Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Μελέτη ζωντανής μεταφοράς LXC και υλοποίηση Save/Restore για LXC στην libvirt με το εργαλείο CRIU

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΙΚΑΤΕΡΙΝΗ ΚΟΥΚΙΟΥ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Μελέτη ζωντανής μεταφοράς LXC και υλοποίηση Save/Restore για LXC στην libvirt με το εργαλείο CRIU

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΙΚΑΤΕΡΙΝΗ ΚΟΥΚΙΟΥ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 12η Οκτωβρίου 2016.

....	....	....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επίκουρος Καθηγητής Ε.Μ.Π	Δημήτριος Τσουμάκος Επίκουρος Καθηγητής Ιόνιου Πανεπιστημίου

Αθήνα, Οκτώβριος 2016

....

Αικατερίνη Κούκιου

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Αικατερίνη Κούκιου, 2016.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Οι Linux Containers (LXC) είναι μία τεχνολογία εικονοποίησης σε επίπεδο λειτουργικού συστήματος και δίνει την δυνατότητα ταυτόχρονης εκτέλεσης πολλαπλών απομονωμένων Linux συστημάτων σε έναν host χρησιμοποιώντας έναν μοναδικό πυρήνα Linux. Το LXC χρησιμοποιεί τεχνολογίες του πυρήνα του Linux όπως Linux kernel cgroups, namespaces. Παρακάτω θα μελετήσουμε την δυνατότητα ζωντανής μεταφοράς (Live migration) Linux Containers. Πιο συγκεκριμένα μελετούμε τον τρόπο με τον οποίο μπορούμε να επιτύχουμε την μεταφορά ενός LXC από ένα φυσικό μηχάνημα σε ένα άλλο, χωρίς ο τελικός χρήστης να διαπιστώσει κάποια διακοπή στις εργασίες του. Χρησιμοποιούμε το εργαλείο CRIU για την καταγραφή της κατάστασης των Linux Containers (checkpoint) και επαναφορά (restore) από αυτά, και παρέχουμε μία υλοποίηση της έμβιας μεταφοράς αυτών. Στην αξιολόγηση του πρωτοτύπου μας παρατηρήσαμε αρκετά καλούς χρόνους ολοκλήρωσης μίας έμβιας μεταφοράς LXC και συγκρίναμε την επίδραση που έχουν διάφορα πρωτόκολλα μεταφοράς και αποθήκευσης δεδομένων καθώς και διάφοροι τρόποι δικτύωσης των υπολογιστών. Τέλος, παρουσιάζουμε μία υλοποίηση για αποθήκευση της κατάστασης ζωντανών Linux Containers και επαναφοράς αυτών από την αποθηκευμένη κατάσταση στο εργαλείο εικονικοποίησης libvirt.

Λέξεις κλειδιά

εικονικοποίηση, Linux Containers, LXC, cgroups, namespaces, live migration, CRIU, libvirt, checkpoint, restore, GbE, Infiniband

Abstract

Linux Containers (LXC) is an operating-system-level virtualization method for running multiple isolated Linux Systems (containers) on a control host using a single Linux kernel. LXC relies on the Linux kernel cgroups functionality and on kinds of namespace isolation functionality. This diploma thesis presents an implementation of live migration of a Linux Container from one physical node to another, which means transferring an instance of a running LXC without affecting the end-user experience and studies its phases. We use CRIU tool for generating snapshots of the running container instances and restoring from the saved state on another physical host. We also study the influence of different transfer protocols, filesystems used for storing the data produced by CRIU on the container's down-time and run tests on top of two different interconnects, namely GbE and Infiniband. Finally, we present our contribution to libvirt virtualization library, in order to support Save and Restore functionality for Linux Containers.

Key words

virtualization, Linux Containers, LXC, cgroups, namespaces, live migration, CRIU, libvirt, checkpoint, restore, GbE, Infiniband

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή Νεκτάριο Κοζύρη που μου έδωσε την ευκαιρία να ασχοληθώ με το πολύ ενδιαφέρον αντικείμενο της εικονικοποίησης.
Θα ήθελα ακόμα να ευχαριστήσω τον κ. Γεώργιο Γκούμα και όλα τα μέλη του εργαστηρίου για τη γενικότερη βοήθειά τους μέσω συζητήσεων που είχα μαζί τους.
Ακόμη, ευχαριστώ ιδιαίτερα τους επιβλέποντες διδακτορικούς φοιτητές Στέφανο Γεράγγελο και Ευάγγελο Αγγέλου, για την άρτια συνεργασία μας και για την διαρκή καθοδήγηση τους για την εκπόνηση αυτής της εργασίας.
Τέλος ευχαριστώ τους Michal Privoznik και Cedric Bosdonnat για τις συμβουλές τους όσον αφορά στο τεχνικό κομμάτι της παρούσας εργασίας.

Αικατερίνη Κούκιου,

Αθήνα, 12η Οκτωβρίου 2016

Contents

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Contents 11

1. Introduction 13
1.1 Motivation 13
1.2 Existing solutions 13
1.3 Thesis structure 15

2. Backround 17
2.1 History of Virtualization 17
2.2 Virtualization fields 17
2.3 Linux Containers 18
2.3.1 Linux Containers internals 18
2.3.2 Use cases for Linux Containers 21
2.4 Checkpoint/Restore projects 21
2.5 Basics and architecture of CRIU 22
2.6 Container migration 24
2.6.1 Naive live migration 25
2.6.2 Precopy live migration 26
2.6.3 Postcopy live migration 26
2.7 Block devices and File Systems 29
2.7.1 Storage 29
2.7.2 Distributed filesystems 29
2.7.3 Special filesystems & tmpfs 30
2.8 Contribution 30

3. Testing live migration 33
3.1 Live migration implementation 33
3.2 Test: ext4 vs tmpfs filesystem for storing the checkpoint data and rsync used for file transfer on 250MB dirtied memory 34
3.3 Test: tmpfs vs ext4 with NFS protocol used for serving the checkpoint data on 250MB dirtied memory 36
3.4 Test: GlusterFS on ext4 vs NFS on ext4 on 250MB dirtied memory 37

4. Add libvirt support for Save and Restore functionality for Linux Containers 41
4.1 Libvirt architecture & libvirt-lxc internals 41
4.2 Basic usage for libvirt Linux Containers 43
4.3 Save and Restore implementation for the LXC driver 44

5. Conclusion . . . 49
5.1 Concluding remarks . . . 49
5.2 Future work . . . 49

1. Εισαγωγή . . . 51
1.1 Κίνητρο . . . 51
1.2 Υπάρχουσες λύσεις . . . 52
1.3 Δομή εργασίας . . . 53

2. Υπόβαθρο . . . 55
2.1 Ιστορία της εικονικοποίησης . . . 55
2.2 Τομείς εικονικοποίησης . . . 55
2.3 Linux Containers . . . 56
2.3.1 Επισκόπηση των Linux Containers . . . 56
2.4 Βασική δομή του CRIU . . . 59
2.5 Έμβια και ψυχρή μεταφορά . . . 60
2.5.1 Αφελής υλοποίηση ζωντανής μεταφοράς LXC . . . 61
2.5.2 Υλοποίηση ζωντανής μεταφοράς LXC με precopy τεχνική αντιγραφής την μνήμης . . . 62
2.5.3 Υλοποίηση ζωντανής μεταφοράς LXC με postcopy τεχνική αντιγραφής την μνήμης . . . 63
2.6 Συνεισφορά . . . 65

3. Αξιολόγηση της ζωντανής μεταφοράς LXC . . . 67
3.1 Υλοποίηση ζωντανής μεταφοράς . . . 67
3.2 Test: ext4 έναντι tmpfs σύστημα αρχείων με την χρήση rsync για μεταφορά των δεδομένων σε εφαρμογή με 250MB μνήμη . . . 68
3.3 Test: tmpfs έναντι ext4 με NFS για την κοινή χρήση των δεδομένων από το CRIU σε εφαρμογή με 250MB μνήμη . . . 69
3.4 Test: GlusterFS με ext4 έναντι NFS με ext4 σε εφαρμογή με 250MB μνήμη . . . 72

4. Προσθήκη των λειτουργιών Checkpoint/Restore στην libvirt . . . 75
4.1 Η αρχιτεκτονική της libvirt και ο libvirt-lxc driver . . . 75
4.2 Χρήση των Linux Containers μέσω του εργαλείου virsh . . . 77
4.3 Υλοποίηση συναρτήσεων Save and Restore για τον libvirt-lxc driver . . . 78

5. Σύνοψη . . . 81
5.1 Συμπεράσματα . . . 81
5.2 Μελλοντικές Κατευθύνσεις . . . 81

Βιβλιογραφία . . . 83

Chapter 1

Introduction

1.1 Motivation

Operating System level virtualization is recently gaining steam due to its lightweight nature. High profile projects including Google Kubernetes [1], Canonical's LXD [2] and Redhat Openshift [3] are responding with innovations to make operating system level virtualization technologies like containers ready for production environments. In parallel to these big names, a number of companies are developing and releasing container tools, container focused operating systems, are offering hosting services and in some cases (eBay, Spotify, etc) powering their entire infrastructure via containers.

In this context, we realize the upcoming need to analyze Live Migration in the field of containers, meaning the process of saving the full state of a running container and restoring it on a different host in a way transparent to running applications and network connections. The reason for studying Live Migration on container based technologies is that the former is an extremely powerful tool, mostly for those interested in administration of data-centers and clusters. We will extensively discuss interesting use cases of combination of containerization and Live Migration later in Chapter 2.

This thesis is divided into two basic parts:

In the first part we are going to study Live Migration of Linux Containers between different physical hosts and compare transfer protocols and storage options for the data used in the migration process, regarding their affection on the container's down-time and therefore the end-user's experience. Moreover, we are going to compare our benchmarks when using two different computer interconnects, namely GbE and Infiniband. We will present our testing scenarios, pointing out the costs for all different phases of the Live Migration in Chapter 3. For this part, we will deal exclusively with Live Migration on Linux Containers that are generated with LXC [4] set of tools and templates. The underlying technology used is CRIU [5] and will delve into it's architecture in Chapter 2.

In the second part of this thesis, we are going to present our contribution to libvirt [6] virtualization toolkit, which provides an implementation that adds support for save and restore operations for Linux Containers. To support this functionality we are going to use once again CRIU tool.

1.2 Existing solutions

At the moment of writing this thesis, there are some operating system-level virtualization technologies that support Live Migration and this can be seen in Table 1.2. As one can notice, not all the systems are available as open source software and the information about some of them is pretty scarce.

OpenVZ [7] and Virtuozzo [8] projects, both support Live Migration by using CRIU as the underlying technology, which they develop as an open source tool. CRIU in fact helps various container based mechanisms to support migration by offering Checkpoint and Restore functionality, which means saving a container's state into files and restoring the container from files. We should mention here the P.Haul project [9]. This is a project on top of the CRIU tool, implementing Live Migration scenario. This project was first presented in Linux Plumpers Conference 2015 [10] and it still in building phase. For the time being, it supports Live Migration for OpenVZ containers, but there is ongoing work for LXC and Docker [11]. Lastly, LXD partially supports Linux Container Live Migration by using CRIU

Mechanism	Partition checkpointing and Live Migration	License
chroot	No	varies by operating system
Docker	No	Apache Licence 2.0
Linux-VServer	No	GNU GPLv2
Imctfy	No	Apache Licence 2.0
LXC	No	GNU GPLv2
LXD	Partial	Apache Licence 2.0
OpenVZ	Yes	GNU GPLv2
Virtuozzo	Yes	Proprietary
Solaris Containers	Partial	Proprietary
FreeBSD jail	Partial	BSD Licence
sysjail	No	BSD Licence
WPARs	Yes	Proprietary
HPUX	Yes	Proprietary
iCore Virtual Accounts	No	Proprietary
Sandboxie	No	Proprietary
Spoon	No	Proprietary
VMware ThinApp	No	Proprietary

Table 1.1: Partition checkpointing and Live Migration support in Operating-system-level-virtualization

as well via P.Haul project, but for the time the implementation is not acceptably fast.

1.3 Thesis structure

This thesis is organized as follows:

Chapter 2: We provide the necessary background for the concepts and entities that are being discussed throughout the thesis.

Chapter 3: We describe the steps we took in implementing the Live Migration for LXC containers, test it in various scenarios and analyze the results.

Chapter 4: We present the architecture of libvirt and describe our implementation that adds Save and Restore support for Linux Containers in libvirt.

Chapter 5: We provide some concluding remarks about this thesis and assess to what extent has it managed to achieve the goals that were set. Finally, we discuss some paths for future work that were out of scope of this thesis.

Chapter 2

Backround

2.1 History of Virtualization

The surging interest in cloud computing lies to the fact that it makes computing cheaper. The cost of licensing software and use of hardware is reduced dramatically, as the clients pay monthly/hourly fees to the cloud providers to use the cloud which includes hardware, software, and maintenance costs. This is what is typically described as a ”pay as you go” model. One aspect of cloud computing efficiency is the number of guests per physical machine. The more guests per machine results to lower costs for the vendors because more machines can run on the same physical infrastructure, which reduces the cost incurred by an individual machine in the cloud. In order to set up such resource sharing cloud infrastructures the main enabling technology is virtualization.

In computing, virtualization is a broad term that refers to the abstraction of computer resources. It includes hiding the physical characteristics of the computing resources, allowing to make a single physical resource appear to function as multiple virtual resources. It can as well make multiple physical resources appear as a single virtual resource. Besides the major concern virtualization tries to solve for cloud providers, that is leveraging of upfront infrastructure costs (e.g purchasing servers), it also comes as a solution to many other problems current data centers are facing; resource management, maintenance, underutilization of servers, disaster-recovery and failover protection.

Although the need of virtualization might appear something new as cloud computing its first seeds were sown as far back as the 1960s, when IBM evolved a method in order to logically divide the system resources provided by mainframe computers between different applications or users. Time sharing solutions where really important at that time as computer resources where not affordable even for large organizations. This is not the case in today’s data centers however, as the need of virtualization 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.

2.2 Virtualization fields

Today the term virtualization is widely applied to a number of concepts including:

- **Hardware Virtualization** which refers to the creation of a virtual machine that acts like a real computer with an operating system. The virtual machine is commonly referred as the guest, the actual machine is referred as the host and the software that facilitates the creation and management of the Virtual Machines is called a hypervisor.
  There are three basic different types of hardware virtualization regarding the level of the awareness of the guest operating system of the fact that is being virtualized:
  - **Full virtualization**: The guest operating system is unaware that it is in a virtualized environment and therefore hardware is virtualized 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 virtualization**: 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.
   – **Paravirtualization**: 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 some operations performed by the guest OS.

  Hardware-assisted virtualization is a way of improving overall efficiency of virtualization. It involves CPUs that provide support for virtualization in hardware, and other hardware components that help improve the performance of a guest environment.

