test Environment

For a Linux container to be live migrated successfully, both the local and remote host must have identical CPU architecture. In addition, if the container is using a particular hardware device (e.g. disk storage, network interface, etc.), the same device must available on destination host.

The evaluation of this project consist of two virtual machines hosted in the cloud. Both virtual servers with identical configuration setup described in Table 5.1.

Performance note: Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments or hardware configuration may vary significantly. Measurements have been made on development-level code, and there is no guarantee that these measurements will be the same on generally available releases.

Operating System	GNU/Linux
Distribution	Fedora 27 Cloud Edition
Architecture	x86_64
Kernel release	4.15.15-300
CPU(s)	4
CPU Model	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
Cache size	30720 KB
CPU Speed	2199.998 MHz
Disk size	25 Gib
RAM Memory	8167424 KiB

Table 5.1: Server Configuration

5.1.2 Experimental Setup

The CRIU tool was compiled from the source code available in the development branch (`criu-dev`) and was installed on both source and destination servers.

The evaluation was performed with four live migration mechanisms:

1. **Page-Server** - a process is checkpointed as a set of image files that are transferred to destination host, while a page server is used live migrate memory pages used by the process.

Dump	
Parameter	**Description**
Freezing time	The amount of time it takes to put the process tree into frozen state.
Frozen time	The total amount of time in which the process was in frozen state.
Memory dump time	Total amount of time during which the memory pages were extracted. This time does not include *Memory write time*.
Memory write time	Total amount of time to write (or transfer) the extracted memory pages to set of image files.
IRMAP resolve time	Time to recursively scan the process tree starting from a "known" location [31] and store all the name–inode pairs.
Memory pages scanned	Total number of scanned memory pages, that CRIU looked at to decide whether to extract them or not.
Memory pages skipped from parent	Total number of memory pages skipped in the checkpoint process due to their existence in previous pre-copy iteration, and have not been modified since the last memory tracking reset.
Memory pages written	Total number of memory pages written into image files.
Lazy memory pages	Number of memory pages send transferred during Post-copy migration.
Restore	
Parameter	**Description**
Pages compared	Total number of memory pages compared for being COW-ed.
Pages skipped COW	Skipped memory pages that were generated during fork() as Copy-on-Write [1].
Pages restored	Total number of restored memory pages.
Restore time	Total amount of time to restore a process tree.
Forking time	The amount of time to fork() the tree of processes (with groups and sessions).

Table 5.2: Checkpoint/Restore Evaluation Parameters

2. **Pre-copy** - a process is pre-dumped in total of four iterations, where after each iteration the process state is saved on source node and transferred over to destination node, and memory pages are transmitted on during each pre-dump via page server.

3. **Post-copy** - a process state is transferred over to destination host where it continues to run while all memory pages are requested on demand from source host.

4. **Image-Cache/Proxy** - Both the process state and the memory pages are transferred directly to destination host where the process is restored immediately after all the content is received. No image files are written to disk.

During checkpoint and restore the parameters described in Table 5.2 have been measured to compare different live migration mechanisms. All these parameters were measured using the built-in CRIU statistics mechanisms [29].

5.2 Results

The evaluation was performed by live migrating three types of processes with all four migration algorithms. The first process type is a memory intensive application – `memhog`. Memhog is a

benchmark program used to test NUMA memory. It achieves this by allocating given amount of memory and intensively write to it for a given number of repetitions. For the purpose of this experiment memhog allocates 1024 MiB of memory with 100 000 repetitions.

The second process type is an HTTP server [2] implemented in Python. The HTTP server is started on the source server and live migrated to the destination server while preserving open TCP connections. A client device (Raspberry PI 3) was used to keep an open connection. This also ensures that the running HTTP server remains active and has successfully restored all connections.

This third process type is a simple loop program that outputs an incremental integer to stdout (the standard output) with one second interval between each iteration.

The results from these experiments showed significant differences between the four live migration algorithms. In Figure 5.1a are shown the results of the live migration of memhog using image-cache and image-proxy. This migration algorithm has shown high fluctuation in all migration parameters in comparison to the other three algorithms.

(a) Image-Cache-Image-Proxy

(b) Page-Server

(c) Pre-Copy

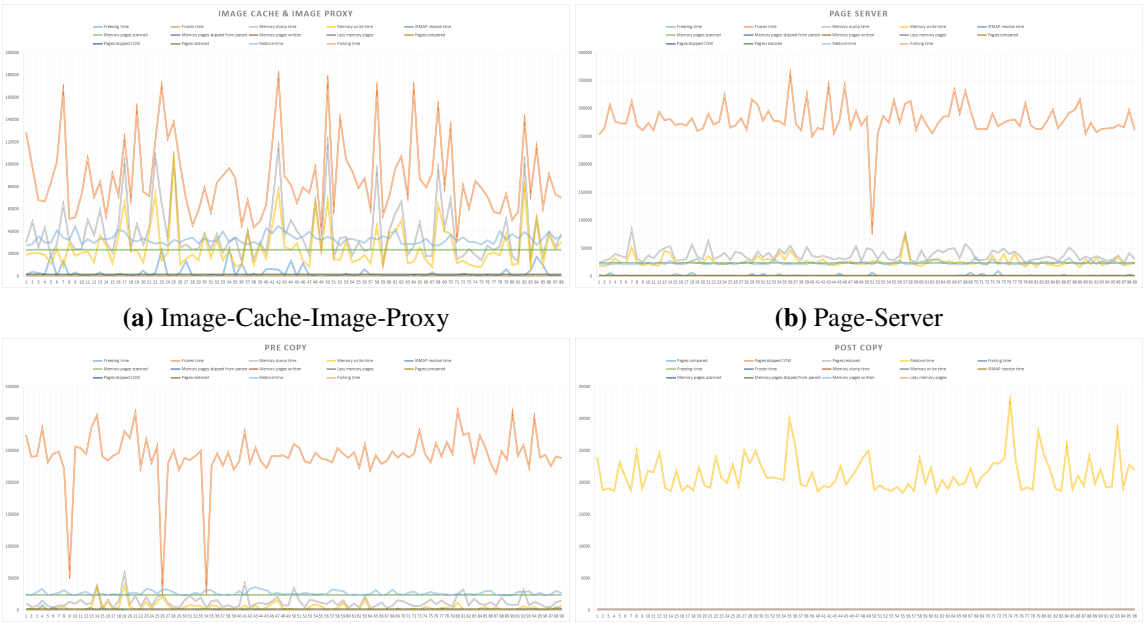
(d) Post-Copy

Figure 5.1: Live migrating memhog with 1024 MiB allocated memory

In contrast to migration with image-cache and image-proxy, the optimisation algorithms Pre-copy (Figure 5.1c), Post-copy (Figure 5.1d) and the not-optimised checkpoint/restore using page-server (Figure 5.1b) have shown more stable and predictable timing results. When migrating memory intensive process such as memhog, Post-copy optimisation performs significantly better than Pre-copy. A significant disadvantage of Pre-copy with this type of processes is that all Pre-dump iterations cover between 90% and 100% of memory images.

The second type of process that was migrated is an HTTP server implemented in Python. The process tree that needs to be migrated consist of a single process – the language interpreter *python*. The HTTP server is bind to address 0.0.0.0 and port 8080. This port is not used on either source and destination host to enable the reattaching on the remote host.

[2] https://docs.python.org/3/library/http.server.html



(a) Image-Cache-Image-Proxy

(b) Page-Server

(c) Pre-Copy

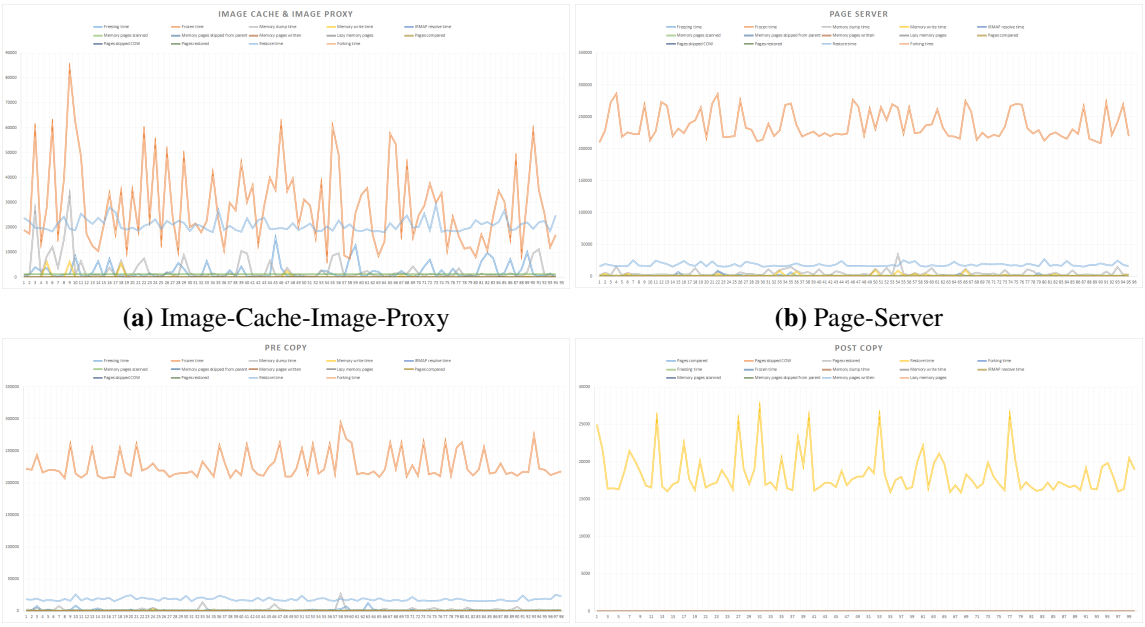
(d) Post-Copy

Figure 5.2: Live migration of HTTP server

The results show significant improvement in terms of overall performance with Pre-copy optimisation. This shows that Pre-copy is not very efficient with the memory intensive applications even when image-cache and image-proxy mechanisms are being used.

In comparison with Pre-copy optimisation and the not-optimised page-server migration algorithms, image-cache and image-proxy achieves significantly lower frozen time. In comparison with Post-copy the difference is negligible because only the process state needs to be transferred to destination host before all memory pages are being requested on demand. Therefore there is space for optimisation with Post-copy. In addition, both “image-cache and image-proxy” and Post-copy do not perform any time consuming disk I/O operations.

The third type of process that was live migrated was a simple loop counter. This process allocates relatively small amount of memory in comparison with the first two types of applications. This process also does not keep any open file descriptors neither it is bind to IP address.



(a) Image-Cache-Image-Proxy **(b)** Page-Server

(c) Pre-Copy **(d)** Post-Copy

Figure 5.3: Live migration of basic loop counter

Nevertheless, image-cache and image-proxy achieve significantly lower frozen time in comparison with Pre-copy optimisation and the not-optimised page-server migration algorithms.

5.3 Analysis

In Table 5.3 is shown the performance difference of the live migration optimisation using image-cache and image-proxy in comparison with Pre-copy, Post-copy and unoptimised migration based on checkpoint/restore using page-server. These results demonstrate the efficiency of the Post-copy migration mechanism when compared with Pre-copy for memory intensive processes such as memhog.

Table 5.3: Live migrating memhog with 1024 MiB allocated memory

Parameter	Mean	Standard Deviation
Image-cache and Image-proxy		
Freezing time	4276.581633 µs	5373.39998 µs
Frozen time	4089.745 µs	3135.8476 µs
Memory dump time	3840.878 µs	2945.1524 µs
Memory write time	3492.592 µs	5215.2707 µs
Memory pages scanned	2110	0
Memory pages written	548.0408163	0.401994196
Pages restored	262175.0309	0.39202378
Restore time	4929.227 µs	1682.996 µs
Page Server		
Freezing time	3691.395833 µs	6284.467296 µs
Frozen time	4708892.646 µs	55725.5212 µs

Memory dump time	4258941.802 µs	30293.11279 µs
Memory write time	4097673.927 µs	39296.63668 µs
Memory pages scanned	2110	0
Memory pages written	547.96875	0.529408573
Pages restored	262174.9688	0.529408573
Restore time	794668.4167 µs	39627.27383 µs
Post-Copy		
Pages restored	15	0
Restore time	21547.66327 µs	3550.867209 µs
Forking time	0.540816327 µs	0.498331243 µs
Pre-Copy		
Freezing time	4330.27 µs	6700.303882 µs
Frozen time	4527517.34 µs	26250.74684 µs
Memory dump time	4257714.6 µs	21657.27245 µs
Memory write time	4090756.18 µs	35556.98858 µs
Memory pages scanned	2106	0
Memory pages skipped from parent	25.94	0.55288453
Memory pages written	518.06	0.244449474
Pages restored	262175	0.449460216
Restore time	792990.202 µs	34354.96427 µs

The results shown in Table 5.4 demonstrate the effectiveness of Pre-copy optimisation and outline the advantages of using image-cache and image-proxy to further optimise live migration of containers.

Table 5.4: Results of Live Migrating an HTTP server

Parameter	**Mean**	**Standard Deviation**
Image-cache and Image-proxy		
Freezing time	3025.534091 µs	3042.79016 µs
Frozen time	88396.80682 µs	87937.14553 µs
Memory dump time	39329.72727 µs	39433.13326 µs
Memory write time	26477.38636 µs	26559.06121 µs
Memory pages scanned	23841	23841.23864
Memory pages written	1627.659091	1627.655217
Pages restored	1627.647727	1627.643724
Restore time	33457.72727 µs	33525.53099 µs
Forking time	0.556818182 µs	0.563145661 µs
Page Server		
Freezing time	1209.373737 µs	1818.465835 µs
Frozen time	279327.0202 µs	29395.61981 µs
Memory dump time	38038.9899 µs	10586.63424 µs

Memory write time	26285.48485 µs	8165.296547 µs
Memory pages scanned	23842	15.55634919
Memory pages written	1627.747475	0.924890759
Pages restored	1627.747475	0.924890759
Restore time	23947.82828 µs	2881.002074 µs
Forking time	0.464646465 µs	0.498748561 µs
Post-Copy		
Pages restored	209	0
Restore time	21301.53125 µs	2729.834397 µs
Forking time	0.65625 µs	0.536736376 µs
Pre-Copy		
Freezing time	958.9494949 µs	1348.788213 µs
Frozen time	241436.798 µs	40857.16239 µs
Memory dump time	12883.87879 µs	8885.423676 µs
Memory write time	4857.494949 µs	6183.802709 µs
Memory pages scanned	23842.33333	15.4344492
Memory pages skipped from parent	1554.373737	52.91494717
Memory pages written	71.7979798	53.46555198
Pages restored	1626.171717	3.771344384
Restore time	26904.46465 µs	3154.823228 µs
Forking time	0.696969697 µs	0.459568209 µs

The results shown in Table 5.5 demonstrate the base case for live migration with a simple loop counter that is migrated to a remote host and still show significant difference between Pre-copy and Post-copy migration as well as the advantages of the image-cache and image-proxy approach

Table 5.5: Results of Live Migrating a Loop Counter

Parameter	**Mean**	**Standard Deviation**
Image-cache and Image-proxy		
Freezing time	2192.925532	3043.044267
Frozen time	28967.34043	16187.59951
Memory dump time	2984.840426	5137.631726
Memory write time	263.3510638	1062.120177
Memory pages scanned	1066	0
Memory pages written	23.93617021	0.479550114
Pages restored	23.93617021	0.479550114
Restore time	20952.44681	2429.645691
Forking time	0.776595745	2.154432617
Page Server		
Freezing time	913.4736842 µs	1486.094556 µs
Frozen time	237055.6 µs	21977.9629 µs

Memory dump time	5041.326316 µs	4720.43782 µs
Memory write time	1448.410526 µs	1987.540016 µs
Memory pages scanned	1066	0
Memory pages written	23.92631579	0.416554925
Pages restored	23.92631579	0.416554925
Restore time	18200.12632 µs	2867.046804 µs
Forking time	0.547368421 µs	0.497751175 µs
Post-Copy		
Pages restored	13	0
Restore time	18362.53 µs	2658.093781 µs
Forking time	0.53 µs	0.499099189 µs
Pre-Copy		
Freezing time	1128.459184 µs	1932.860343 µs
Frozen time	226871.6939 µs	20063.01723 µs
Memory dump time	2541.091837 µs	3127.412362 µs
Memory write time	246.2857143 µs	315.6710701 µs
Memory pages scanned	1066	0
Memory pages skipped from parent	17.46938776	0.823185471
Memory pages written	6.459183673	0.498331243
Pages restored	23.92857143	0.457366017
Restore time	18432.70408 µs	2570.092598 µs
Forking time	0.714285714 µs	0.880630572 µs

5.4 Interpretation of Results

The evaluation results show that a significant effect has the speed with which memory pages are modified during live migration. Live migration of memory intensive processes shows best results with the Post-copy optimisation, when combined with image-cache and image-proxy the improvement is not very significant. The image-cache and image-proxy migration mechanism show greatest improvement in combination with the Pre-copy optimisation. In particular, the “Freezing time” and “Memory dump time” are decreased. The “Restore time” of image-cache and image-proxy in comparison with standard Pre-copy and Post-copy migration.