- **Operating system-level virtualization** is a server virtualization method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one. Such instances, which are sometimes called containers (OpenVZ, LXC, Docker), virtualization engines (VEs) or jails (FreeBSD jail or chroot jail).
- **Application Virtualization** is software technology that encapsulates computer programs from the underlying operating system on which it is executed. The application behaves at run time like it is directly interfacing with the original operating system and all the resources managed by it, but can be isolated or sandboxed to varying degrees.
- **Network Virtualization** is the process of combining hardware and software network resources and network functionality into a single, software-based administrative entity, a virtual network.
- **Storage Virtualization**, the process of completely abstracting logical storage from physical storage.
- **Virtual memory**, which is giving an application program the impression that it has contiguous working memory, isolating it from the underlying physical memory implementation.

In next Section, we will delve into Operating system level virtualization, and namely Linux Containers.

2.3 Linux Containers

2.3.1 Linux Containers internals

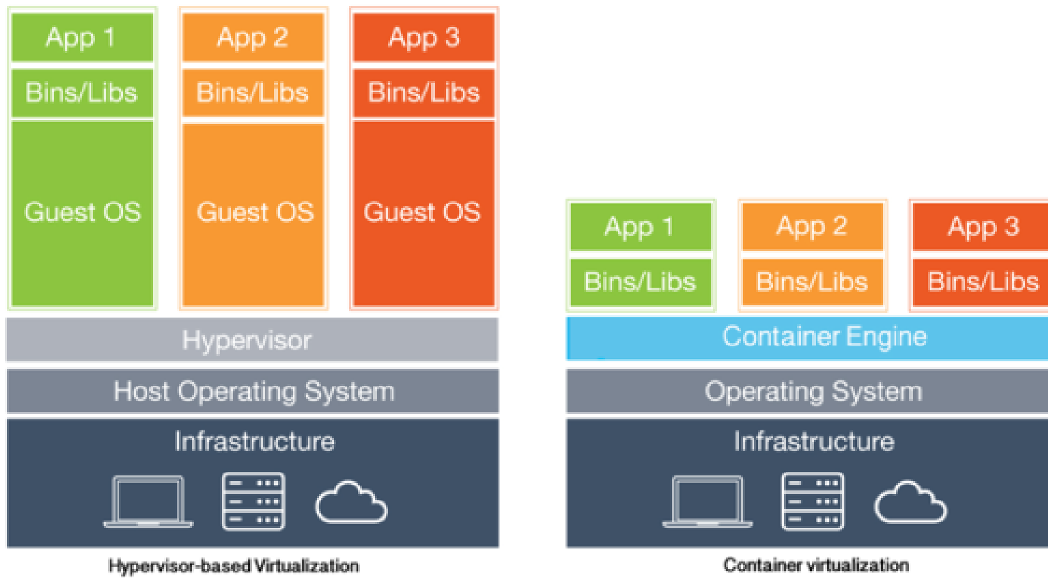
A *Linux Container* is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a single control host. The containers share the kernel with the host and provide an independent environment that has its own CPU, memory, block I/O, network and the resource control mechanism.

Several components are needed for Linux Containers to function correctly, most of them are provided by the Linux kernel. In particular, kernel namespaces which ensure process isolation and cgroups which are employed to control the system resources. SELinux is used to increase security, by assuring separation between the host and the guest OS and also between the individual guests.

What makes containers so popular nowadays, is what separates them from Virtual Machines. The main difference is that whereas containers provide a way to virtualize an OS in order for multiple workloads to run on a single OS instance, in VMs the hardware is being virtualized to run multiple OS instances. In practice, container oriented implementations share the kernel of the host operating system which is not the case in Virtual Machine technologies such as KVM. Therefore, it is usually possible to launch a much larger number of containers than virtual machines on the same hardware. Figure 2.1 visualizes the difference between hypervisor-based virtualization technologies such as KVM and Container Virtualization such as Linux Containers.

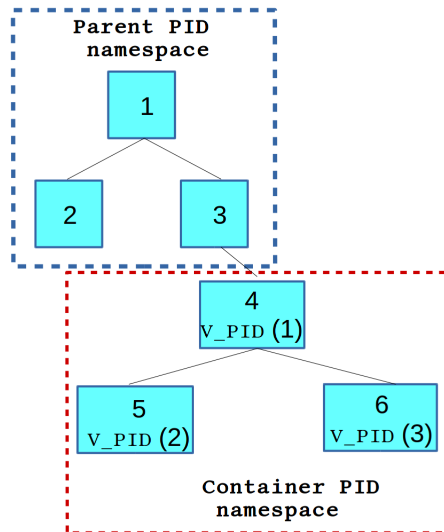
Linux Containers are typically comprised of the above components. [12] [13]

Figure 2.1: Full virtualization vs containerization technologies



- **Namespaces**
  - Utsname namespace: UTS namespaces provide isolation of two system identifiers: the hostname and the NIS domain name.
  - Ipc namespace: IPC namespaces isolate certain IPC resources, namely, System V IPC objects.
  - Pid namespace: PID namespaces isolate the process ID number space, meaning that processes in different PID namespaces can have the same PID as seen in Figure 2.2. PID namespaces allow containers to provide functionality such as suspending/resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs.
  - User namespace: User namespaces isolate security-related identifiers and attributes, in particular, user IDs and group IDs, the root directory, keys, and capabilities. A process's user and group IDs 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.
  - Mount namespace: Mount namespaces isolate the set of filesystem mount points, meaning that processes in different mount namespaces can have different views of the filesystem hierarchy
  - Network namespace: Network namespaces provide isolation of the system resources associated with networking
  - Multiple /dev/pts instances
- **Control Groups**
  - CPU (cpu, cpuset, cpuacct): This CPU system is often used to restrict a set of processes to a specific number of CPUs or amount of ”CPU time”.

Figure 2.2: Virtual vs Real PIDs



- Memory: The memory subsystem controls memory allocation and limits for a group of processes. Limit can be hard, soft and have ”pressure” applied.
- BLKIO: The BLKIO controls disk read or write speeds, operations per second, queue controls, wait times and other operations on an associated major and minor numbered block device. This subsystem does need to be explicitly enabled, but offers significant control over I/O when compared to other more traditional methods or filesystem specific controls. Support for this BLKIO subsystem is unfortunately weak within many container platforms, largely due to filesystem implementations.
- Devices: The devices cgroup subsystem is typically a whitelist, formatted for devices based on type (char vs block) and device major and minor numbers. A special ’all’ type applies to all device types, major and minor numbers and is typically used as a default deny before whitelisting explicit devices. Most containers will have access to commonly-used devices such as /dev/null and /dev/urandom . Network: The recently-added Network classifier cgroup can provide a method to tag network packets with a ”classid” value.
- Freezer: This subsystem allows a cgroup of tasks to be “frozen” by essentially sending a SIGSTOP signal and later ”unfrozen” or ”thawed” by sending a corresponding SIGCONT signal. This can be useful to pause a system or entire set of applications when there is no expected or intended use.

- **Root Capabilities**
  - They help to enforce namespaces in so-called ”privileged” containers by reducing the power of root, in some cases to no power at all.
- **Pivot_root**
  - is a syscall to ”pivot” into the new container environment, by changing the root file system.
- **Mandatory Access Control** (MAC) such as AppArmor and SELinux are not required for creating containers, but are often a key element to their security. MAC helps to enforce the security implemented by other container features.

Some of the above capabilities require that the kernel is configured with the corresponding option, namely use of user namespaces requires a kernel that is configured with the CONFIG_USER_NS option, selinux support requires CONFIG_SECURITY_SELINUX etc.

Having in mind the above, aka the namespaces, cgroups, etc there are three basic system calls that come in handy when it comes with creating Linux Containers. These are clone, setns and unshare. The clone system call, creates a new process, similar to fork system call but is more customizable since it allows the new process to isolate from the namespaces needed by providing the required flags (CLONE_NEWCGROUP for cgroup root directory, CLONE_NEWPID for new pid namespace, CLONE_NEWNS for new view for the mount points etc). Then setns system call allows the calling process to join an existing namespace, whereas unshare system call moves the calling process to a new namespace as implied by the given flags.

2.3.2 Use cases for Linux Containers

Above follow some usage scenarios of containers:

Run multiple versions of applications: With containers you can quickly launch new instances of applications and test them by just creating a new containerized environment from them, which is a matter of some seconds. That is especially helpful for cases when continuous integration systems make automated builds for new commits in order to check if the new patches have broken the build for some tests for some specific distribution. This job can be achieved with containers that can be created and then destroyed in some seconds.

Easy backups: With containers backing up is as simple as snapshotting your container. This feature is crucial for long running applications, such as bio-informatics programs, that can run for many months till they produce results. In such cases, a crash in the system will result in the loss of applications progress, which as mentioned can be quite large.

Multiple Linux distributions: LXC offers a wide choice of container OS templates. This allows you to to test applications in different Linux environments, and in cases where applications support a particular Linux variant you can deploy your application in that OS container.

Moving or upgrading servers: With containers moving your applications becomes a simple matter of snapshotting the container they are contained in and restoring them on a new server.

2.4 Checkpoint/Restore projects

”Checkpoint” is a technique that consists of saving a snapshot of the application’s state, which usually includes register set, address space, allocated resources etc. After saving an application’s state into files, with ”Restore” operation it is possible to re-construct the original running process from the saved image and resume it from exactly the interrupted point. Checkpoint and Restore operations have many usage scenarios which include load balancing a cluster, periodic state save of applications to avoid recomputation in case of a crash, applications behavior analysis by restoring on another machine, container live migration and many more. The two main approaches for providing Checkpoint/Restore operations are in userspace or in the operating system kernel.

Some projects that offer checkpoint functionality implementation are described above:

Fault Tolerance Interface (FTI) [14] is a library that aims to give computational scientists the means to perform fast and efficient multilevel checkpointing in large scale supercomputers.

Berkeley Lab Checkpoint/Restart (BLCR) [15] focuses on checkpointing parallel applications that communicate through MPI. BLCR is a hybrid kernel/user implementation of checkpoint/restart and does not require any changes made in the application code. Its work is broken down into 4 main areas:

Figure 2.3: Checkpoint/Restore projects through time

Sprite OS	BLCR	DMTCP	OPENVZ	Linux C/R	FTI CRIU
1984	2003	2007	2005	2008	2011

Checkpoint/Restart for Linux (CR), Checkpointable MPI Libraries, Resource Management Interface to Checkpoint/Restart and Development of Process Management Interfaces.

DMTCP [16] is a tool for transparently checkpointing the state of an arbitrary group of programs spread across many machines and connected by sockets. It does not modify the user's program or the operating system. Among the applications supported by DMTCP are Open MPI, Python, Perl, and many programming languages and shell scripting languages. With DMTCP, if you want to C/R some application you should launch one with DMTCP library (dynamically) linked from the very beginning. One other disadvantage of the DMTCP is the way it handles restoration of object with a predefined PID. The kernel provides no APIs for forking a process with the desired PID. To address that, DMTCP fools a process by intercepting the getpid() library call and providing fake PID value to the application. Such behavior is very dangerous, as application might see wrong files in the /proc filesystem if it will try to access one via its PID.

OpenVZ has in-kernel checkpoint/restore, sources can be found in kernel/cpt/.

Linux Checkpoint/Restart [17] was a project from around 2008 to around 2010 to implement checkpoint/restart of Linux processes. It's implemented in kernel space mostly and offers a userspace API which consist of two new systems calls for checkpoint and restart. These are `long checkpoint(pid, fd, flags, logfd)` and `long restart(pid, fd, flags, logfd)`.

Sprite OS is a Unix-like distributed operating system which supported process migration and allowed programs to be moved between machines at any time. The project was slowly shut down by 1994.

CRIU tool is a userspace checkpoint/restore tool which does not focus on specific group of applications and also supports containers (LXC, Docker, OpenVZ). CRIU is application transparent unlike previous userspace implementations. This will be extensively discussed in Section 2.5

Figure 2.3 visualizes the projects described above through time.

2.5 Basics and architecture of CRIU

CRIU is a project that implements Checkpoint/Restore functionality for Linux in userspace. Using this tool, it is possible to freeze a running application (or part of it) and checkpoint it to persistent storage as a collection of files. One can then use the files, restore the application from them and run it from the point it was frozen at. The distinctive feature of CRIU project is that it is mainly implemented in user space, rather than in the kernel and the fact that is application transparent. In this Chapter, we will present the design of CRIU, usage and inside details.

In contrast with previous checkpoint/restore projects, CRIU has some major advantages that make it the most fitting tool when it comes for container checkpointing:

- Supports containers (LXC, OpenVZ and Docker) by allowing namespaces, cgroups, ptys and technologies used in containers.

- It is transparent to the applications and can Checkpoint/Restore unmodified applications.
- There is no run time overhead after Checkpoint/Restore finishes.
- Retains the behavior of the C/R-ed programs.
- Network connections migration is supported.
- Uses standard OS kernel, but requires a relatively new one (>3.11)
- Can be used without preloading special libraries before app start.
- Can be used as non-root user.

To accomplish the checkpoint/restore functionality, CRIU intensively uses ptrace and mmap system calls. Combining these system calls CRIU implements the so called, parasite code injection, which is a way to run a system call on a foreign process by injecting some PIE code by mmap-ing on the foreign process and then running that code via ptrace. Moreover, CRIU gets a lot of process information from the proc pseudo-filesystem which provides an interface to kernel data structures. For example, it uses **/proc/<PID>/children** to acquire knowledge of the process tree, **/proc/<PID>/pagemap** which shows the mapping of each of the process's virtual pages into physical page frames or swap area. Futhermore, it uses **/proc/<PID>/fd** to get a list of process'es open file descriptors, and **/proc/<PID>/mounts** which lists the mount points of the process's own mount namespace.

To analyze this a bit further, the checkpointing procedure consists of the following three stages:

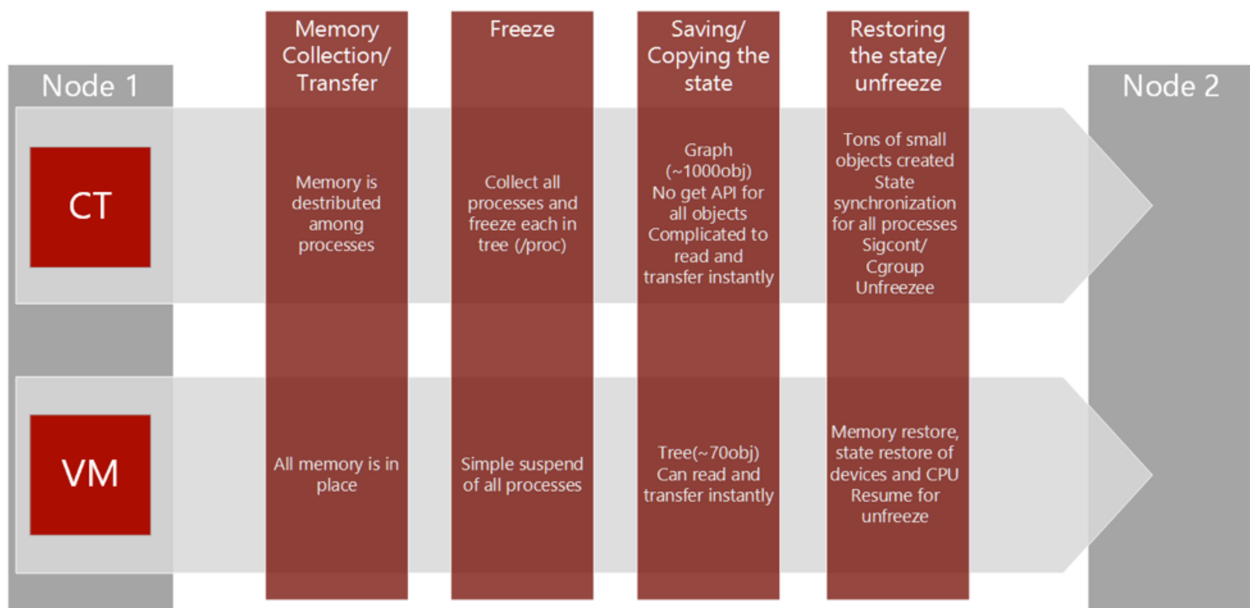
- Freeze processes - move processes to a previously known state and disable network so that processes's state is kept consistent during the checkpoint. To freeze the processes CRIU uses cgroup freezer or PTRACE_SEIZE. In order to freeze the network CRIU uses netfilter project or network namespaces.
- Dump the container - collect and save the state of a container into dump files. As mentioned above, for this purpose CRIU either parses proc files or it runs parasite injection code on processes to get credentials, memory contents or signals. The information gathered, is all stored in some files in protocol buffer format. Then CRIU dumps the pages, and memory pages are copied via vmsplice and splice system calls.
- Stop the container - kill all processes and unmount the container's filesystem.