Chapter 6

Discussion & Explanation of Difficulties Found

Overall many challenges have been met and some of them successfully resolved during the development cycle of the project. Most of the issues were related to the checkpoint-restore tool that was used to create a stateful snapshot of a container. One of these challenges was the lack documentation. Fortunately we had the opportunity to extend the existing documentation available on the CRIU website [1]. In order to understand how the software work we had to read the source code of the CRIU project as well as the LXC, and RunC implementations of container checkpoint/restore.

Another advantage was the opportunity to communicate directly with the CRIU developers via mailing-lists and IRC channel. For some of the problems we met during development, such as failure to mount temporary file system from a process running with CAP_SYS_ADMIN capability and output error message "Invalid argument", we were advised to add `printk()` in the kernel source or use `ftrace()` to investigate this sort of problems.

Another challenge was to understand how the Libvirt and the LXC diver are implemented. This was very time consuming process because the only way to do that was read and understand the implementation as a whole. Although it seems very simple to checkpoint/restore a singe process with CRIU, it becomes much more complicated when a whole process tree is checkpointed, along with associated namespaces, cgroups, file descriptors, SElinux context, etc.

6.1 CRIU Limitations

One of the most important requirements when using CRIU for process migration is that the libraries used by the checkpointed process must be exactly the same on the source and destination system. For the restore to be successful, files that are used by the process, must have exactly the same size and path with regards to the root file system. For example, the current working directory of a checkpointed process must exist on the host where the process will be restored, and when migrating a `python` process, the file `/usr/lib/locale/locale-archive` must be exactly the same on both host and destination machine. Although this requirement conveys the impression of a severe restriction, it is not a challenge for container migration. Every Linux container has dedicated root file system. For example, when using CRIU to migrate a container (i.e., a set of processes) the root file system of that container will include all required files and libraries for all processes running inside the container.

Another limitation to note is that CRIU cannot (currently) be used to migrate applications

[1] https://criu.org/

which are directly accessing hardware through `ioctl()`. If such an application needs to be checkpointed and restarted or migrated CRIU provides an interface to create plug-ins that can be used to extract the state of the hardware on the source system and then put the hardware back into the same state during restore.

It is also important to note that it is not possible to checkpoint processes that are already being ptrace'd (e.g. gdb, strace) since CRIU is using this system call to inject a *parasite code* and gain access to memory pages of the checkpointed process.

Process identifiers (PID) are unique for each running program. However, when checkpointed and later restored, the process might not be able to obtain the same PID. In such case the restore fails with "Pid X does not match expected Y". This problem could be solved in several ways. The simplest approach is prior restore to verify that the required PID is available (e.g. by sending `SIGKILL` signal). The second approach is to restore the process in a new PID namespace, this will guarantee the availability of the PID without affecting the rest of the host processes. The latter solution applies for the case of containers when they do not explicitly share the host PID namespace.

6.2 Limitations of Linux Containers

The shared kernel approach used in container-based virtualisation has several limitations. Because all containers share the same kernel, for example, Windows containers can not be run on top of a Linux host. Server multi-tenancy brings up another trade-off in terms of security. Containers do not isolate resources as well as hypervisors [17] do because the host kernel is exposed to all processes inside containers. Another limitation is that the reduced number of system calls allowed for processes running into unprivileged containers. For example, it is not possible to perform actions like mounting a block device, or creating a special character file inside a user namespace because such operations require capabilities like CAP_SYS_ADMIN and CAP_MKNOD [77], which are not available for unprivileged users.

6.3 Limitations of Pre-copy Live Migration

The pre-copy memory transfer technique has one important limitation when used with applications that write to memory pages faster than these pages could send over the network.

During the course of the evaluation this problem was reproduced by utilising the `memhog` tool [4], which is part of the `numactl` package [6]. This program is used for benchmarks and testing. It allocates memory with NUMA (Non Uniform Memory Access) policy [5].

Three test cases have been used with parameters that perform memory allocation with size of 50, 1024, 4096 Megabytes, repeating 100 000 times. The command used to start this process is `memhog -r100000 <MEMORY_SIZE>m`. The Pre-copy algorithm was implemented using the CRIU pre-dump command. Each pre-copy migration was repeated 100 times and the average result was calculated (shown in Table 6.1).

The results demonstrate that the pre-copy algorithm, in this particular case, is very inefficient. Every pre-dump iteration copies 100% of the memory pages. To compare this special test-case example with singe dump migration, let assume that N is the number of pre-copy iterations. This test case has an overhead of N-times more disk space used during live migration on both source and destination host, as well as N-times more data is transferred over the network. This shows

Memory Size	Size of transferred memory pages file				
	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
50m	52.55 MiB	52.55 MiB	52.55 MiB	52.55 MiB	52.55 MiB
1024m	1073.86 MiB	1073.86 MiB	1073.86 MiB	1073.86 MiB	1073.86 MiB
4096m	4083.79 MiB	4083.79 MiB	4083.79 MiB	4083.79 MiB	4083.79 MiB

Table 6.1: Pre-copy migration with five iterations

Memory Size	Size of memory pages file
50m	52.55 MiB
1024m	1073.86 MiB
4096m	4083.79 MiB

Table 6.2: Single Pre-copy iteration

that Pre-copy approach could be very inefficient. Moreover, if the maximum number of pre-copy iterations is not enforced, the live migration for similar application might not be able to complete in finite amount of time.

In comparison with single dump of memory pages for with same three test cases we obtain similar migration performance to the last iteration of pre-copy. The results are shown in Table 6.2.

6.4 Limitations of Post-copy Live Migration

One important limitation of Post-copy migration is that during live migration if either of the hosts (source or destination) fails, or the network connection between them drops, the process migration will fail. In such case the running state of the process can not be recovered because the process is running on the destination node and memory pages are kept on the source during migration. The problem is that after modification, changes are not send back to the source host, and therefore it will not be able to recover the latest state of the process. There are a number of proposed solutions [58, 49], however to best of my knowledge, none of them has been effectively implemented. The solutions suggest a mechanism to incrementally checkpoint the memory state on the destination node and propagate memory changes back at the source node. This mechanism can provide a consistent backup image at the source node that one can fall back upon in case the destination node fails in the middle of post-copy migration.

CRIU implements Post-copy migration by running a page-server daemon that is used to connect with destination host. A significant limitation of Post-Copy migration as implemented with CRIU is that it does not implement encryption for the connection between source and destination node. This implies that a man-in-the-middle attack could be performed during Post-copy live migration with CRIU.

6.5 Limitations of image-cache & image-proxy

When live migration process using the `--remote` flag with `image-cache` and `image-proxy`, one very important assumption is being made - we have enough memory. Live migrating such process literally duplicates all process memory pages on the source node. This problem implies that we will not be able to migrate a process if we do not have RAM memory space available, that is equivalent to the amount of memory used by the process.

Similarly to Post-copy, the image-cache and image-proxy do not offer encryption out of the box. However, this problem could be solved by implementing the cache and proxy alternatives with libvirt to achieve *tunnelled live migration*. In such scenario an encrypted TLS socket could be used to transfer the image files and memory pages.

6.6 Independent and Dependent Variables

The independent variables are those variables, whose variations does not depend on the value of any another variable. In the present study the choice of container runtime system is an independent variable because all container runtime systems explored in this work (Docker, LXC/LXD, RunC and Libvirt-LXC) implement live migration using features provided by the CRIU tool. Therefore all measured metrics depend only on the implementation in CRIU and do not depend on the container runtime system.

Dependent variables are those variables, whose values are determined with the help of the variable. The performance metrics - CPU utilisation, disk utilisation, down time and total migration time, are very dependant on the resource usage and the type of memory operations performed by processes running inside a container. If the load of container is changed, then all the values of these metrics are likely to change as well.

All experiments conducted throughout the evaluation of this study measure efficiently the performance of the live migration by using processes with different intensity and type of performed memory operations.

Chapter 7

Summary and Conclusion

This chapter will conclude this project by proposing possible future work optimisations.

7.1 Project Summary & Critical Evaluation

Looking over the initial project requirements, the implemented features that extend the Libvirt-LXC driver successfully fulfil all of them.

All four scenarios have been used during the evaluation and the results show that the current implementation provides sufficient support for all of them.

The functional requirements **FR1**, **FR2** and **FR3** have been completed by the current implementation of the LXC driver and they also guarantee that all previously implemented features continue to work as expected.

FR4 and **FR5** have been completed by extending the PoC implemented in [103]. These requirements have been successfully fulfilled with implementation of the commands *save* and *restore*.

The non-functional requirement **NFR1** was guaranteed with the threshold and limitation parameters defined in the `lxc-migration.h` header file, as described in Table 4.3. **NFR2** has been fulfilled by storing the XML configuration of a container along with all other image files produced during checkpoint. During live migration this XML definition is transmitted a over tcp socket a an serialised object that is used directly by the remote host. **NFR3** is completed at the present time, as both CRIU and Libvirt are compatible with recent versions of Linux.

During the development of this project the design and implementation have been consulted with both Libvirt and CRIU communities to ensure that all changes will be accepted for merge in a future release.

7.2 Future Work Suggestions

There has been significant progress with regards to making Live process migration possible for the Linux kernel. However several issues related to checkpoint and restore of processes still remain to be solved.

A process that is bind to particular IP address needs to be restored on a system where this same IP address is available [30]. This problem is challenging because just changing the IP address and letting the things work as they used to is not possible due to how the TCP protocol works. A TCP connection operates according to the protocol in a way that one cannot proceed the packet flow with one IP address changed, the client would just ignore such packets. A simple solution is

to leave such connections closed when dumping a container.

Another challenging problem is that when a new user namespace is created its capability get dropped off, thus any kernel aspect guarded with capable() may fail. In particular the `mknod()` and `mount()` system calls do not function as a privileged process would expect.

This problem is important for Libvirt-LXC containers with enabled user namespace because in such containers device nodes are created as a real root on a temporary file system, just before the init process is being started. When CRIU performs dump it extracts all these special character device files into a tar archive, again as real root on the host. However, on restore, the file extraction from the archive is performed as the unprivileged user running inside the user namespace. This makes CRIU fail during restore. A possible solution is to perform the container restore as a privileged user that has the permission to perform `mknod()` system call.

Another issue is that processes running inside a Linux container with enabled user namespace are not allowed to perform `chown()` system call. This is due to restriction in the Kernel that does not allow unprivileged processes to change the ownership of files that they do not own. However, when running in user namespace a process is given the illusion to be a privileged process and therefore it fails unexpectedly when tries to `chown()` a file. Solution to this problem are not yet known or proposed to best of my knowledge.

7.3 Conclusion & Personal Reflection

Despite the many use cases and ongoing research efforts, process migration has not achieved widespread use. One reason for this is the high complexity of adding transparent migration to operating systems originally designed to run stand-alone, since designing new OS kernel with migration in mind is not a realistic option. Another reason is that there has not been a compelling commercial argument for operating system vendors to support process migration.

Checkpoint-restart is an approach that offers a compromise between introduced complexity and provided functionality. This is achieved by introducing components that can run on more loosely-coupled systems and restricting the types of processes that could be migrated.

Virtual machine migration has been very successful since its first introduction in [22]. It provides all befits of live process migration with reduced complexity. This feature has become an important management functionality for platform virtualisation vendors, such as VMware, and open source virtualisation and emulation projects such as QEMU and KVM.

Linux containers provide lower overhead in performance when compared with VMs. Applications running inside a container have the same performance as applications running on bare-metal servers with advantages providing isolation and resource management.

Live migrating Linux containers using checkpoint-restore mechanism enables simplified implementation that works within userspace for recent versions of the Linux kernel. The advantage of migrating a Linux container instead of a single process is that the process tree that is being isolated with kernel namespaces and control groups is completely self-contained, with its own inter-process communication, process identifiers, mount points, devices, root file system, etc. This allows to avoid the problem of residual decencies on the source node.

The CRIU project implements all basic components needed to achieve live migration for Linux containers. This has been realised in the Docker project as well as RunC, and LXC/LXD.

At the time of writing this report, all these projects implement very basic form of live migration. Only RunC provides the flexibility to achieve Pre-copy and Post-copy migration and LXC/LXD enables Pre-copy optimisation when supported.

The Libvirt project implements the concept of Linux containers with the Libvirt-LXC driver. This implementation does not have dependency on the LXC project and shares most of the domain independent features of Libvirt.

Libvirt has the advantage to manage both VM Hypervisors and Linux containers from a single **stable** API. This allows to build tools on top of Libvirt that can make use of virtualisation capabilities implemented in the Linux Kernel.

For cloud service and hosting providers to achieve efficiency in managing their workload while preserving the quality of the service, is very important to make balance between both VM and containers. Containers allow to run applications within userspace with almost no performance overhead, while VMs allow to run completely isolated kernel instance that can provide suitable running environment for different any type of applications. It is preferable to use Linux containers when possible but it is also necessary to provide means to run applications that, for example require different kernel, within a VM.

Appendix A

User Manual

The following user manual is based on the Libvirt [1] and CRIU [2] documentations as well as articles about Libvirt-LXC [93] driver with extended and explained usage examples.

A.1 Virsh command line tool

The libvirt project includes a general-purpose command-line tool called `virsh` that provides an interface for administration of virtualisation domains. It can be used to manage the capabilities of LXC domains – create, modify, and destroy containers, display information about existing containers, or manage resources, storage, and network connectivity of a container.

Libvirt-LXC supports two types of Linux containers – *persistent* and *volatile*. Persistent containers are preserved after reboot. They are defined with an XML configuration file. Temporary containers are deleted as soon as the contained application finishes, you can create them with the `virsh create` command. In Table A.1 are described the most frequently used `virsh` commands that are in connection with Libvirt-LXC containers.

Command	Description
define	Create a new container based supplied XML configuration file.
undefine	Deletes a container. If the container is in running state, it will be converted to a transient container.
start	Starts a previously-defined container. By specifying the `--console` option, the you will be connected directly to the pseudo terminal of the newly created container. If the `--autodestroy` option is specified, the container will be automatically stopped when `virsh` is disconnected.
autostart	Configures a domain to start automatically on system boot.
create	Defines and starts a non-persistent container in one step. The temporary container is based on supplied libvirt XML configuration file.
console	Connects to the virtual console of the container.
shutdown	Coordinates with the domain operating system to perform a graceful shutdown. The exact behaviour can be specified in the container's XML definition.
destroy	Immediately (forcefully) terminates the container.

Table A.1: Common virsh commands

[1] `https://libvirt.org/docs.html`
[2] `https://criu.org/Main_Page`

In order to modify the XML configuration file of a domain, use the `edit` command followed by the container name. This command will open the default text editor and on save, it will validate the changes before applying them.

A.2 Creating a Container

To create a Linux Container using the virsh utility, follow these steps:

1. Create a Libvirt configuration file in the XML format. Below is given an example of XML definition for a container with name “container_name” and memory limit of 1048576

```
<domain type="lxc">
   <name>container_name</name>
   <memory>1048576</memory>
   <os>
       <type arch="x86_64">exe</type>
       <init>/bin/bash</init>
   </os>
   <features>
       <privnet/>
   </features>
   <devices>
       <emulator>/usr/libexec/libvirt_lxc</emulator>
       <filesystem type="mount">
              <source dir="/var/lib/libvirt/filesystems/container_name"/>
          <target dir="/"/>
       </filesystem>
       <console type="pty"/>
   </devices>
</domain>
```

2. Import a new container to Libvirt.

```
virsh -c lxc:/// define container_name.xml
```

3. Starting, Connecting to, and Stopping a Libvirt-LXC Container

   To start a previously-defined container, use the following command as root:

```
virsh -c lxc:/// start container_name
```

   Replace container_name with a name of the container. Once a container is started, connect to it using the following command:

```
virsh -c lxc:/// console container_name
```

   Note that if a container uses the `/bin/bash` process as the init process with a PID of 1, exiting the shell will also shut down the container.