The restore procedure consists of the following three stages:

- Restart the container - create a container with the same state as previously saved. Practically, what CRIU does here is forks children as same as in dump tree, forcing the virtual PIDs to be the same as the original PIDs, in the dump files. Then CRIU restores shared memory, cgroups, namespaces, GID/SID remap, etc. Technically the above are implemented between others, with calls of clone, setns, unshare, pivot_root system calls.
- Restart processes inside the container in the frozen state.
- Resume the processes' execution and enable network.

The most important requirement for CRIU to function is Checkpoint/Restore support from the kernel (CONFIG_CHECKPOINT_RESTORE). In addition, it will need Namespaces support, Inotify support for userspace etc. Most of the required setup options are used to expose an API that can be used to interrogate the kernel about process state information.

Figure 2.4: VM vs CT migration



2.6 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.

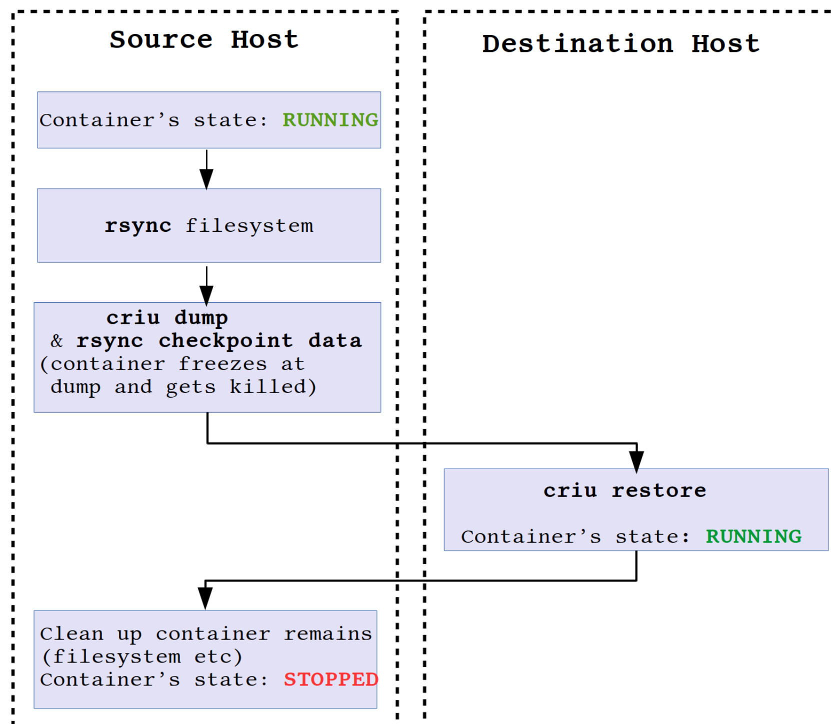
- Stop the container.
- Copy its filesystem to another server.
- Start it.

Cold migration is not really useful since if involves shutting down the container, something that can be unacceptable if for example the running applications will lose their progress if stopped. A solution to this is live migration. *Live migration* refers to the process of moving a running Linux Container between different physical machines without disconnecting the client. Memory, storage, and network connectivity of the Linux Container are transferred from the original guest machine to the destination guest machine.

Migrating container instances across distinct physical hosts is a useful tool for administrators of data centers and clusters: it allows a clean separation between hardware and software, and facilitates fault management, load balancing, and low-level system maintenance. For the migration process all basic work is done in two main operations: Checkpointing and Restoring. Checkpointing, as described in Section 2.4, refers to the procedure that the container's complete state is saved into disk files. For this purpose LXC tools, OpenVZ, Virtuozzo and Docker projects use CRIU. The reason why the above procedure is doable is because a container is an isolated entity, meaning that all the inter-process relations, such as parent-child relationships and inter-process communications, are within the container boundaries.

To this point, we should take a closer look so as to understand why container migration is technically a more difficult task to solve than VM migration. A visualization of the differences on the migration process for the two different mechanisms can be viewed in Figure 2.4. A point one could extract out of this, is that the VM's can be seen as a "black box", with all memory allocated inside this very box, whilst containers are represented by a number of processes with memory distributed among

Figure 2.5: Naive CT migration



them. Thus, memory collection for the container is a process of gathering the memory for multiple processes whereas for the VM's case all memory is in place and controllable by the hypervisor. For the same reason, freezing a container is harder, since it requires freezing a whole process tree, every process separately. That is not the case when suspending a VM however. When restoring the container, again we shall meet the same difficulties. In particular, we should recreate one by one all processes and restore their memory, keeping in mind all the synchronization issues. On the other hand, VM restoration is as simple as restoring VM's memory and devices and resuming CPU.

Bellow, follow three possible implementations for Linux Container live migration using CRIU tool, depending on how container's memory state is transferred from the source to the destination.

2.6.1 Naive live migration

As mentioned earlier in Section 2.5, with CRIU it is possible to dump one running container into a collection of files, which can be used later to restore the container from them. If these files from the checkpoint operation are transferred from the source node of the migration process to the destination node of the migration process and both nodes have access to the container's filesystem (for example via shared storage), this can be considered as the simplest example of live migration scenario using CRIU. The above process is shown in Figure 2.5. However, for processes that use a lot of memory and they dirty their memory fast, this implementation will result in very large downtimes for the container. The limiting factor in this case is the interconnect between the source and destination node of the migration, and namely the link speed that migration data will be transferred on.

The above functionality is already available for Linux Containers from LXC tools by using `lxc-checkpoint` and `lxc-checkpoint -r` commands since LXC version 1.1.0 and rsync or equivalent to transfer the checkpoint data. The naive migration for Linux Containers is also implemented within LXD hypervisor with `lxc move` command.

2.6.2 Precopy live migration

One existing solution to decrease process downtime during migration is pre-copy technique. The difference from the previous approach is that in precopy technique the memory is copied from source to destination node in multiple runs. During each run only memory pages which have changed since the last run are re-transferred to destination host. This technique can result in great improvements in cases where the dirty page rate is smaller than the link speed. CRIU supports this icremental pre-dump phase by using memory changes tracking. To understand how CRIU achieves this ”memory tracking” we shall notice two recent patches in the /proc pseudo filesystem [18]. These are:

- **/proc/$pid/clear_refs:** By writing 4 to this file we ask the kernel to clear the soft-dirty bit for all the pages associated with the process. This is used (in conjunction with /proc/[pid]/pagemap) by the check-point restore system to which pages of a process have been dirtied since the file /proc/[pid]/clear_refs was written to.
- **/proc/$pid/pagemap:** This file shows the mapping of each of the process’s virtual pages into physical page frames or swap area.

CRIU uses the two above files to support effective iterative checkpointing phase when it is part of a migration scenario. This functionality can be invoked in the following way:

```
1 for i in {1..MAX_ITER}; do
2     criu pre-dump --track-mem --prev-images-dir $checkpointdir/$((i-1))
3                               --images-dir $checkpointdir/$i [options]
4     rsync [OPTIONS] $checkpointdir/$i/ $dest_host:/$checkpointdir/$i/
5 done
6 criu dump --prev-images-dir $checkpointdir/$((MAX_ITER))
7           --images-dir $checkpointdir/$((MAX_ITER+1)) [options]
8 rsync [OPTIONS] $checkpointdir/$((MAX_ITER+1))/
9                 $dest_host:/$checkpointdir/$((MAX_ITER+1))/
```

By using `-prev-images-dir` option, the user asks CRIU to find images from previous dump or predump phases. If possible, CRIU will not dump memory pages that have not changed since that time. With `-track-mem option` we ask CRIU to reset memory changes tracker. If done, the next dump will have chances to successfully find not changed pages compared with –prev-images-dir. A good visualization of the above process can be seen in Figure 2.6.

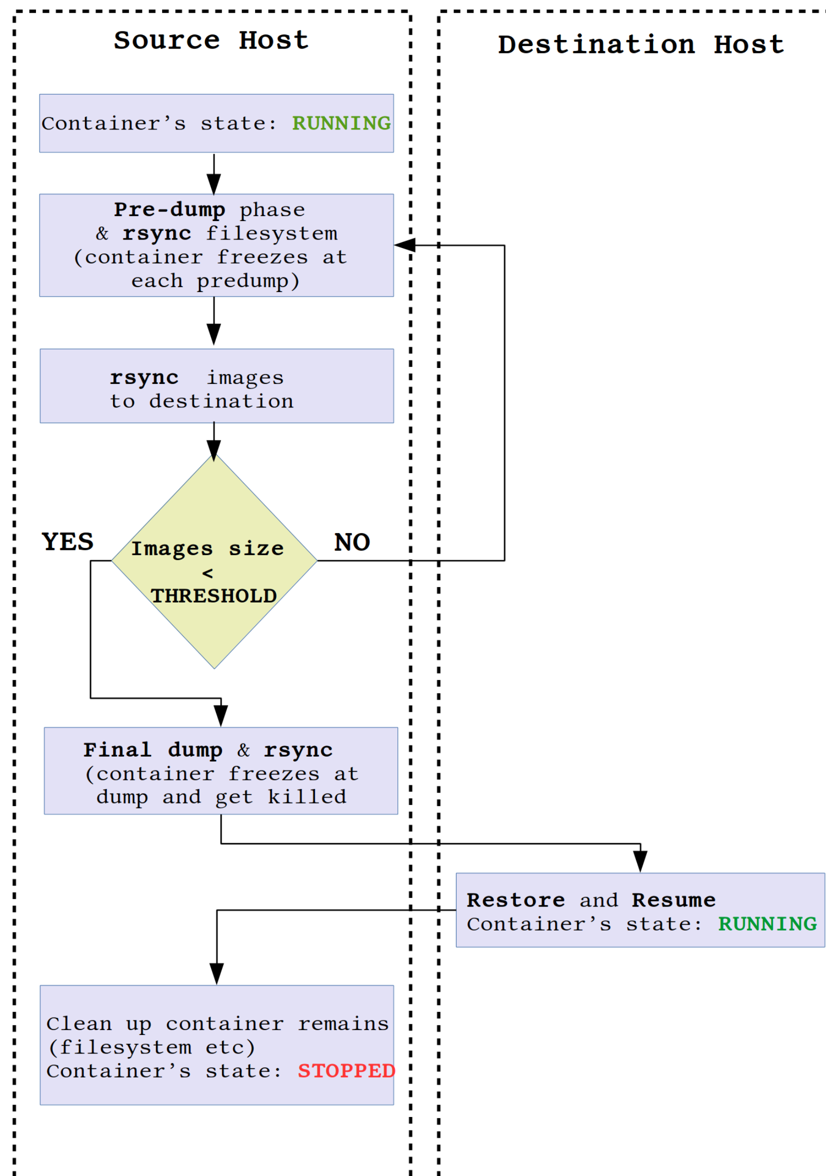
2.6.3 Postcopy live migration

One last migration technique CRIU also supports is postcopy. Postcopy handles memory transfers differently in the manner that it tries to restore the container on destination host without transferring it’s memory pages; when a page is needed that is not yet transferred those pages are copied on the fly while the container is running.

CRIU project has recently introduced this new feature in the development branch that enables the post-copy live migration scenario [19]. The main idea behind these patches is memory externalization and namely userfaultfd() syscall. The userfaultfd [20] syscall provides userland a protocol to control the userfaults in a transparent way. In particular, with userfaultfd a program can run with part (or all) of it’s memory residing on a remote node. Bellow follows an inexhaustive explanation on how this ”lazy pages” feature is used for a container live migration scenario:

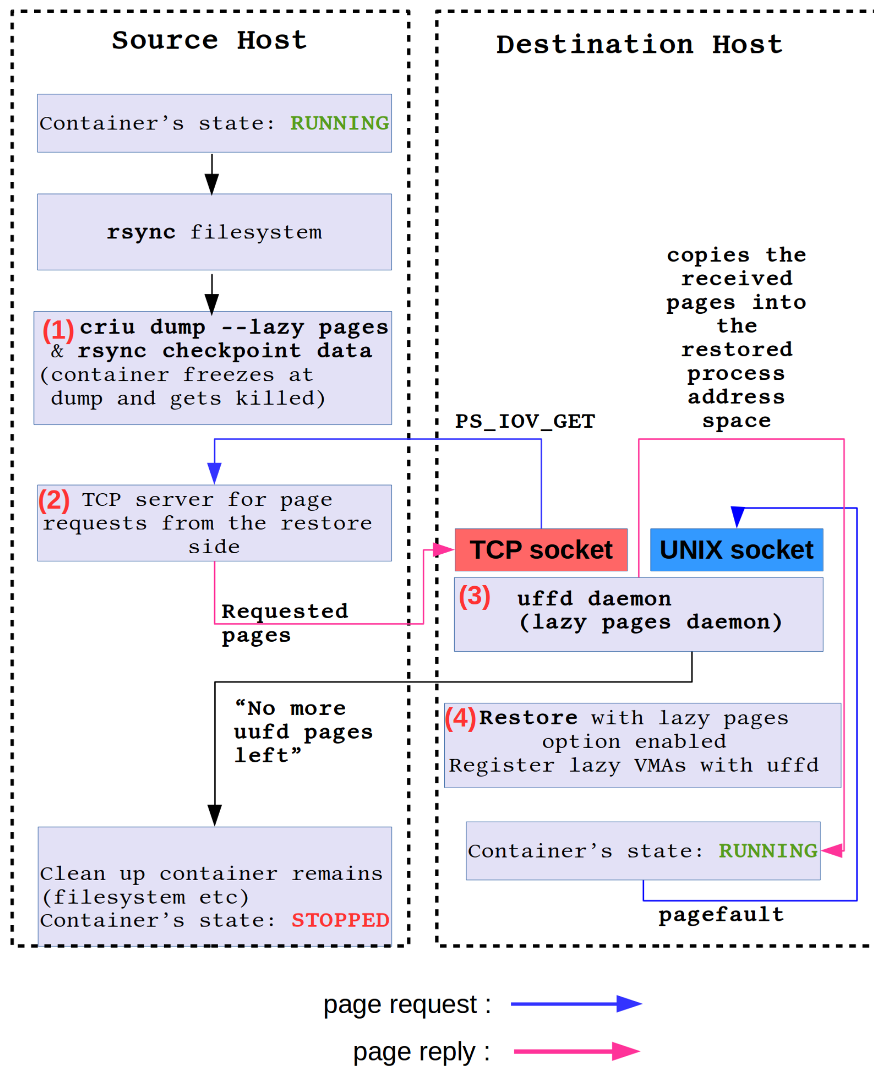
- Pages are divided into two categories: lazy pages and non-lazy pages.
- criu dump collects the process memory into pipes and transfers non lazy pages into images.