   To stop a running container, execute the following command as root:

```
virsh -c lxc:/// shutdown container_name
```

If a container is not responding, it can be shut down forcefully by executing:

```
virsh -c lxc:/// destroy container_name
```

4. Modifying a Container To modify any of the configuration settings of an existing container, run the following command as root:

   ```
   virsh -c lxc:/// edit container_name
   ```

   With container_name, specify the container whose settings you wish to modify. The above command opens the XML configuration file of the specified container in a text editor and lets you change any of the settings. The default editor option is `vi`, it can be changed by setting the `EDITOR` environment variable to your editor of choice.

   Once the file has been edited, save the file and exit the editor. After doing so, virsh edit automatically validates your modified configuration file and in case of syntax errors, it prompts you to open the file again. The modified configuration takes effect next time the container boots. To apply the changes immediately, reboot the container (as root):

   ```
   virsh -c lxc:/// reboot container_name
   ```

5. Automatically Starting a Container on Boot To start the container automatically on boot, use the following command as root:

   ```
   virsh -c lxc:/// autostart container_name
   ```

   To disable this automatic start, type as root:

   ```
   virsh -c lxc:/// autostart --disable container_name
   ```

   Use the virsh dominfo command to test your new setting:

   ```
   virsh -c lxc:/// dominfo test-container | grep Autostart
   ```

6. Removing a Container To remove an existing container, run the following command as root:

   ```
   virsh -c lxc:/// undefine container_name
   ```

   Undefining a container simply removes its configuration file. Thus, the container can no longer be started. If the container is running and it is undefined, it enters a transient state in which it has no configuration file on the disk. Once a transient container is shut down, it can not be started again.

7. Monitoring a Container To view a simple list of all existing containers, both running and inactive, type the following command as root:

   ```
   virsh -c lxc:/// list --all
   ```

   Once you know the name of a container, or its process ID if it is running, view the metadata of this container by executing the following command:

```
virsh -c lxc:/// dominfo container_name
```

When running a large number of containers simultaneously, you may want to gain an overview of containerized processes without connecting to individual containers. In this case, use the `systemd-cgls` command that groups all processes within a container into a cgroup named by the container. As an alternative, use the `machinectl` command to get information about containers from the host system. First, list all running containers as shown in the following example:

View the status of one or more containers by executing:

```
machinectl status -l container_name
```

8. Networking with Linux Containers

   The guests created with `virsh` can by default reach all network interfaces available on the host system. If the container configuration file does not list any network interfaces, the network namespace is not activated, allowing the containerized applications to bind to TCP or UDP addresses and ports from the host operating system. It also allows applications to access UNIX domain sockets associated with the host. To forbid the container an access to UNIX domains sockets, add the `<privnet/>` flag to the `<features>` parameter of the container configuration file. With network namespace, it is possible to dedicate a virtual network to the container. This network has to be previously defined with a configuration file in XML format stored in the `/etc/libvirt/lxc/networks/` directory. Also, the virtual network must be started with the `virsh` net-start command.

   To connect your container to a predefined virtual network, type as root:

   ```
   virsh attach-interface container_name type eth0 --mac mac --config
   ```

   Replace `container_name` with the name of the container, `eth0` with the name of the network interface, and `mac` with a mac address to be given to that container (e.g. `00:0a:95:9d:68:16`). The `type` could be either `network` or `bridge`. The `--config` option is used to make the network attachment persistent.

   To disconnect the container from the virtual network, type as root:

   ```
   virsh detach-interface container_name [network|bridge] --config
   ```

   A network bridge is a link-layer layer device which forwards traffic between networks based on MAC addresses. It makes forwarding decisions based on a table of MAC addresses which it builds by listening to network traffic and thereby learning what hosts are connected to each network. A software bridge can be used within a Linux host in order to emulate a hardware bridge, especially in virtualization applications for sharing a NIC with one or more virtual NICs. Ethernet bridging is useful for machines with permanent wired LAN connection. Once the host networking is configured to have a bridge device, you can use this bridge for a virtual network. This requires creating a configuration file and then loading it into libvirt.

9. Mounting Devices to a Container

   To mount a device to the guest file system, type as root:

   ```
   virsh attach-device contianer_name file --config
   ```

   Where `file` stands for a libvirt configuration file for this device and the `--config` option is used to make this change persistent.

   To detach a previously mounted device, type:

   ```
   virsh detach-device container_name file --config
   ```

   where `file`, and `--config` have the same meaning as with `attach-device`.

A.3 Save & Restore Container State

A Libvirt-LXC container can be checkpointed as a set of image files to persistent storage. This can files then can be used to restore the running state of the container.
To checkpoint running container type:

```
virsh save container_name <Path>
```

Where `<Path>` is the absolute path where the image files of that container will saved.
To restore the running state of a container use the following command:

```
virsh restore <Path>
```

Where `<Path>` is the absolute path where are stored the image files of a container.

A.4 Live Migrate Container

Live migration support in Libvirt requires a remote connection to be configured and to ensure that it is possible to connect with the remote host. The following command will migrate a running instance of a container from the local to remote host.

```
virsh migrate container_name lxc+tls://<remote_host>:<port> --live
```

Where `<remote_host>` is the host name of the remote server and `<port>` is the port used to establish connection. The default port for TLS is 16509.

Libvirt can use the `remote` driver to connect to a libvirt daemon instance on remote server. There are several options that can be used to establish connection. The default transport, if no other is specified, is `tls`. Remote libvirt supports a range of transports:

1. **tls**

   TLS 1.0 (SSL 3.1) authenticated and encrypted TCP/IP socket, usually listening on a public port number. To use this you will need to generate client and server certificates. The standard port is 16514.

2. **unix**

   Unix domain socket. Since this is only accessible on the local machine, it is not encrypted, and uses Unix permissions or SELinux for authentication. The standard socket names are `/var/run/libvirt/libvirt-sock` and `/var/run/libvirt/libvirt-sock-ro` (the latter for read-only connections).

3. **ssh**

   Transported over an ordinary ssh (secure shell) connection. Requires Netcat (nc) installed and libvirtd should be running on the remote machine. You should use some sort of ssh key management (eg. ssh-agent) otherwise programs which use this transport will stop to ask for a password.

4. **ext**

   Any external program which can make a connection to the remote machine by means outside the scope of libvirt.

5. **tcp**

   Unencrypted TCP/IP socket. Not recommended for production use, this is normally disabled, but an administrator can enable it for testing or use over a trusted network. The standard port is 16509.

6. **libssh2**

   Transport over the SSH protocol using libssh2 instead of the OpenSSH binary. This transport uses the libvirt authentication callback for all ssh authentication calls and therefore supports keyboard-interactive authentication even with graphical management applications. As with the classic ssh transport netcat is required on the remote side.

7. **libssh**

   Transport over the SSH protocol using libssh instead of the OpenSSH binary. This transport uses the libvirt authentication callback for all ssh authentication calls and therefore supports keyboard-interactive authentication even with graphical management applications. As with the classic ssh transport netcat is required on the remote side.

Preferred connection is TLS (the default) or SSH using public-key authentication. It is important to establish encrypted connection with the remote host prior migration, especially if this host is publicly available on the network.

Appendix B

Maintenance Manual

B.1 Installation instructions

B.1.1 Installing Libvirt

Libvirt uses the standard configure/make/install steps:

```
xz -c libvirt-x.x.x.tar.xz | tar xvf -
cd libvirt-x.x.x
./configure
```

The `configure` script can be given options to change its default behaviour. To get the complete list of the options it can take, pass it the `--help` option like this:

```
./configure --help
```

The use of `sudo` might be required with the `make install` command below. Using `sudo` is only required when installing to a location your user does not have write access to. If you are installing to a location that your user does have write access to, then you can instead run the `make install` command without putting `sudo` before it.

```
./configure [possible options]
make
sudo make install
```

B.1.2 Installing CRIU

CRIU can be compiled by simply running `make` in the source directory. To install CRIU run `make install`

B.2 Software dependencies

B.2.1 Libvirt dependencies

The dependencies registered as part of the libvirt package will normally assure that all necessary binaries are present, but on some systems (e.g. gentoo) the practice of building your own packages, and allowing "minimal" packages, can lead to problems. For example, the binary `/sbin/ip`, which is part of the `iproute` package (called `iproute2` in some cases) is required to set the IP addresses of the bridge devices used by libvirt, but it's possible on gentoo to build the `iproute2` package "minimally" which results in no `/sbin/ip`. The symptom will be that libvirt complains it can't set the bridge device IP address.