Figure 2.6: live migration with incremental dumping phase



- When dump phase finishes, criu dump starts a TCP server who will handle page requests from the restore side.
- The checkpoint directory containing non-lazy pages is transferred to the restore side
- On the restore side, a daemon (uffd daemon) is started and has two main jobs: first, to receive uffds from criu restore; second, to send requests to the dump side for the missing pages.
- When a page fault occurs on the restore side, the daemon sends a request for the page to the dump side.
- The dump side extracts the requested pages from the pipe and splices them into the TCP socket.
- Lastly, the daemon copies the received pages into the restored process address space
- As potentially not all memory pages are requested, the uffd daemon starts to transfer unrequested

Figure 2.7: postcopy live migration using lazy-pages



memory pages into the restored process so that the uffd daemon can shut down after a certain time.

Figure 2.7 visualizes the above post copy technique as implemented with the lazy pages of CRIU.

It is clear by now, that there are various approaches on how memory pages are transferred from source to destination host each with its own advantages and drawbacks which are however out of the scope of this thesis. Another factor that plays an important role when live migrating container instances is the underlying storage, meaning whether the files are kept on disk or ram and also the way data are made accessible on destination host. In the next Chapter we are going to study some file systems that we are going to use in our testing scenarios.

2.7 Block devices and File Systems

Most storage media and memory are organized in terms of blocks, although modern concepts like transparent huge pages [21] complicate things a bit. The devices, both the physical and the logical as exported by the operating system kernel, that are organized and addressed in terms of blocks are called ”block devices”. Filesystems lay on top of block devices and export a file hierarchy to the user space, in which data is organized as files and not longer just “meaningless” blocks. In other words, a file system is a method of organizing and retrieving files from a storage medium. File systems usually consist of files separated into groups called directories. Directories can contain files or additional directories. Filesystems can be categorized into some types, some of them include:

- ”Normal” filesystems (btrfs, ext2/3/4, XFS, ZFS, UFS, etc)
- Pseudo filesystems (procfs, sysfs, swap, etc)
- Special Filesystems (tmpfs, romfs, aufs, etc)
- Network and Cluster Filesystems (NFS, GlusterFS, SMB, etc)

2.7.1 Storage

There are basically two types of technologies that are used with storage systems, file level storage and block level storage. The former, are popular with network-attached storage (NAS) Systems whilst the latter are popular with storage area network (SAN) systems. A storage area network (SAN) is a network which provides access to consolidated, block level data storage. It does not provide file abstraction but block-level operations. However, file systems built on top of SANs do provide file level access, and are known as shared-disk file systems. These, generally add mechanisms for concurrency control avoiding corruption and unintended data loss even when multiple clients try to access the same files at the same time. Network-attached storage (NAS) is a file level computer data storage server connected to a computer network providing data access to a heterogeneous group of clients. To this end, the main difference between the two storage options is that SAN (Storage Area Network) provides only block-based storage and leaves file system concerns on the ”client” side whilst NAS appears to the client OS (operating system) as a file server. The differences between NAS, SAN, and DAS which means that digital storage is directly attached to the computer can be visualized in Figure 2.8. For the purpose of this thesis we are going to study the following list of network protocols used to serve NAS.

2.7.2 Distributed filesystems

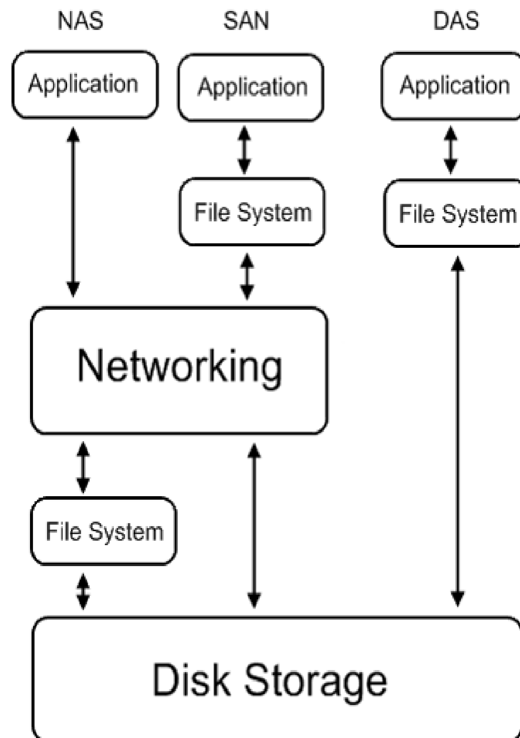
Distributed filesystems are clustered file systems, which means that are shared by being simultaneously mounted on multiple servers. They do not share block level access to the same storage but use a network protocol instead. These are commonly known as network file systems, even though they are not the only file systems that use the network to send data. Distributed file systems can restrict access to the file system depending on access lists or capabilities on both the servers and the clients, depending on how the protocol is designed.
For the purpose of this thesis, we are going to study two distributed file systems NFS and GlusterFS.

NFS, the network filesystem allows to access files on remote hosts in exactly the same way as a user would access local files. In order to provide NFS file sharing NFS implementations use a combination of kernel-level support on the client side and some daemons on the server side that provide the file data. The steps used in order to use a NFS are:

- A client requests to mount a directory from a remote host on a local directory using mount command.
- mount command will then try to connect to the mountd mount daemon via RPC.
- If the client is permitted to mount the directory it will be returned a file handle from the server.

After that when the client accesses the files over NFS, the kernel will place an RPC call to the NFS daemon (nfsd) on the server machine.

Figure 2.8: Storage options



GlusterFS, is a product of Red Hat that aggregates various storage servers over Ethernet or Infiniband RDMA interconnect into one large parallel network file system. GlusterFS has a client and server component. Servers are typically deployed as storage bricks, with each server running a glusterfsd daemon to export a local file system as a volume. The glusterfs client process, which connects to servers with a custom protocol over TCP/IP, InfiniBand or Sockets Direct Protocol, creates composite virtual volumes from multiple remote servers. One more important point is that GlusterFS is defined to be used in user space, i.e. File System in user space (FUSE). Most of the functionality of GlusterFS is implemented as translators, including file-based mirroring and replication, file-based striping, file-based load balancing, volume failover, scheduling and disk caching, storage quotas, and volume snapshots.

2.7.3 Special filesystems & tmpfs

tmpfs is a temporary filesystem that resides in memory and/or swap partitions, depending on how much it is filled. Mounting directories as tmpfs can speed up accesses to their files. Its important to mention that directories mounted as tmpfs, have their contents cleared upon reboots. Generally, I/O intensive tasks and programs that run frequent read/write operations can benefit from using a tmpfs folder. Later in Chapter 3 we are going to use tmpfs, to effectively speedup accesses to the migration data.

2.8 Contribution

In this work, we examine how the underlying filesystem used for storing the produced files from CRIU, the network technologies used for file transfer and lastly some configuration options that exist in CRIU tool, affect the live migration's downtime. It's important to realize the above before attempting any implementation, since in the live migration scenario even the slightest improvements

regarding the downtime can be really important. We study the naive migration scenario and present the results. However, our scripts support pre-copy migration as well. Then we present our contribution to libvirt virtualization library, in order to add support for Save and Restore operations for libvirt Linux Containers. Our implementation can be reused later to support live migration, but this is out of the scope of this thesis.

Chapter 3

Testing live migration

In this Chapter, we are going to present various tests on live migrating a Linux Container. We will examine the costs associated with live migration when the produced by CRIU files are made accessible on destination host through rsync transfer or NFS exported directory or even via GlusterFS replicated volume. Furthermore, we are going to study to what extent we benefit from avoiding disk storage by storing the files on RAM with the use of tmpfs. Lastly, we are going compare the migration costs when migrating between machines with the following underlying interconnect options: Infiniband and Gigabit Ethernet. The exact costs that we are going to study are the following:

- Frozen time of the container during dump phase.
- Time needed to transfer the image files from source host to destination host if using rsync.
- Frozen time of the container during restore phase.

Bellow we are going to present the different scenarios that we are going to study.

3.1 Live migration implementation

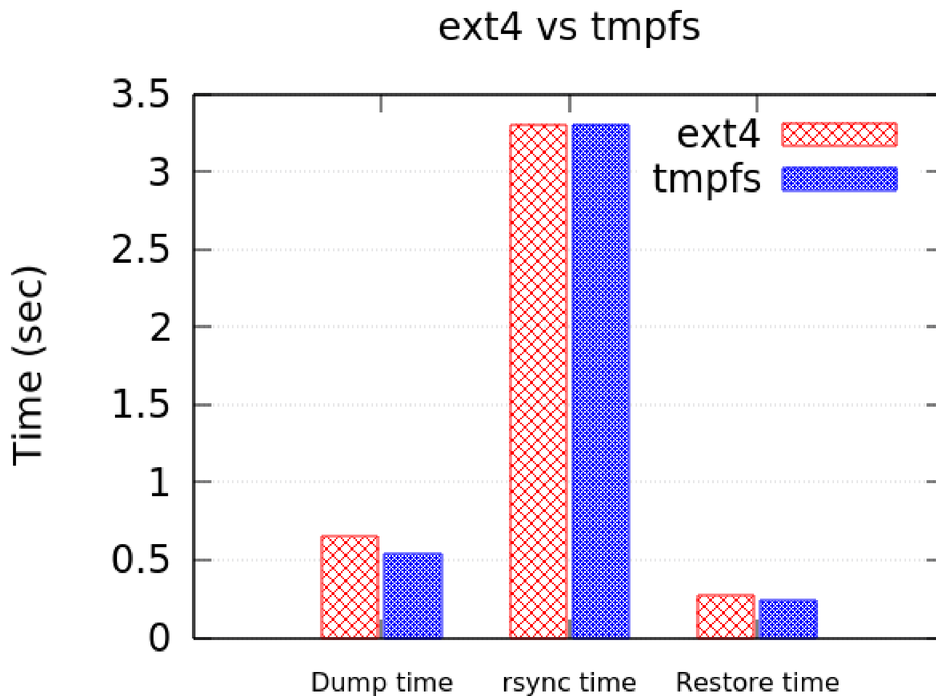
Our implementation, which can be found in https://github.com/KKoukiou/lxc-migration, is basically a wrapper around CRIU project, in order to implement LXC container naive and precopy live migration with some basic configuration. The code is written is bash and we call CRIU binaries from inside the script. From the user's perspective there is only a configuration file, called migration.conf and all operations can be controlled through that. The options that can be customized are the following:

- Container's name to migrate, as viewed by LXC tools
- Destination host's hostname or ip
- Maximum number of iterations of the dump process. There is a hardcoded memory threshold and if dumped files, are lower than this size the iterative dump phase stops and the final dump is performed.
- Storage and transfer options for the files, namely all pairs of disk or tmpfs and rsync or NFS for the transfer.

  The migration's configuration file, looks like the following:

```
1 cat migration.conf
2 #Checkpoint data can be stored and transferred from source to destination
3 #with the following options:
4 # 1: checkpoint on disk,rsync,restore on disk
5 # 2: checkpoint on disk. Destination host shares the checkpoint directory
6 over nfs, so no rsync is needed.
7 # 3: checkpoint on tmpfs, rsync restore on tmpfs
```

Figure 3.1: tmpfs vs ext4 with rsync on 250MB dirtied memory




```
 8  # 4: checkpoint on tmpfs. Destination host has nfs over tmpsf on the
 9  # 5: checkpoint directory shared with GlusterFS replicated Volume
10  # Options above should be one of the above {1,2,3,4,5}
11  # In cases 2 and 4 the nfs server is the destination host
12  Containers_Name: u1
13  Destination_Host: root@xenon8.cslab.ece.ntua.gr
14  Max_Iterations: 3
15  Checkpoint_Option: 3
```

3.2 Test: ext4 vs tmpfs filesystem for storing the checkpoint data and rsync used for file transfer on 250MB dirtied memory

In this test we will study the two different filesystem options for the image files produced by CRIU, namely ext4 and tmpfs filesystems. In particular, we will study the following two scenarios:

- The migration data are stored on ext4 filesystem on source host. We use rsync to transfer the CRIU files from source to destination host and we store the files on destination host on ext4 as well.
- The migration data are stored on tmpfs filesystem on source host. We use rsync to transfer the CRIU files from source to destination host and we store the files on destination host on tmpfs as well.

Figure 3.1 visualizes the live migration costs for the two above scenarios when using GbE as interconnect. For this test, the migration completes in only one iteration, which means there is no pre-dump phase.

It is clear that the greatest contributors to the downtime in this scenario are with cost order:

- The image transfer phase.
- The frozen time of the container during dump phase.
- The frozen time of the container during restore phase.

Regarding the frozen time of the container during dump phase, a solution would be to use incremental dumps and concurrent transfers after each pre-dump phase. What is important in this setup is that during the pre-dump phases the container gets frozen for quite less time than during normal dump phase. Also, in some scenarios, every dump after the first one could speed up. However, this does not apply always. In particular, if the dirty page rate is a lot higher than the transfer speed, an incremental dump would not result in smaller subsequent dumps. In this case, the pre-dump phase will not converge to the desired threshhold of pages, and thus we have to explicitly stop the pre-dump incremental phase, presumably not gaining any advantage for the final dump. On the other hand, if we know that the dirtying page rate of the applications running inside the container is small enough so that subsequent dump phases will find already monitored pages unchanged, there is good chance that an icremental dumping phase can help by splitting the unwanted single large dump into many smaller predumps where memory gets dumped and transfered without that the container is being frozen and one last quick final dump.

As far as the transfer time is concerned the most important factor here is the underlying interconnect (GbE, 10GbE, 40GbE, 100GbE, Infiniband) which affects the link rate. However, when we implemented this scenario over Infiniband interconnect we noticed no speed up for the rsync times at all. The problem here lies on the fact that rsync is limited by the CPU and not the link speed on our test machines. In particular, while running rsync on directories on top of tmpfs or ext4 we noticed the CPU usage staying stable at 100% which implied that the CPU frequency became on overhead for the rsync process.

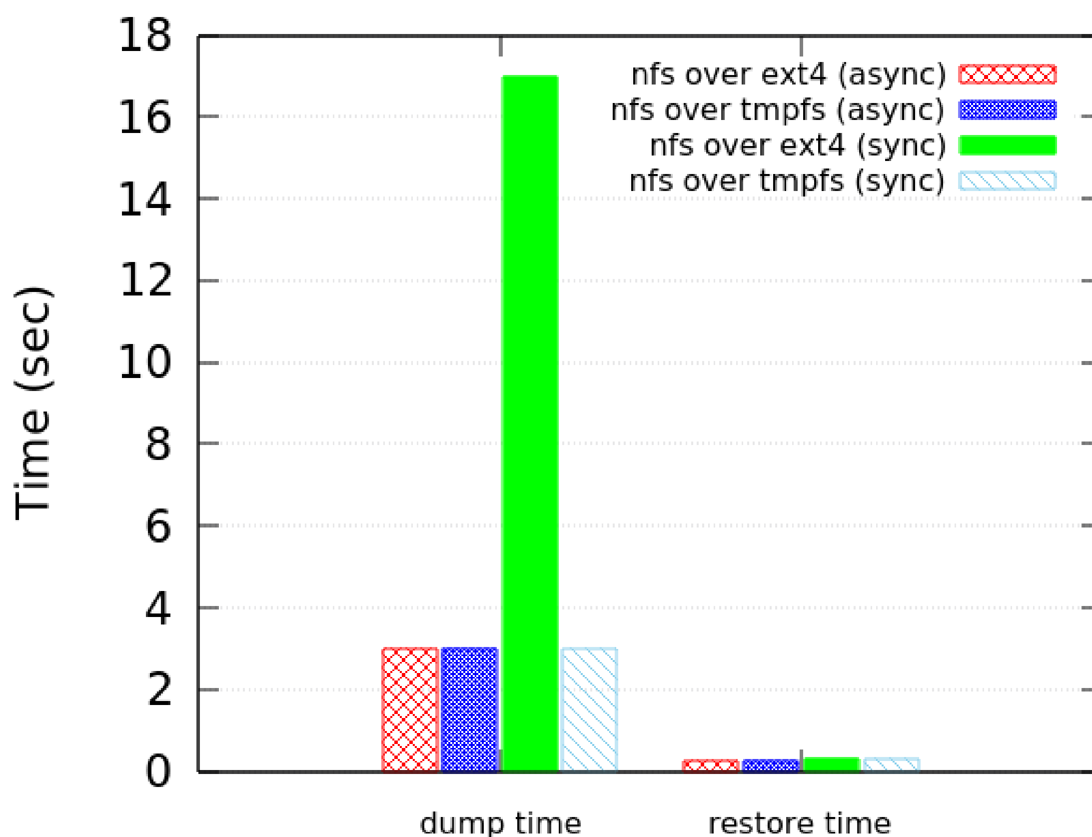
For the time there in no way to speed up the restore phase apart using a computer setup with higher CPU frequency.

To better understand the costs, we need to analyze the results a bit further. Regarding the frozen time during dump phase, we can notice that tmpfs has the same performance as ext4. It is important to understand that this observation is true for our testing setup because all the data produced by CRIU fit in page cache (RAM). So, since there are no data that get swapped out of disk cache, aka RAM during each test, the tmpfs gives no better performance. However, if other applications where using the RAM as well, then the data when stored on ext4 could have been forced out of RAM, and in this scenario we would definitely have the filesystem overhead due the lower write and read speed from disk. One other reason that an ext4 filesystem could have had more cost than a tmpfs, if ever the checkpoint data where about to get swapped out, is that this is a journaling filesystem. In such filesystems a special file called a journal is used to repair any inconsistencies caused due to improper shutdown of a system. This journal checksumming sacrifices speed by adding extra cost, but makes recovery possible in case of system crash.

As far as the transfer time is concerned, we can notice that it's the same for both filesystem scenarios. The explanation to this lies in the disk cache. When CRIU creates the IMG files which in our example have a total size of 250MB these data stay in RAM until it becomes full and start swapping out. That is however not going to happen in our example, since the whole data-set fits within page cache. So rsync basically reads the files of the checkpoint directory that were just created on the source host and currently lie on RAM, transfers the data and writes them on destination host. Once again the writes are executed on files that get cached on RAM and never swapped out during the test so we see no additional overhead.

The previous explanation applies for the frozen time during restore, meaning that in both filesystems all the data are read from RAM which never becomes full.

Figure 3.2: tmpfs vs ext4 with NFS on 250MB dirtied memory on GbE



3.3 Test: tmpfs vs ext4 with NFS protocol used for serving the checkpoint data on 250MB dirtied memory

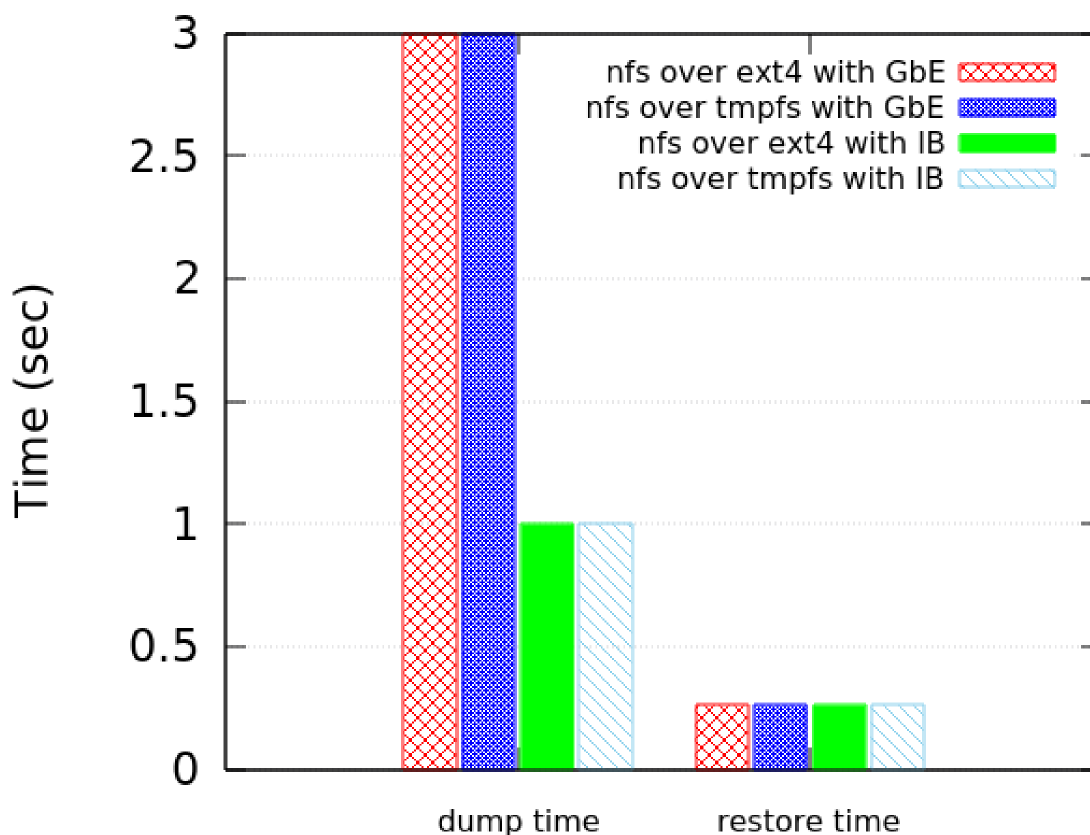
In this scenario, the destination host acts as a NFS server and exports the directory on which the checkpoint data are going to be saved. The source host, who is the client, requests access to the checkpoint directory by issuing the mount command. After that, every write from the source host on the checkpoint directory, will update the dump files for destination host, thus making all checkpoint data finally available on destination. For these tests, we use NFS v3 with UDP transfer protocol.

Figure 3.2 visualizes the costs associated with live migration for the NFS exported directory when stored on memory or on disk with GbE network interconnect. Moreover, we study here the synchronous and asynchronous behavior in NFS server. The option sync (synchronous) in the client context means that all changes to the according filesystem are immediately committed to the server. The respective write operations are being waited for. For directories stored on mechanical drives that means a huge slow down since the system has to move the disk heads to the right position. In contrast, with async the system buffers the write operation and optimizes the actual writes; meanwhile, instead of being blocked the process in userland continues to run and the files will only be transmitted usually when the file is closed. sync option on the server context (the default) causes the server to only reply to say the data was written when the storage backend actually informs that the data was written. Option async in the server context gets the server to merely respond as if the file was written on the server irrespective of if it actually has written it.

For this test we use async mode for the client and study both modes for the server and the dump phase completes in only one iteration, which means there is no pre-dump phase.

As expected, the sync mode for the server causes huge delays during dump with ext4 underlying

Figure 3.3: tmpfs vs ext4 with NFS on 250MB dirtied memory and GbE vs IB



filesystem, since with this option the NFS server follows the NFS protocol and replies to requests only when any changes made by that request have been committed to stable storage (e.g. disc drive). Especially when we are dealing with mechanical disk as the underlying storage this can have tremendous impact, as after each write on disk the process get blocked to wait on confirmation. The same does not apply with sync option when data are stored on top of tmpfs filesystem since the responses from the server are very fast. So, for the rest three scenarios, the dump phase has exactly the same cost, and that is because data use page cache and nothing get's swapped out.

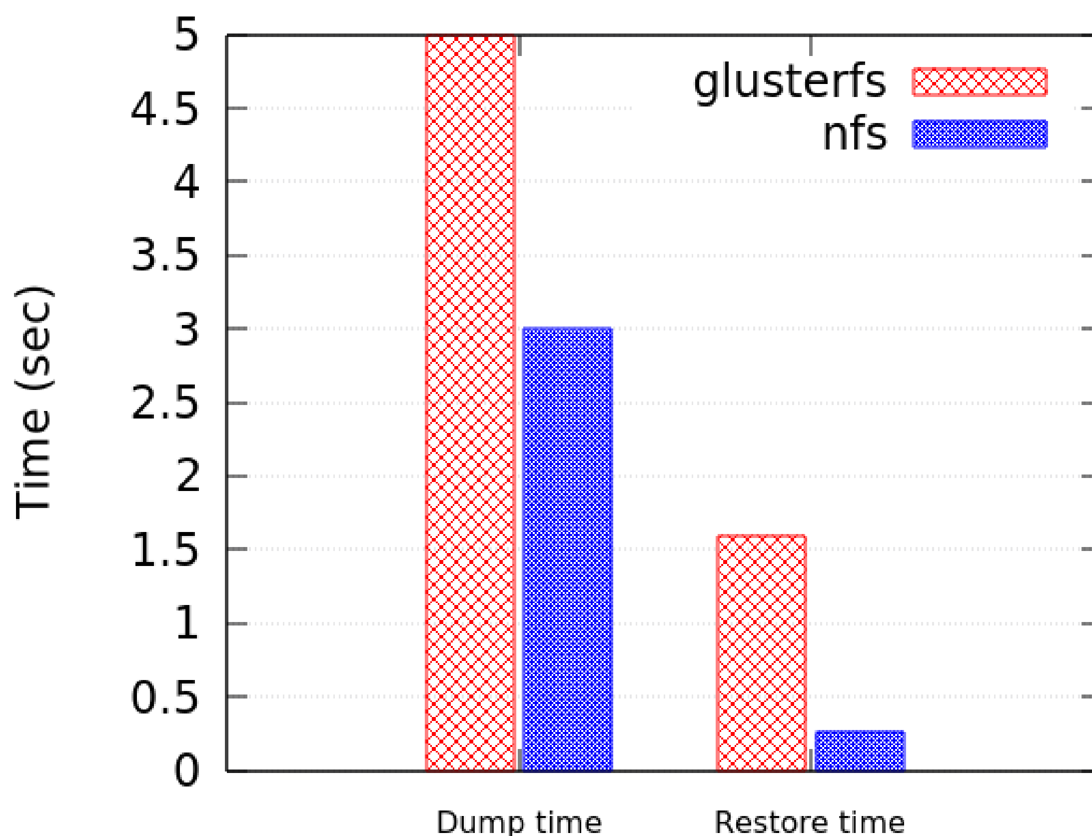
During restore, since no writes are made the sync/async modes do not affect the migration's cost and since data are cached on the destination's server RAM, tmpfs and ext4 again make no difference.

The second experiment here was to test the affection of the underlying interconnect. Figure 3.3 visualizes the expected better performance of the Infiniband over the Gigabit Ethernet interconnect during the dump phase. We present at this point only the async mode for the NFS export, since this is the one that is faster. What's going on underneath is that when the files are written on the mounted NFS exported directory during dump phase data transfer occurs and since the IPoIB setup is a lot faster we mention the speed up when using the aforementioned setup. During restore phase, since the files are already stored on the destination host, there is no affection.

3.4 Test: GlusterFS on ext4 vs NFS on ext4 on 250MB dirtied memory

As mentioned earlier, GlusterFS is a software only file system. Here data is stored on native file systems like ext4, xfs etc. For the purpose of this test, the underlying filesystem on which data are stored on ext4. In addition, we configured the two nodes to share replicated directory for the checkpoint data and all the communication happens on top of TCP.

Figure 3.4: GlusterFS vs NFS on 250MB dirtied memory

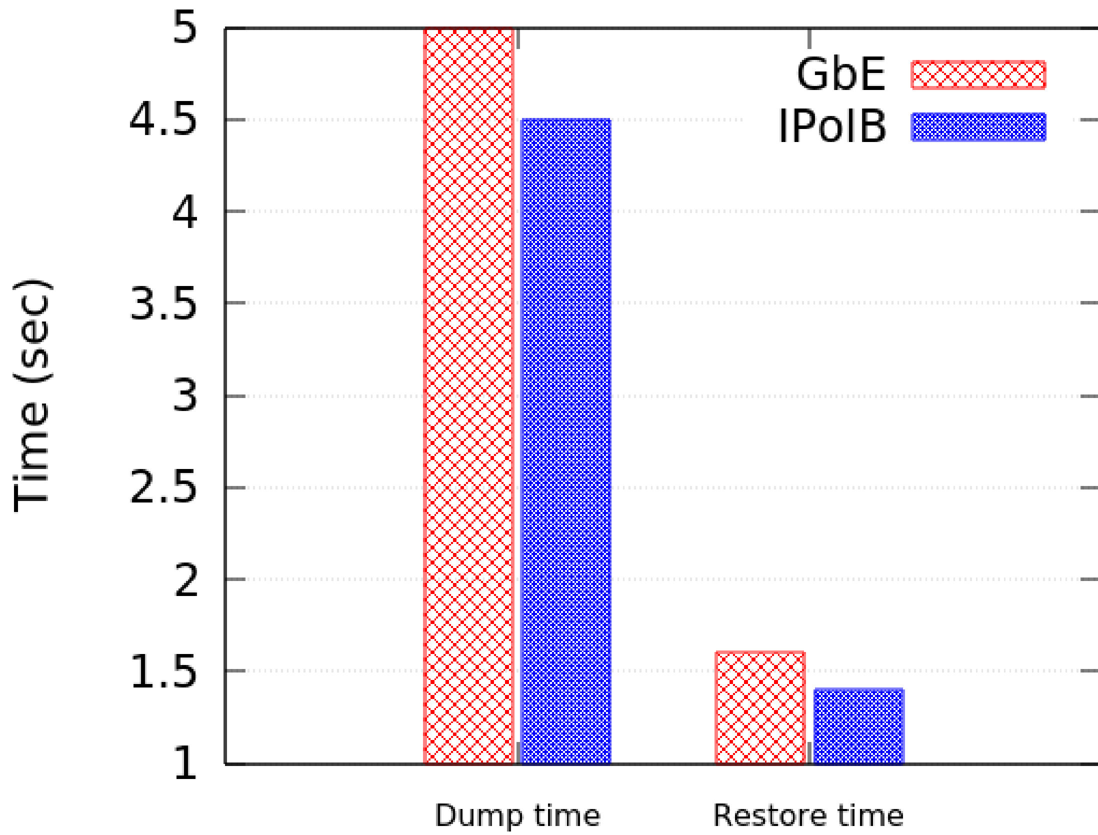


At first sight, we notice that NFS outperforms GlusterFS both during dump phase and restore phase. Here, we will propose some possible explanation behind this behavior. Firstly, our NFS export uses UDP protocol for the file transfer whilst GlusterFS uses TCP. There are some vital differences between the two protocols that result is the significant deviations, when transferring the files. The main difference is that TCP performs handshakes, these are SYN, SYN-ACK, ACK whilst UDP is a connectionless protocol, so there are no handshakes at all. Moreover, TCP rearranges data packets in the order specified fact which adds extra cost, while UDP has no inherent order as all packets are independent of each other. To sum up, TCP is heavy-weight as it requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control. On the other hand, UDP is lightweight. There is no ordering of messages, no tracking connections, etc. All the above result is greater frozen time during dump, as the data transfer costs more in our GlusterFS configuration.