Here is a list of the networking-related packages that need to be installed on the host for libvirt networking to work properly:

Package	**Description**
avahi	Avahi is a system which facilitates service discovery on a local network.
ceph-libs	Ceph is a distributed storage system that runs on commodity hardware and delivers object, block and file system storage.
curl	curl is a tool to transfer data from or to a server.
dbus	D-BUS is a system for sending messages between applications.
e2fsprogs	The e2fsprogs package contains a number of utilities for creating, checking, modifying, and correcting any inconsistencies in second, third and fourth extended (ext2/ext3/ext4) file systems.
fuse2	This package contains the FUSE v2 userspace tools to mount a FUSE file system.
gcc-libs	This package is needed in order to compile C code.
gettext	The GNU gettext package provides a set of tools and documentation for producing multi-lingual messages in programs.
gnutls	GnuTLS is a secure communications library implementing the SSL, TLS and DTLS protocols and technologies around them.
iproute2	The iproute package contains networking utilities which are designed to use the advanced networking capabilities of the Linux kernel.
iptables	The iptables utility controls the network packet filtering code in the Linux kernel.
libcap-ng	Libcap-ng is a library that makes using posix capabilities easier.
libgcrypt	General purpose cryptographic library based on the code from GnuPG.
libgpg-error	Support library for libgcrypt.
libnl	A collection of libraries providing APIs to netlink protocol based Linux kernel interfaces.
libpcap	A portable C/C++ library for network traffic capture.
libpciaccess	X11 PCI access library.
libsasl	Simple Authentication and Security Layer, a method for adding authentication support to connection-based protocols.
libssh2	A library implementing the SSH2 protocol.
libx11	This package provides a client interface to the X Window System.
libxau	This package provides the main interface to the X11 authorisation handling, which controls authorisation for X connections, both client-side and server-side.
libxcb	This package contains the library files needed to run software using libxcb-composite, the composite extension for the X C Binding.
libxdmcp	This package provides the main interface to the X11 display manager control protocol library, which allows for remote logins to display managers.

libxml2	XML is a metalanguage to let you design your own markup language. A regular markup language defines a way to describe information in a certain class of documents.
netcf	Netcf is a library used to modify the network configuration of a system. Network configurations are expressed in a platform-independent XML format, which netcf translates into changes to the system's 'native' network configuration files.
numactl	NUMA scheduling and memory placement tool.
openssl	Secure Socket Layer (SSL) binary and related cryptographic tools.
parted	Disk partition manipulator.
polkit	Application development toolkit for controlling system-wide privileges.
python2 or python3	Several Python scripts are used to generate documentation and API bindings.
bridge-utils	Utilities for configuring the Linux Ethernet bridge.
module-init-tools	This dummy package is provided to support the transition from module-init-tools to kmod and should be removed afterwards.
dnsmasq	Small caching DNS proxy and DHCP/TFTP server.
radvd	Router Advertisement Daemon.
ebtables	Ethernet bridge frame table administration.

B.2.2 CRIU dependencies

The migration requires both Libvirt and CRIU to be installed on the machine. CRIU has the following dependencies:

Package	**Description**
protobuf	A language-neutral, platform-neutral extensible mechanism for serializing structured data.
protobuf-c	This is protobuf-c-rpc, a library for performing RPC with protobuf-c.
protobuf-c-devel	This package contains protobuf-c headers and libraries.
protobuf-compiler	compiler for protocol buffer definition files.
protobuf-devel	This package contains Protocol Buffers compiler for all languages and C++ headers and libraries.
protobuf-python	This package contains Python libraries for Google Protocol Buffers.
pkg-config	This package is used to check build library dependencies.
python-ipaddr	This library is used by CRIT to pretty-print ip.
libbsd	This package is used to provide setproctitle() support that will allow to make process titles of service workers to be more verbose.

libcap-devel	Development files (Headers, libraries for static linking, etc) for libcap.
libnet-devel	The libnet-devel package includes header files and libraries necessary for developing programs which use the libnet library.
libnl3-devel	This package contains various headers for using libnl3.
asciidoc	AsciiDoc is a text document format for writing short documents, articles, books and UNIX man pages.
xmlto	This is a package for converting XML files to various formats using XSL stylesheets.

B.3 Future adaptations and extensions

Libvirt and CRIU are both open-source projects that have many contributors. Both projects are always looking for new contributors to participate in ongoing activities. While code development is a major part of the project, assistance is needed in many other areas including documentation writing, bug triage, testing, application integration, website / wiki content management, translation, branding, social media, etc. eneral tips for contributing patches are described below.

1. Discuss any large changes on the mailing list first. Post patches early and listen to feedback.
2. Post patches using `git send-email`.
3. In your commit message, make the summary line reasonably short (60 characters is typical), followed by a blank line, followed by any longer description of why your patch makes sense.
4. Split large changes into a series of smaller patches, self-contained if possible, with an explanation of each patch and an explanation of how the sequence of patches fits together.

Appendix C

Pre-copy Migration

Listing C.1: pre-copy-migration

```
1  #!/bin/bash
2
3  ###########################################################
4  # Settings
5  #
6  # ITER      - Number of Pre-copy iterations.
7  # IMAGES_DIR  - Folder where image files will be stored.
8  # PORT      - Port used for migration.
9  # PID     - PID of process to migrate.
10 ###########################################################
11
12 ITER=5
13 IMAGES_DIR=<DIR>
14 DEST_HOST=<HOST_IP_ADDRESS>
15 PORT=<PORT>
16 PID=$(pidof $1)
17
18 ###################################
19 # Create directory tree where the
20 # checkpoint will be stored.
21 ###################################
22
23 DIRS=''
24 for ((i=0; i<=$ITER; i++)); do
25   DIRS="$DIRS $IMAGES_DIR/$i"
26 done
27
28 FINAL_DUMP_DIR=$IMAGES_DIR/$(($ITER+1))
29
30 rm -rf $IMAGES_DIR
31 mkdir -p $DIRS $FINAL_DUMP_DIR
32
33 ssh $DEST_HOST "rm -rf $DIRS $FINAL_DUMP_DIR; mkdir -p $DIRS
      $FINAL_DUMP_DIR"
34
35 ####################################
36 # Initial pre-dump
37 # It does not have --prev-images-dir
38 ####################################
39
```

```
40 ssh $DEST_HOST "criu page-server --images-dir $IMAGES_DIR/0 --port
       $PORT --daemon"
41
42 criu pre-dump \
43   --tree $PID \
44   --images-dir $IMAGES_DIR/0 \
45   --page-server \
46   --address $DEST_HOST \
47   --port $PORT \
48   --shell-job \
49   --track-mem
50
51
52 ##############################
53 # Pre-dump loop
54 ##############################
55
56 for (( i=1; i<=$ITER; i++ )); do \
57
58   PREV_DIR=$IMAGES_DIR/$(($i-1))
59
60   ssh $DEST_HOST "criu page-server --images-dir $IMAGES_DIR/$i
         --port $PORT --prev-images-dir $PREV_DIR --daemon"
61
62   criu pre-dump \
63     --tree $PID \
64     --images-dir $IMAGES_DIR/$i \
65     --prev-images-dir $PREV_DIR \
66     --page-server \
67     --address $DEST_HOST \
68     --port $PORT \
69     --shell-job \
70     --track-mem
71 done
72
73 ##############################
74 # Final dump
75 ##############################
76
77 PREV_DIR=$IMAGES_DIR/$(($i-1))
78
79 ssh $DEST_HOST "criu page-server --images-dir $FINAL_DUMP_DIR
       --port $PORT --prev-images-dir $PREV_DIR --daemon"
80
81 criu dump --tree $PID \
82   --display-stats \
83   --images-dir $FINAL_DUMP_DIR \
84   --prev-images-dir $IMAGES_DIR/$ITER \
85   --leave-stopped \
86   --page-server \
87   --address $DEST_HOST \
88   --port $PORT \
89   --shell-job \
90   --track-mem
91
92
93 ##############################################
```

```
94  # Copy image files to remote host
95  # Here we also copy symlinks to preserve links
96  # between previous image dirs.
97  ##############################################
98
99  rsync -a $FINAL_DUMP_DIR/ $DEST_HOST:$FINAL_DUMP_DIR
100 kill -9 $PID
101 ssh $DEST_HOST "criu restore --images-dir $FINAL_DUMP_DIR
        --shell-job --restore-detached"
```

Appendix D

Post-copy Migration

Listing D.1: post-copy-migration

```
#!/bin/bash

#############################################################
# Settings
#
# IMAGES_DIR - Folder where image files will be stored
#              This path is used on both src and dst.
# PORT       - Port used for migration
# PID        - PID of process to migrate
# SRC_HOST   - Address of source node
# DEST_HOST  - Address of destination node
#############################################################

IMAGES_DIR=/tmp/dump
SRC_HOST=$(gethostip -d src)
DEST_HOST=$(gethostip -d dst)
PORT=1234
PID=$(pidof $1)

########################################
# Start Process Dump with --lazy-pages
########################################
criu dump \
--tree $PID \
--images-dir $IMAGES_DIR \
--lazy-pages \
--address 0.0.0.0 \
--port $PORT \
--shell-job \
--display-stats &

sleep 2
########################################
# Copy image files to destination host
########################################
scp -r $IMAGES_DIR/* $DEST_HOST:$IMAGES_DIR

########################################
# Start userfaultfd daemon
########################################
```

```
41 ssh $DEST_HOST "cd $IMAGES_DIR && criu lazy-pages --page-server
      --address $SRC_HOST --port $PORT --display-stats && sleep 1 &&
      pkill -9 $1" &
42
43 sleep 1
44
45 ssh -t $DEST_HOST "cd $IMAGES_DIR && criu restore -D $IMAGES_DIR
      --lazy-pages --shell-job --display-stats"
```

Appendix E

Page-server Migration

Listing E.1: page-server-migration

```
#!/bin/bash

############################################################
# Settings
#
# IMAGES_DIR - Folder where image files will be stored
#              This path is used on both src and dst.
# PORT       - Port used for migration
# PID        - PID of process to migrate
# DEST_HOST  - Address of destination node
############################################################

DEST_HOST=$(gethostip -d dst)
IMAGES_DIR=/tmp/dump
PORT=1234
PID=$(pidof $1)

#################################
# Create directory tree where the
# checkpoint will be stored.
#################################

rm -rf $IMAGES_DIR
mkdir -p $IMAGES_DIR

ssh -q $DEST_HOST "rm -rf $IMAGES_DIR; mkdir -p $IMAGES_DIR"

#####################################
# Migration
#####################################

ssh -q $DEST_HOST "criu page-server --images-dir $IMAGES_DIR
    --port $PORT --daemon"

criu dump \
  --tree $PID \
  --images-dir $IMAGES_DIR \
  --display-stats \
  --page-server \
  --address $DEST_HOST \
  --port $PORT
```

```
41
42 ###############################################
43 # Copy image files to remote host
44 ###############################################
45
46 rsync -a $IMAGES_DIR/ $DEST_HOST:$IMAGES_DIR
47 ssh -q -t $DEST_HOST "criu restore --images-dir $IMAGES_DIR
       --display-stats --restore-detached && kill $PID"
48
49 kill -9 $PID > /dev/null
```

Appendix F

Image-cache & Image-proxy Migration

Listing F.1: image-cache-proxy-migration

```
#!/bin/bash

############################################################
# Settings
#
# IMAGES_DIR - Folder where image files will be stored
#              This path is used on both src and dst.
# PORT       - Port used for migration
# PID        - PID of process to migrate
# DEST_HOST  - Address of destination node
############################################################

DEST_HOST=$(gethostip -d dst)
IMAGES_DIR=/tmp/dump
PORT=1234
PID=$(pidof $1)


rm -rf $IMAGES_DIR &> /dev/null
mkdir -p $IMAGES_DIR
ssh -q $DEST_HOST "rm -rf $IMAGES_DIR; mkdir -p $IMAGES_DIR"


#####################################
# Start cache server on remote node.
#####################################

ssh -q $DEST_HOST "criu image-cache -d --port $PORT -D $IMAGES_DIR"


#####################################
# Start proxy on source node.
#####################################

criu image-proxy -d --port $PORT --address $DEST_HOST -D
    $IMAGES_DIR


#####################################
# Dump on the process on source node.
#####################################
```

```
41
42 criu dump -t $PID -D $IMAGES_DIR --remote --display-stats
43
44 ssh -q $DEST_HOST "criu restore --remote -D $IMAGES_DIR
       --restore-detached --display-stats && kill -9 $PID"
45
46 kill -9 $PID &> /dev/null
```

Bibliography

[1] Augeas configuration api. http://augeas.net/, Accessed: 2018-04-29.

[2] Gnu autoconf. https://www.gnu.org/software/autoconf/autoconf.html, Accessed: 2018-04-29.

[3] Gnu automake. https://www.gnu.org/software/automake/, Accessed: 2018-04-29.

[4] memhog - allocate memory with policy for testing. https://github.com/numactl/numactl/blob/master/memhog.c, Accessed: 2018-04-29.

[5] numa - numa policy library. https://linux.die.net/man/3/numa, Accessed: 2018-04-29.

[6] Numa policy support. https://github.com/numactl/numactl, Accessed: 2018-04-29.

[7] Criu - checkpoint/restore in user space. https://access.redhat.com/articles/2455211, Oct 2016. Accessed: 2018-03-28.

[8] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

[9] Adrian Reber. From Checkpoint/Restore to Container Migration. https://rhelblog.redhat.com/2016/09/26/from-checkpointrestore-to-container-migration/, Sept 2016.

[10] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel. *Processes migrate in Charlotte*. University of Wisconsin-Madison, Computer Sciences Department, 1986.

[11] Malcolm P. Atkinson, Laurent Daynes, Mick J. Jordan, Tony Printezis, and Susan Spence. An orthogonally persistent java. *ACM Sigmod Record*, 25(4):68–75, 1996.

[12] Amnon Barak and Richard Wheeler. *MOSIX: An integrated unix for multiprocessor workstations*. International Computer Science Institute, 1988.

[13] Block IO Controller. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt, Jan 2016.

[14] Neil Brown. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt, March 2010.

[15] Rodrigo Bruno and Paulo Ferreira. Alma: Gc-assisted jvm live migration for java server applications. In *Proceedings of the 17th International Middleware Conference*, page 5. ACM, 2016.

[16] Edouard Bugnion, Scott W Devine, and Mendel Rosenblum. Virtual machine monitors for scalable multiprocessors, June 13 2000. US Patent 6,075,938.

[17] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[18] capabilities(7). Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/capabilities.7.html, Feb 2018.

[19] CFS Scheduler. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt, Aug 2016.

[20] cgroup namespaces(7). Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html, Sept 2017.

[21] Kasidit Chanchio and Phithak Thaenkaew. Time-bound, thread-based live migration of virtual machines. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 364–373. IEEE, 2014.

[22] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[23] clone(2). Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/clone.2.html, Sept 2017.

[24] David L Cohn, William P Delaney, and Karen M Tracey. Arcade: A platform for heterogeneous distributed operating systems. 1989.

[25] Jonathan Corbet. Notes from a container. https://lwn.net/Articles/256389/, Oct 2007.

[26] Geoff Coulson, Gordon S Blair, Philippe Robin, and Doug Shepherd. Extending the chorus micro-kernel to support continuous media applications. In *International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 54–65. Springer, 1993.

[27] CPU Accounting Controller. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/cgroup-v1/cpuacct.txt, Jan 2016.

[28] credentials(7). Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/credentials.7.html, Dec 2016.

[29] CRIU Documentation. Statistics. https://criu.org/Statistics, Dec 2014.

[30] CRIU Documentation. Change IP address. https://criu.org/Change_IP_address, Jan 2017.

[31] CRIU Documentation. Irmap. https://criu.org/Irmap, Jan 2017.

[32] CRIU Documentation. Disk-less migration. https://criu.org/Disk-less_migration, Sept 2016.

[33] CRIU Documentation. Memory changes tracking. https://criu.org/Memory_changes_tracking, Sept 2016.

[34] Bubai Das, Kunal Kumar Mandal, and Suvrojit Das. Improving total migration time in live virtual machine migration. In *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015*, pages 57–61. ACM, 2015.

[35] Partha Dasgupta, Richard Joseph LeBlanc, and William F Appelbe. The clouds distributed operating system: Functional description, implementation details and related work. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 2–9. IEEE, 1988.