Regarding the frozen time during restore, we shall notice here again that the GlusterFS configuration results in very poor read performance. One possible explanation behind this is how GlusterFS locates the data. GlusterFS spreads load using a distribute hash translation (DHT) of file names to it's subvolumes and locates data algorithmically using a Hashing Algorithm. Knowing nothing but the path name and file name, any storage system node and any client requiring read or write access to a file in a Gluster storage cluster performs a mathematical operation that calculates the file location. All the above result in the poorer read performance of the GlusterFS compared to NFS export during restore. To sum this up, it is important to make clear that we do not conclude that GlusterFS generally has poorer performance than NFS when used to share data between two nodes, but for our testing scenario it is not suitable.

Although, by now we understand that GlusterFS is not the best option for our usage scenario, we shall study the behavior when GlusterFS related file transfer occurs on top of IPoIB. The results of

Figure 3.5: GbE vs IB interconnect and GlusterFS on 250MB dirtied memory



this test are shown in Figure 3.5. It is clear that there is indeed some speed up both on dump and restore phases, since communication between the nodes is a lot faster. However, the overhead here is not only the link speed but also the userspace operations that need to be done in terms of the FUSE filesystem.

Chapter 4

Add libvirt support for Save and Restore functionality for Linux Containers

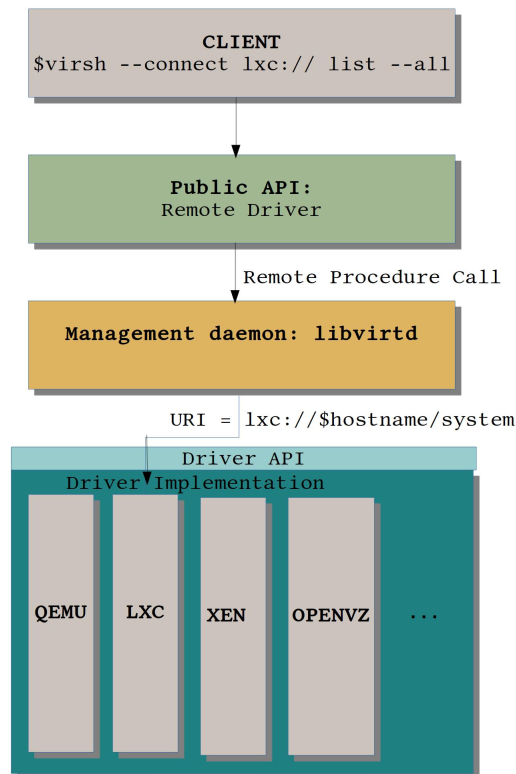
Libvirt is a collection of software mostly implemented in C with main goal to manage the virtualization capabilities of recent versions of Linux and other OSes. It provides a common way for managing multiple different virtualization providers and hypervisors and generally virtualization technologies like virtual networking and virtual storage. Libvirt consists of three main components: an API library, a daemon (libvirtd), and a command line interface called virsh. Libvirt's utilities offer various operations to be performed on the guest OSes such as start, stop, pause, save, restore, and migrate, within the limits of the hypervisor for those operations. Libvirt also offers node resource operations which are needed for the management and provisioning of guests such as interface setup, firewall rules, storage management and general provisioning APIs. Lastly, it is also possible to use libvirt to control domain instances on remote hosts, as long as there is a running libvirt daemon.

4.1 Libvirt architecture & libvirt-lxc internals

At this point is is clear that libvirt offers a generic way of handling different virtualization technologies. For handling the multiple different virtualization providers and consequently the hypervisor specific calls libvirt architecture introduces the idea of Drivers. For all supported hypervisors there is a referring driver, resulting in the existence of the following internal drivers:

- LXC - Linux Containers
- OpenVZ
- QEMU
- Test - Used for testing
- UML - User Mode Linux
- VirtualBox
- VMware ESX
- VMware Workstation/Player
- Xen
- Microsoft Hyper-V
- IBM PowerVM (phyp)
- Virtuozzo
- Bhyve - The BSD Hypervisor

Figure 4.1: libvirt drivers

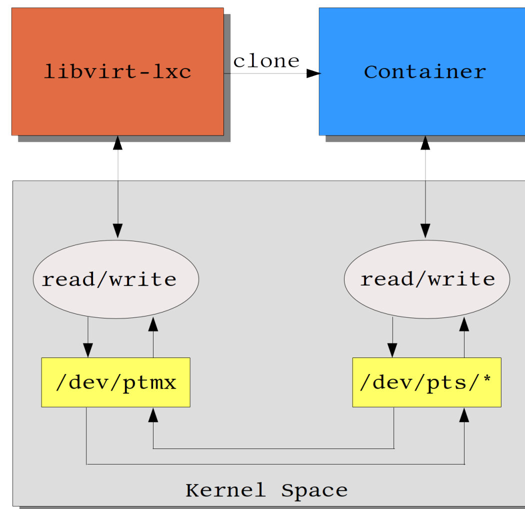


Each driver is responsible to load the driver's specific functions, in order to be used by the public API of libvirt. When using an API call, one should choose one of the available drivers to perform that call on. For the majority of supported hypervisors, access to libvirt drivers is performed as follows. First the client application connects to a so called remote driver, providing URI to identify the specific driver they want to connect at. Then the remote driver routes the calls to libvirtd daemon via an RPC call. libvirtd 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. Figure 4.1 visualizes the above-described process as if a client application issues a command via virsh CLI.

For the purpose of this thesis we are going to deal exclusively with the LXC driver, that is the one who manages Linux Containers. It is important to mention here that the libvirt LXC driver, sometimes also called libvirt-lxc has no dependency on the LXC userspace tools, which we used in order to create the Linux Containers for our testing scenarios earlier in Chapter 3. Instead, libvirt LXC driver directly utilizes the required kernel features to build the containers.

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". libvirt-lxc's functionality is to set up the container and provide console access to the container. In order to achieve isolation of libvirt Linux Containers heavily utilize the functionality of clone() system call with the required flags. By default, the flags used are CLONE_NEWPID, CLONE_NEWNS, CLONE_NEWUTS, CLONE_NEWUSER, CLONE_NEWIPC and CLONE_NEWNET which specify which namespaces the calling process does not share with the container. So, libvirt-lxc process from the hosts OS, calls clone() system call in order to create a new isolated process, from which the container will be generated from. As for the container to have stdin, stdout, stderr libvirt-lxc utilizes the use of pseudoterminal interfaces(PTYs). A pseudoterminal is a pair of virtual character devices that provide a bidirectional communication channel. One end of the

Figure 4.2: Pseudoterminals & libvirt-lxc



channel is called the master; the other end is called the slave. The slave end of the pseudoterminal provides an interface that behaves exactly like a classical terminal. The container is supplied in with the slave end of the PTY, whilst the container's controller process is supplied with the master end, and in this way the container is provided with 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, as shown in Figure 4.2. When the container setup is complete, libvirt-lxc executes container's init process, which is provided by the user in the XML configuration file, where the container's information are stored. libvirt-lxc controller who is the parent process of the container, receives a SIGCHLD signal in case the container terminates, crashes, or generally changes state and invokes the appropriate operations. 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 filesystem than the process who has cloned him, pivot_root system call is used.

4.2 Basic usage for libvirt Linux Containers

Libvirt Linux Containers can be managed among other tools with virsh CLI, by issuing `virsh --connect lxc:/// $COMMAND`. Bellow, follows an overview of some basic usage of libvirt LXC containers using virsh. As previously mentioned, information about the configuration of the container are stored in a XML file, which basically looks like the following.

Function	Command
Load containers configuration into libvirt (for persistent domains)	`virsh -c lxc:/// define guest.xml`
Remove containers configuration from libvirt (for persistent domains)	`virsh -c lxc:/// undefine guest`
Starts the container	`virsh -c lxc:/// start guest`
Stops the container	`virsh -c lxc:/// shutdown guest`
Forcefully stops the container	`virsh -c lxc:/// destroy guest.xml`
Creates container (used for non persistent domains)	`virsh -c lxc:/// create guest.xml`
Starts the container	`virsh -c lxc:/// start guest`
Connect to container's console	`virsh -c lxc:/// console guest`

Table 4.1: virsh command for LXC

```
1  cat sh\_container.xml
2  <domain type='lxc'>
3    <name>vm1</name>
4    <memory>500000</memory>
5    <os>
6      <type>exe</type>
7      <init>/bin/sh</init>
8    </os>
9    <vcpu>1</vcpu>
10   <clock offset='utc'/>
11   <on\_poweroff>destroy</on\_poweroff>
12   <on\_reboot>restart</on\_reboot>
13   <on\_crash>destroy</on\_crash>
14   <devices>
15     <emulator>/usr/libexec/libvirt\_lxc</emulator>
16     <interface type='network'>
17       <source network='default'/>
18     </interface>
19     <console type='pty' />
20   </devices>
21 </domain>
```

In Table 4.2 one can see the basic command used on LXC containers.

4.3 Save and Restore implementation for the LXC driver

Bellow we will outline the steps we followed in order to add Checkpoint and Restore functionality [22] in lxc driver by mimicking the functionality of the `lxc-checkpoint` and `lxc-checkpoint -r` commands that exist in LXC userspace tools. As with the LXC tools approach, the underlying technology we use is CRIU, which we integrate into the sources of libvirt, thus allowing libvirt to call CRIU binaries. With our implementation we are going to support two extra commands for the libvirt-lxc driver:

- `virsh -c lxc:/// save guest checkpoint_directory`
- `virsh -c lxc:/// restore checkpoint_directory`

The first, saves the present state of the "guest" container into a collection of files in the directory specified. The second, restarts a container which was previously saved with `virsh save` command into a collection of files.

In terms of technical implementation, we have expanded LXC driver source namely src/lxc/lxc_driver.{c,

h} files, by adding two new functions: lxcDomainSave and lxcDomainRestore.

When `virsh save` command is ran the libvirtd daemon polls lxc driver which invokes the lxcDomainSave lxc driver specific function. Firstly, it performs some locking operations, since container's state is about to be changed from RUNNING to STOPPED and thus during the state change queries from other threads can result in unexpected behavior. In order to perform effective locking we have introduces job control mechanism to the lxc driver [1] [3] [2] [4] [5] [6] [7] . Then some further checks are performed, like checking if the container is running so that it can be saved or if the container is persistent so that the appropriate structures are reorganized after the state change. Furthermore, the XML configuration file of the container should be saved with the checkpoint files, so that it can be used to when restoring. Then, lxcCriuSave function is called, that is a newly defined function in src/lxc/lxc_criu.{c, h} which is a wrapper for CRIU binaries in order to dump the running container. In this file `criu dump` is called with the appropriate options some of which are:

- –enable-external-sharing, –enable-external-masters: These flags enable external shared or slave mounts to be resolved automatically on restore when CRIU is asked so.
- –ext-mount-map" "/dev/console:console, –ext-mount-map", "/dev/tty1:tty1: These two are external mount and should be handled explicitly so as to restore them. The same should be done, for higher numbered ttys as well.
- –external tty[rdev:dev]: That is because the container's master end of the tty is outside of what is dumped. If this option is not passed, CRIU will complain about a slave end without corresponding master end.
- –tcp-established to handle TCP connections, –file-locks to dump file locks, –link-remap to allow one to link unlinked files etc.

lxcDomainRestore function, which is defined inside src/lxc/lxc_driver{c, h} and is the first function invoked after `virsh restore` command is ran, comes with a more complicated implementation. To fully understand how we implement the restore operation, we should take a closer look into what we checkpointed earlier with `virsh save` command. The files CRIU produces store the information for the container, meaning the subtree starting from the container's init process among with all other processes inside the container. libvirt-lxc process, that is the controller of the container as we described earlier is not dumped, and therefore on the restore scenario it should be reconstructed. After libvirt-lxc process is setup as needed, we will be ready to call our CRIU wrappers, to restore the previously saved container as a child of the libvirt-lxc process. Similarly as before, before invoking any CRIU operation that is about to change the container's state from STOPPED to RUNNING we have to make sure to perform the needed locking operations in order to ensure consistency. In order to support Restore operation by making as few changes as possible, we have changed virLXCProccessStart which is located in src/lxc/lxc_process{c, h} into a more generic one. This function used to start a libvirt-lxc instance with the referring container. Our modified version extends the previous one as takes one extra argument: a directory file descriptor and in case of container restore which is used to point to the directory to find the CRIU image files. The basic setup jobs the controller code needs to perform remain as before, with some slight modifications. For example, the controller still needs to create the /dev/ptmx instances in order to control container's console from libvirt-lxc, but does not need to perform namespaces setup, cgroup setup, and setup of the pseudottys inside the container since all these information are stored in the dumped files and will be handled by CRIU when restoring. In other words, in the case of restoring, libvirt-lxc process performs a subset of the operations compared to a normal initialization of a container. After all required operations are completed, lxcCriuRestore function as implemented in src/lxc/lxc_criu.{c, h} is invoked and restores the container from the saved state as a child of the libvirt-lxc process. This function is again a CRIU wrapper, for the `criu restore` command.

Figure 4.3 visualizes the `virsh save` operation as implemented internally and Figure 4.4 visualized the `virsh restore` operation.

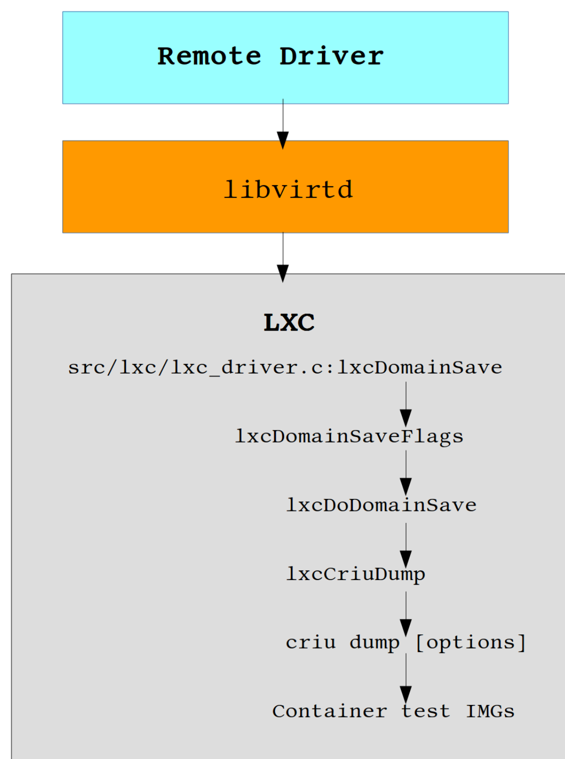


Figure 4.3: virsh save implementation

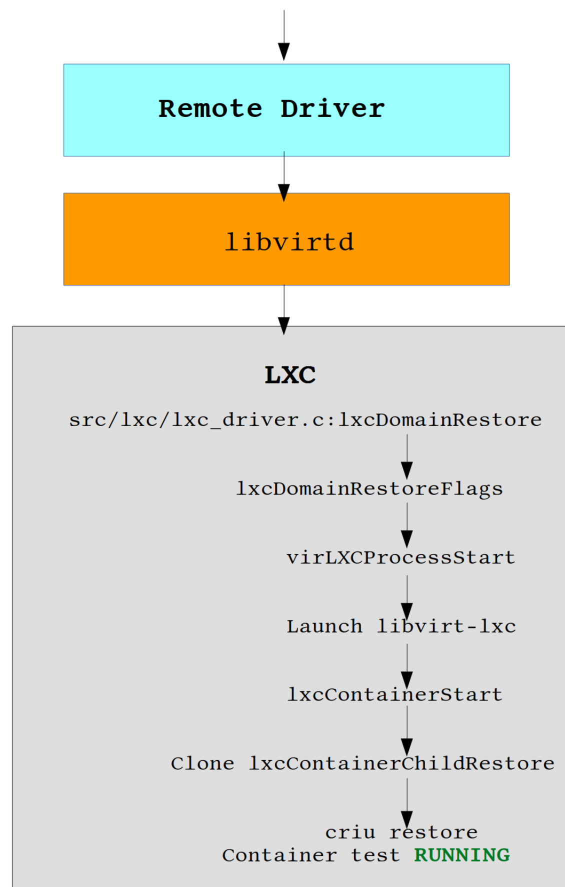


Figure 4.4: virsh restore implementation

Chapter 5

Conclusion

5.1 Concluding remarks

Previous work on Linux Containers Checkpoint/Restart, aka LXC checkpoint command, is not implemented in a way that will support effective Live Migration. On the other hand, other implementations for Live Migration of libvirt Linux Containers [23] are not considering the crucial aspect of merging their implementation with libvirt project and focus more on the research part. With this thesis, we tried to address the main difficulties when live migrating a Linux Container and implemented Save and Restore support for libvirt-lxc driver.

5.2 Future work

After sending the above described implementation's patches to libvirt list, we realized that there are some basic design decisions that need to be reconsidered in order that our Checkpoint/Restore implementation can be extended for the Live Migration scenario. Currently, our implementation of Checkpoint saves the snapshot to disk, into a collection of files. Then during Restore operation these files are read from the directory they are stored at, and the container get reconstructed. However, in the case of Live Migration, saving files to Disk and reading from Disk during restore will result in non optimal downtimes as seen earlier in Chapter 3. On the other hand we cannot rely on solutions like NFS exported directories to enhance downtimes, since we need to keep the dependencies and requirements as few as possible. Therefore we realized the need to patch CRIU in order to be able to stream the produced files instead of storing them on directories. In this way, the lxc-driver can then decide either to store the files on disk on the simple checkpoint/restore scenario or use the internal streaming protocols libvirt has, to send the data to destination host, where the container will be restored. In this way, there is not overhead for storing the files if not necessary which will surely result in lower downtimes, but instead CRIU can stream the data via sockets where needed. The latter, is a work in progress [24], and will enable us to merge the patches for the Save/Restore implementation in libvirt project. We are also working on a project to support Live Migration of Linux Container in libvirt [25] again having in mind to merge the patches with libvirt project. To end this, there are some more things to be done to enhance our Save/Restore implementation for libvirt. Some of these are:

- Use CRIU in order Save/Restore network connections.
- Currently our implementation support only one pty per container. Support for higher numbered ptys has a straightforward implementation as well.
- Support dumping of containers with mounts of type efivarfs.

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Οι τεχνολογίες εικονικοποίησης σε επίπεδο λειτουργικού συστήματος τα τελευταία χρόνια χρησιμοποιούνται και αναπτύσσονται σε μεγάλο βαθμό λόγω τον υψηλών τους επιδόσεων. Η Google με το Kubernetes [1], η Canonical με το LXD [2] και η Redhat με το Openshift [3] είναι μερικά από τα παραδείγματα μεγάλων εταιριών που καταπιάνονται με τεχνολογίες εικονικοποίησης επιπέδου λειτουργικού συστήματος αναπτύσσοντάς τις κατάλληλα ώστε να προωθηθούν στην παραγωγή. Παράλληλα με τα μεγάλα αυτά ονόματα στον χώρο της πληροφορικής, μεγάλος αριθμός από εταιρίες αναπτύσσουν εργαλεία και λειτουργικά συστήματα επικεντρωμένα στις τεχνολογίες εικονικοποίησης επιπέδου λειτουργικού συστήματος αλλά και προσφέρουν τις υπηρεσίες τους χρησιμοποιώντας τις παραπάνω τεχνολογίες. Για παράδειγμα, το eBay και το Spotify προσφέρουν τις υπηρεσίες τους καθημερινά χρησιμοποιώντας τις προαναφερθέντες τεχνολογίες.

Το πιο ευρέως χρησιμοποιούμενο προϊόν της εικονικοποίησης επιπέδου λειτουργικού συστήματος είναι οι containers. Οι containers παρέχουν ένα ελαφρύ εικονικό περιβάλλον που απομονώνει ένα σύνολο διεργασιών και περιορίζει τους πόρους που αυτές χρησιμοποιούν. Η απομόνωση αυτή εξασφαλίζει ότι οι διεργασίες μέσα στον container δεν μπορούν να δουν τις διεργασίες έξω από αυτόν. Ανάλογα με την λειτουργία που προσφέρουν, οι containers μπορούν να διαχωριστούν σε OS containers [7] [4] και application containers [11]. Στην πρώτη περίπτωση οι containers προσομοιώνουν ένα λειτουργικό σύστημα ενώ στην δεύτερη περίπτωση απλά δημιουργούν ένα απομονωμένο περιβάλλον για να τρέξει μία εφαρμογή, χωρίς να τους περιορίζει ωστόσο κάτι σε αυτή την χρήση.

Σε αυτό το πλαίσιο, αναγνωρίζουμε την επερχόμενη ανάγκη να αναλύσουμε την διαδικασία της ζωντανής μεταφοράς για τους OS containers. Με τον όρο "ζωντανή μεταφορά" εννοούμε την διαδικασία αποθήκευσης της κατάστασης ζωντανών στιγμιοτύπων containers και επανεκτέλεση αυτών από το σημείο στο οποίο αποθηκεύτηκαν σε άλλο φυσικό μηχάνημα, με τρόπο τέτοιο ώστε η διαδικασία της μεταφοράς να μην είναι αντιληπτή ούτε από τις εφαρμογές που πιθανών τρέχουν μέσα στον container ούτε από τις συνδέσεις δικτύου. Η παραπάνω λειτουργία, αποτελεί εξαιρετικά σημαντικό εργαλείο και χρησιμεύει κατά κύριο λόγο για την διαχείριση cluster. Αναλυτικά θα μελετήσουμε τα οφέλη που προσφέρει η ζωντανή μεταφορά των containers στο Κεφάλαιο 2.

Η παρούσα διπλωματική εργασία, χωρίζεται σε δύο βασικά μέρη:

Στο πρώτο μέρος θα μελετήσουμε την ζωντανή μεταφορά Linux Containers μεταξύ δύο φυσικών μηχανημάτων και θα μετρήσουμε τον χρόνο κατά τον οποίον οι containers είναι "παγωμένοι", δηλαδή μη αποκρίσιμοι συγκρίνοντας την χρησιμοποίηση διαφόρων τρόπων αποθήκευσης και μεταφοράς των δεδομένων. Θα παρουσιάσουμε για τα διάφορα σενάρια όπου ελέγξαμε την μεταφορά τα αντίστοιχα αποτελέσματα και θα τα αναλύσουμε παρέχοντας ανάλογες προτάσεις βελτίωσης. Στην συνέχεια θα παρουσιάσουμε την συνεισφορά μας στο εργαλείο εικονικοποίησης libvirt, ώστε να παρέχει δύο νέες λειτουργίες: την αποθήκευση της κατάστασης ενός ζωντανού container σε αρχεία και την επαναφορά του από αυτά που θα ονομάζουμε από εδώ και στο εξής Save και Restore. Και στις δύο παραπάνω υλοποιήσεις θα χρησιμοποιήσουμε το εργαλείο CRIU το οποίο μας παρέχει τρόπο να αποθηκεύουμε την κατάσταση των containers σε αρχεία που θα χρησιμοποιούνται για την επαναφορά των containers σε δεύτερο χρόνο. Το CRIU θα το παρουσιάσουμε εκτενώς στο Κεφάλαιο 2.

Μηχανισμός	Αποθήκευση κατάστασης και ζωντανή μεταφορά	License
chroot	No	varies by operating system
Docker	No	Apache Licence 2.0
Linux-VServer	No	GNU GPLv2
lmctfy	No	Apache Licence 2.0
LXC	No	GNU GPLv2
LXD	Partial	Apache Licence 2.0
OpenVZ	Yes	GNU GPLv2
Virtuozzo	Yes	Proprietary
Solaris Containers	Partial	Proprietary
FreeBSD jail	Partial	BSD Licence
sysjail	No	BSD Licence
WPARs	Yes	Proprietary
HPUX	Yes	Proprietary
iCore Virtual Accounts	No	Proprietary
Sandboxie	No	Proprietary
Spoon	No	Proprietary
VMware ThinApp	No	Proprietary

Table 1.1: Υποστήριξη αποθήκευσης κατάστασης και ζωντανή μεταφορά σε τεχνολογίες εικονικοποίησης επιπέδου λειτουργικού συστήματος

1.2 Υπάρχουσες λύσεις

Την στιγμή συγγραφής αυτής της διπλωματικής εργασίας, υπάρχουν κάποιες υλοποιήσεις εικονικοποίησης επιπέδου λειτουργικού συστήματος που υποστηρίζουν ζωντανή μεταφορά. Ο Πίνακας 1.2 δείχνει την κατάσταση για τις πιο γνωστές υλοποιήσεις. Είναι εύκολο να παρατηρήσει κανείς ότι πολλές από τις τεχνολογίες που αναφέρονται δεν έχουν υλοποιήσεις ανοιχτού κώδικα (open source) και για αυτό τον λόγο πληροφορίες σχετικά με την υλοποίηση της ζωντανής μεταφοράς στις αντίστοιχες τεχνολογίες είναι δυσεύρετες.

Από τις παραπάνω υλοποιήσεις, ορισμένες σχετίζονται στενά με την υλοποίηση ζωντανής μεταφοράς Linux Containers που θα μελετήσουμε σε αυτή την εργασία. Συγκεκριμένα, το OpenVZ και το Virtuozzo [8] υποστηρίζουν ζωντανή μεταφορά χρησιμοποιώντας το εργαλείο CRIU [5]. Επιπλέον, το project P.Haul [9] είναι ένα project πάνω από το CRIU, που έχει σκοπό να υλοποιήσει την ζωντανή μεταφορά για διάφορες τεχνολογίες container όπως OpenVZ, Docker και LXC. Ωστόσο, είναι ακόμα σε αρχικό στάδιο. Τέλος, το project LXD [2], που λειτουργεί σαν hypervisor για το LXC είναι επίσης σε στάδιο ανάπτυξης και πρόκειται να υποστηρίξει ζωντανή μεταφορά Linux Containers μέσω του P.Haul.

1.3 Δομή εργασίας

Η παρούσα διπλωματική εργασία είναι οργανωμένη ως εξής:

Κεφάλαιο 2: Παρέχουμε το αναγκαίο υπόβαθρο για την κατανόηση των εννοιών και τεχνολογιών που χρησιμοποιούνται στην παρούσα εργασία.

Κεφάλαιο 3: Περιγράφουμε τα βήματα που ακολουθήσαμε για την υλοποίηση της ζωντανής μεταφοράς των Linux Containers και τρέχουμε πειράματα μελετώντας τα αποτελέσματα.

Κεφάλαιο 4: Παρουσιάζουμε την αρχιτεκτονική της libvirt και περιγράφουμε την υλοποίηση μας, που παρέχει υποστήριξη για τις λειτουργίες Save και Restore για Linux Containers.

Κεφάλαιο 5: Παρέχουμε κάποια τελικά συμπεράσματα για την παρούσα εργασία και αξιολογούμε κατά πόσο κατάφερε να επιτύχει τους στόχους που τέθηκαν στην αρχή. Τελικά, συζητούμε για επιπλέον δουλειά που απομένει να γίνει σαν συνέχεια αυτής της εργασίας.

Κεφάλαιο 2

Υπόβαθρο

2.1 Ιστορία της εικονικοποίησης

Τα τελευταία χρόνια όλο και πιο πολύ διογκώνεται το ενδιαφέρον των επιχειρήσεων για την υιοθέτηση του Cloud. Πρόσβαση από παντού, μείωση κόστους και ασφάλεια είναι κάποια από τα πιο βασικά πλεονεκτήματα της τεχνολογίας αυτής. Από την μεριά των παροχέων των υπηρεσιών Cloud ένας από τους παράγοντες αξιολόγησης της αποδοτικότητας είναι ο αριθμός των guest μηχανημάτων, δηλαδή φιλοξενούμενων μηχανημάτων ανά φυσικό μηχάνημα. Όσο περισσότεροι guests υπάρχουν ανά φυσικό μηχάνημα τόσο πιο μικρό είναι το κόστος για τους παρόχους αφού πολλά φιλοξενούμενα εικονικά μηχανήματα μοιράζονται τις ίδιες φυσικές υποδομές και άρα το κόστος για το κάθε εικονικό μηχάνημα μειώνεται. Η τεχνολογία εικονικοποίησης είναι επομένως αυτή που επιτρέπει την ταυτόχρονη χρησιμοποίηση των φυσικών πόρων.