[36] Damien De Paoli and Andrzej Goscinski. The rhodos migration facility. *Journal of Systems and Software*, 40(1):51–65, 1998.

[37] Device Whitelist Controller. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/cgroup-v1/devices.txt, Jan 2016.

[38] Oracle Documentation. Oracle solaris zones introduction. https://docs.oracle.com/cd/E36784_01/html/E36848/zones.intro-1.html#scrolltoc, Accessed: 2018-02-24.

[39] Peter Domel. Mobile telescript agents and the web. In *Compcon'96.'Technologies for the Information Superhighway'Digest of Papers*, pages 52–57. IEEE, 1996.

[40] Fred Douglis. *Transparent process migration in the Sprite operating system*, volume 90. Citeseer, 1990.

[41] Pavel Emelyanov. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt, Apr 2013.

[42] Alex Fishman, Mike Rapoport, Evgeny Budilovsky, Izik Eidus, et al. Hvx: Virtualizing the cloud. In *HotCloud*, 2013.

[43] fork(2). Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/fork.2.html, Sept 2017.

[44] Dan Freedman. Experience building a process migration subsystem for unix. In *USENIX Winter*, volume 91, pages 349–356, 1991.

[45] Freezer Subsystem. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt, Jan 2016.

[46] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *ACM SIGOPS Operating Systems Review*, volume 33, pages 154–169. ACM, 1999.

[47] Tejun Heo. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/cgroup-v2.txt, Oct 2015.

[48] Himanshu Arora . An overview of Linux processes. https://www.ibm.com/developerworks/community/blogs/58e72888-6340-46ac-b488-d31aa4058e9c/entry/an_overview_of_linux_processes21, July 2012.

[49] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

[50] Dong Huang, Bingsheng He, and Chunyan Miao. A survey of resource management in multi-tier web applications. *IEEE Communications Surveys & Tutorials*, 16(3):1574–1590, 2014.

[51] Docker Inc. Storage Drivers - Docker Documentation. https://docs.docker.com/storage/storagedriver/, Apr 2017.

[52] Open Container Initiative. Oci image format specification. https://github.com/opencontainers/image-spec/blob/master/layer.md, Accessed: 2018-02-27.

[53] Internet Engineering Task Force (IETF) . RFC6298 - Computing TCP's Retransmission Timer. https://tools.ietf.org/html/rfc6298, Apr 2013.

[54] Poul-Henning Kamp. Freebsd - jails. https://wiki.freebsd.org/Jails, 2017. Accessed: 2018-02-24.

[55] keyrings(7). Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/keyrings.7.html, Sept 2017.

[56] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 25:1–25:14, Berkeley, CA, USA, 2007. USENIX Association.

[57] Ramon Lawrence. A survey of process migration mechanisms. *Dept. of CS, Univ. of Manitoba, May*, 29, 1998.

[58] Zhou Lei, Exiong Sun, Shengbo Chen, Jiang Wu, and Wenfeng Shen. A novel hybrid-copy algorithm for live migration of virtual machine. *Future Internet*, 9(3):37, 2017.

[59] libvirt. Bindings for other languages and integration API modules. `https://libvirt.org/bindings.html`, March 2017.

[60] libvirt. Terminology and goals. `https://libvirt.org/goals.html`, March 2017.

[61] libvirt. Connection URIs. `https://libvirt.org/uri.html`, March 2018.

[62] libvirt. Domain XML format. `https://libvirt.org/formatdomain.html`, March 2018.

[63] libvirt. Remote support. `https://libvirt.org/remote.html`, March 2018.

[64] libvirt Documentation. Contributor Guidelines. `https://libvirt.org/hacking.html`, May 2017.

[65] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the unix kernel. 1992.

[66] LWN. Ability to monitor task memory changes. `https://lwn.net/Articles/546966/`, Apr 2013.

[67] LXC-MONITOR(1). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man1/lxc-monitor.1.html`, Feb 2018.

[68] Memory Resource Controller. Linux Kernel Documentation. `https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt`, Feb 2018.

[69] Paul B Menage. Adding generic process containers to the linux kernel. In *Proceedings of the Linux Symposium*, volume 2, pages 45–57. Citeseer, 2007.

[70] Richard A. Meyer and Love H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[71] Barton Miller and David Presotto. Xos: an operating system for the x-tree architecture. *ACM SIGOPS Operating Systems Review*, 15(2):21–32, 1981.

[72] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[73] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[74] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393, March 2015.

[75] mount namespaces(7). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man7/mount_namespaces.7.html`, Sept 2017.

[76] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans

Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[77] namespaces(7). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man7/namespaces.7.html`. Feb 2018.

[78] network namespaces(7). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man7/network_namespaces.7.html`, Feb 2018.

[79] Network Working Group. Uniform Resource Identifiers. `http://www.ietf.org/rfc/rfc2396.txt`, August 1998.

[80] Mathias Noack. *Diploma Thesis Comparative Evaluation of Process Migration Algorithms*. PhD thesis, Citeseer, 2003.

[81] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, 2002.

[82] John K Ousterhout. Tcl/tk engineering manual. *Sun Microsystems*, 1994.

[83] Paul Jackson Paul Menage. Linux Kernel Documentation. `https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt`, Jan 2017.

[84] Paul Jackson Paul Menage. Linux Kernel Documentation. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`, Jan 2018.

[85] PID namespaces(7). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man7/pid_namespaces.7.html`, Nov 2017.

[86] pivot_root(2). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man2/pivot_root.2.html`, Sep 2017.

[87] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[88] POSIX message queues(7). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man7/mq_overview.7.html`, Sept 2017.

[89] Michael L Powell and Barton P Miller. *Process migration in DEMOS/MP*, volume 17. ACM, 1983.

[90] prctl(2). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man2/prctl.2.html`, Feb 2018.

[91] ptrace(2). Linux Programmer's Manual. `http://man7.org/linux/man-pages/man2/ptrace.2.html`, Sept 2017.

[92] Richard F Rashid and George G Robertson. *Accent: A communication oriented network operating system kernel*, volume 15. Acm, 1981.

[93] RedHat Inc. Linux Containers with libvirt-lxc. `https://access.redhat.com/articles/1365153`, March 2015.

[94] Michael Richmond and Michael Hitchens. A new process migration algorithm. *ACM SIGOPS Operating Systems Review*, 31(1):31–42, 1997.

[95] Christoph Rohland. Linux Kernel Documentation. `https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt`, March 2010.

[96] Ellard Thomas Roush. The freeze free algorithm for process migration. Technical report, Champaign, IL, USA, 1995.

[97] sigaction(2). Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/sigaction.2.html, Sept 2010.

[98] Panayotis A Skordos. Parallel simulation of subsonic fluid dynamics on a cluster of workstations. In *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*, pages 6–16. IEEE, 1995.

[99] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[100] Sundar Srinivasan. A Sneak-Peek into Linux Kernel - Chapter 2: Process Creation. http://sunnyeves.blogspot.co.uk/2010/09/sneak-peek-into-linux-kernel-chapter-2.html, Sept 2010.

[101] System V interprocess communication mechanisms(7). Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/svipc.7.html, March 2016.

[102] Jorge Nerin Terrehon Bowden. Linux Kernel Documentation. https://www.kernel.org/doc/Documentation/filesystems/proc.txt, Apr 2009.

[103] Katerina Koukiou (Αικατερίνη Κούκιου). Live lxc transport study and implementation save/restore for lxc to libvirt with the criu tool (Μελέτη ζωντανής μεταφοράς lxc και υλοποίηση save/restore για lxc στην libvirt με το εργαλείο criu). Master's thesis, National Technical University of Athens, 2016.

[104] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. *SIGOPS Oper. Syst. Rev.*, 19(5):2–12, December 1985.

[105] user namespaces(7). Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/user_namespaces.7.html, Feb 2018.

[106] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. In *ACM SIGOPS Operating Systems Review*, volume 17, pages 49–70. Acm, 1983.

[107] Roman Zajcew, Paul Roy, David L Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, et al. An osf/1 unix for massively parallel multicomputers. In *USENIX Winter*, pages 449–468, 1993.

[108] Edward Zayas. Attacking the process migration bottleneck. *ACM SIGOPS Operating Systems Review*, 21(5):13–24, 1987.

[109] Edward R Zayas. The use of copy-on-reference in a process migration system. 1987.