Η εικονικοποίηση είναι ένας γενικός όρος των υπολογιστικών συστημάτων που αναφέρεται σε έναν μηχανισμό αφαίρεσης, στοχευμένο στην απόκρυψη λεπτομερειών της υλοποίησης και της κατάστασης ορισμένων υπολογιστικών πόρων από τους χρήστες των πόρων αυτών. Με την εικονικοποίηση είναι δυνατόν να παρουσιάζεται ένας φυσικός πόρος να φαίνεται ως πολλαπλοί και αντίστροφα, πολλοί φυσικοί πόροι σαν ένας. Εκτός από την μείωση του κόστους και για τους πελάτες και για τους παρόχους Cloud υπηρεσιών, η εικονικοποίηση είναι λύση σε πληθώρα άλλων προβλημάτων όπως διαχείριση πόρων, συντήρηση servers, υποχρησιμοποίηση των servers, δυναμικό καταμερισμό φορτίου κ.α..

2.2 Τομείς εικονικοποίησης

Υπάρχουν πολλά είδη εικονικοποίησης και διαχωρίζονται ανάλογα με το τεχνολογία που επιχειρούν να προσομοιώσουν και τον τρόπο που το πετυχαίνουν αυτό. Ακολουθούν τα σημαντικότερα:

- **Εικονικοποίηση υλικού**. Σε αυτή την μορφή εικονικοποίησης ένα λογισμικό ελέγχου (hypervisor) το οποίο εκτελείται σε πραγματικό υλικό προσομοιώνει ένα υπολογιστικό περιβάλλον, που ονομάζουμε εικονική μηχανή. Η εικονική μηχανή τρέχει ένα φιλοξενούμενο λογισμικό και είναι απομονωμένη από το υπόλοιπο σύστημα. Υπάρχουν τρεις βασικοί τρόποι εικονικοποίησης υλικού και διαχωρίζονται ανάλογα με το πόσο το φιλοξενούμενο λειτουργικό σύστημα έχει γνώση ότι εικονικοποιείται:

  - **Πλήρης εικονικοποίηση**: Σε αυτή την περίπτωση το λειτουργικό σύστημα του εικονικού μηχανήματος δεν έχει γνώση ότι υπόκειται σε εικονικοποίηση και ως εκ τούτου δεν είναι τροποποιημένο. Αυτό έχει ως αποτέλεσμα το υλικό να προσομοιώνεται από το πραγματικό μηχάνημα.
  - **Μερική εικονικοποίηση**: Στη μερική εικονικοποίηση, η εικονική μηχανή εξομοιώνει κάποιο μέρος του υποκείμενου hardware περιβάλλοντος. Μία από τις μορφές της μερικής εικονικοποίησης είναι η εικονικοποίηση του χώρου διευθύνσεων ώστε κάθε εικονικό μηχάνημα να έχει ανεξάρτητο χώρο διευθύνσεων.

- **Παραεικονικοποίηση**: Η παραεικονικοποίηση είναι μία βελτίωση της τεχνολογίας της εικονικοποίησης υλικού όπου το φιλοξενούμενο λειτουργικό σύστημα έχει επίγνωση ότι τρέχει σε περιβάλλον προσομοίωσης. Σε αυτό το πλαίσιο το φιλοξενούμενο λειτουργικό σύστημα έχει κατάλληλα διαμορφωμένους οδηγούς συσκευών ώστε αντί να εκτελούν εντολές προς το υλικό να εκτελούν εντολές προς το λογισμικό ελέγχου (hypervisor).

- **Εικονικοποίηση σε επίπεδο λειτουργικού συστήματος** είναι μία μέθοδος εικονικοποίησης όπου το φυσικό μηχάνημα και τα φιλοξενούμενα περιβάλλοντα μοιράζονται τον ίδιο πυρήνα. Στην πραγματικότητα, το ίδιο το λειτουργικό σύστημα του φυσικού μηχανήματος παρέχει μηχανισμούς ώστε να μπορούν να συνυπάρξουν πολλαπλά εικονικά απομονωμένα περιβάλλοντα που ονομάζονται containers. Παραδείγματα της τεχνολογίας αυτής είναι οι OpenVZ, LXC, Docker containers ενώ στην ίδια κατηγορία εικονικοποίησης ανήκουν και οι FreeBSD jails και το croot jail.

- **Εικονικοποίηση εφαρμογών** η οποία αναφέρεται στον εγκλεισμό της εφαρμογής σε ένα απομονωμένο εικονικό περιβάλλον (sandbox). Κατά την εκτέλεση εφαρμογών σε εικονικοποιημένο περιβάλλον προσβάσεις σε φυσικούς πόρους του συστήματος εγκλωβίζονται και αντιστοιχίζονται σε προσβάσεις σε εικονικούς πόρους. Η χρηστικότητα αυτής της τεχνολογίας έγκειται στο γεγονός ότι μπορεί κανείς να τρέξει εφαρμογές σε περιβάλλοντα που δεν έχουν προβλεφθεί να τις υποστηρίζουν. Για παράδειγμα το πρόγραμμα Wine επιτρέπει σε Microsoft Windows προγράμματα να τρέχουν σε Linux συστήματα.

- **Εικονικοποίηση δικτύου** που είναι η διαδικασία της συνένωσης των πόρων υλικού και λογισμικού σε ένα ενιαίο πρόγραμμα διαχείρισης, το εικονικό δίκτυο.

- **Εικονικοποίηση αποθήκευσης** που είναι η διαδικασία της παρουσίασης των πραγματικών αποθηκευτικών συσκευών σαν λογικές μονάδες (πχ. τεχνολογία RAID).

- **Εικονική μνήμη** που αναφέρεται στην τεχνική που χρησιμοποιεί το λειτουργικό σύστημα ώστε οι εφαρμογές να πιστεύουν ότι έχουν πρόσβαση σε συνεχόμενο χώρο διευθύνσεων, ωστόσο στην πραγματικότητα η μνήμη τους δεν είναι συνεχόμενη στο φυσικό μέσο αποθήκευσης.

2.3 Linux Containers

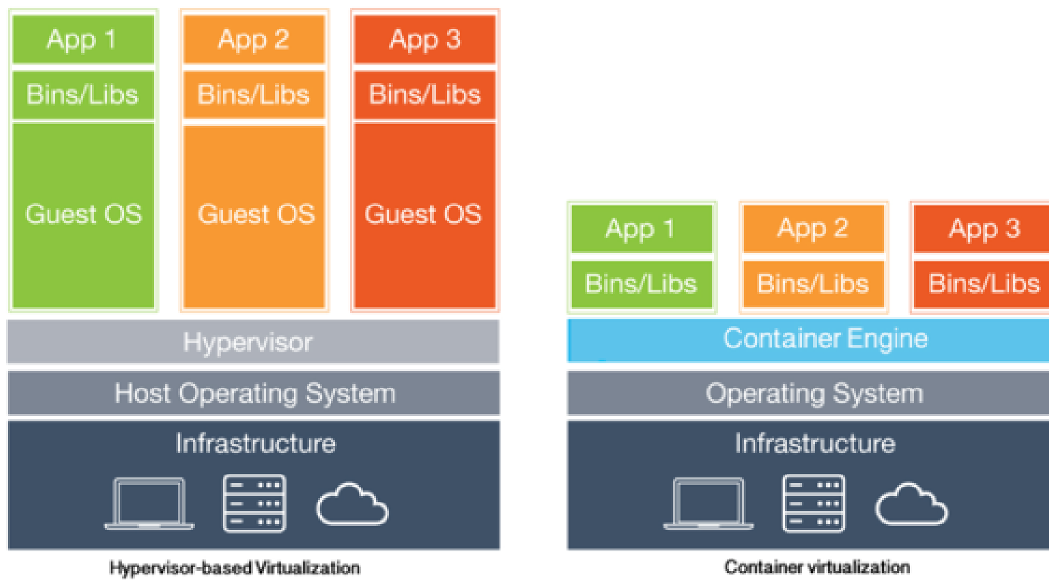
2.3.1 Επισκόπηση των Linux Containers

Οι *Linux Containers* είναι μία τεχνολογία εικονικοποίησης σε επίπεδο λειτουργικού συστήματος που δίνει την δυνατότητα να τρέχει κανείς πολλαπλά απομονωμένα Linux συστήματα (containers) σε ένα μοναδικό φυσικό μηχάνημα (host). Οι containers μοιράζονται τον πυρήνα του host μηχανήματος και παρέχουν ένα απομονωμένο περιβάλλον που έχει την δικιά του CPU, μνήμη, block I/O, δίκτυο και μηχανισμό ελέγχου χρησιμοποίησης των πόρων του συστήματος.

Υπάρχουν κάποιες απαιτήσεις από το σύστημα ώστε να είναι δυνατή η δημιουργία Linux Containers, οι περισσότερες απο τις οποίες παρέχονται από τον πυρήνα του Linux. Συγκεκριμένα, ένας Linux Container χρειάζεται απομόνωση χώρου ονομάτων (Namespaces) που χρησιμεύουν για την δημιουργία απομονωμένου περιβάλλοντος και cgroups που χρησιμοποιούνται για να περιορίσουν την χρησιμοποίηση των πόρων του συστήματος. Τέλος για την ενίσχυση της ασφάλειας χρησιμοποιείται το SELinux που εξασφαλίζει την απομόνωση του host μηχανήματος από τους φιλοξενούμενους containers.

Είναι σημαντικό να κατανοήσει κανείς τα χαρακτηριστικά που διαφοροποιούν την τεχνολογία των containers από αυτή των εικονικών μηχανών για να αναγνωρίσει τα οφέλη που παρέχουν και τις περιπτώσεις στις οποίες η χρήση των containers συνιστάται έναντι των εικονικών μηχανών. Οι containers εικονικοποιούν ένα λειτουργικό σύστημα το οποίο μοιράζονται πολλές εφαρμογές. Αντίθετα, στην τεχνολογία των εικονικών μηχανών το υλικό (hardware) είναι αυτό που προσομοιώνεται και τρέχει πολλαπλά λειτουργικά συστήματα. Στην πράξη, οι containers μοιράζονται το πυρήνα του

Figure 2.1: Πλήρης εικονικοποίηση υλικό και εικονικοποίηση επιπέδου λειτουργικού συστήματος



λειτουργικού συστήματος του μηχανήματος που τους φιλοξενεί, κάτι που δεν ισχύει στις εικονικές μηχανές, όπως π.χ. στο KVM. Ορισμένες φορές μάλιστα είναι δυνατόν τον φιλοξενούμενο μηχάνημα να μοιράζεται τις βιβλιοθήκες και τα εκτελέσιμα του πραγματικού μηχανήματος. Αυτός ο διαμοιρασμός αρχείων έχει ως αποτέλεσμα το μέγεθος ενός container να είναι πολύ μικρότερο (μερικά Megabyte) από μίας εικονικής μηχανής (μερικά Gigabyte). Τα παραπάνω έχουν σαν αποτέλεσμα οι containers να είναι πιο ελαφριά τεχνολογία και για αυτό τον λόγο είναι δυνατόν να φιλοξενεί κανείς περισσότερους containers από εικονικές μηχανές στο ίδιο υλικό. Επιπλέον δεδομένου ότι οι containers μοιράζονται το λειτουργικό σύστημα του πραγματικού μηχανήματος, η συντήρηση του λειτουργικού συστήματος ανάγεται μόνο στην συντήρηση ενός λειτουργικού που καλύπτει όλα τα εικονικά μηχανήματα. Ωστόσο, στην περίπτωση των VMs καθένα πρέπει να συντηρείται ξεχωριστά (bug fixes, patches κτλ). Τέλος, λόγω του διαμοιραζόμενοι πυρήνα, οι containers πρέπει να τρέχουν σε μηχάνημα με ίδιο λειτουργικό σύστημα με αυτό που θα τρέχουν αυτοί. Δηλαδή οι Linux Container πρέπει αναγκαστικά να φιλοξενούνται από Linux μηχάνημα, χωρίς ωστόσο να είναι αναγκαστικά ίδιες και οι διανομές (εφικτή η ύπαρξη Gentoo container σε Debian host). Η Εικόνα 2.1 παρουσιάζει την διαφορά μεταξύ τεχνολογιών εικονικοποίησης που στηρίζονται σε containers και εικονικοποίηση υλικού.

Οι Linux Containers διαμορφώνεται κατά κύριο λόγο από τα παρακάτω εργαλεία του Linux. [12] [13]

- **Απομόνωση χώρου ονομάτων ή Namespaces**
  - Utsname namespace: Τα UTS namespaces παρέχουν απομόνωση δύο παραμέτρων του συστήματος: το hostname και το NIS domain name.
  - Ipc namespace: Τα IPC namespaces παρέχουν απομόνωση για συγκεκριμένους τρόπους επικοινωνίας διεργασιών ώστε να ελέγχουν τα κοινόχρηστα δεδομένα.
  - Pid namespace: Τα PID namespaces παρέχουν απομόνωση για τον χώρο αναγνωριστικών διεργασίας (PID namespace). Αυτό σημαίνει ότι διεργασίας που ανήκουν σε διαφορετικό PID namespace μπορούν να έχουν το ίδιο εικονικό PID όπως φαίνεται στην Εικόνα 2.2.
  - User namespace: Τα User namespaces απομονώνουν τα αναγνωριστικά σχετικά με την ασφάλεια όπως τα αναγνωριστικά χρήστη (user IDs) και ομάδας (group IDs), τον κατάλογο ρίζα (root directory) και άλλα. Αυτό επιτρέπει μία διεργασία να έχει μη προνομιούχο

αναγνωριστικό (unprivileged user ID) έξω από κάποιο user namespace αλλά ταυτόχρονα να έχει δικαίωμά υπερχρήστη (root) μέσα στο namespace.

  – Mount namespace: Τα Mount namespaces παρέχουν απομόνωση στα σημεία προσάρτησης (mount points) του λειτουργικού συστήματος. Αυτό σημαίνει ότι διεργασίες σε διαφορετικά mount namespaces είναι δυνατόν να έχουν διαφορετική εικόνα της ιεραρχίας του συστήματος αρχείων.
  – Network namespace: Τα Network namespaces παρέχουν απομόνωση των πόρων του συστήματος που σχετίζονται με την δικτύωση.
  – Πολλαπλά /dev/pts

- **Control Groups**
  – CPU (cpu, cpuset, cpuacct): Αυτό το σύστημα χρησιμοποιείται για τον περιορισμό κάποιων διεργασιών στην χρησιμοποίηση συγκεκριμένου αριθμού κεντρικών μονάδων επεξεργασίας και καθορισμένου χρόνου χρησιμοποίησης αυτών.
  – Memory: Αυτό το σύστημα χρησιμοποιείται για τον έλεγχο της δέσμευσης και χρησιμοποίησης μνήμης από τις διεργασίες.
  – BLKIO: Αυτό το σύστημα ελέγχει τις ταχύτητες διαβάσματος και γραψίματος στον δίσκο, της εντολές που εκτελούνται ανά δευτερόλεπτο, τους χρόνους αναμονής και ανάλογες λειτουργίες που σχετίζονται με συσκευές αποθήκευσης (block devices).
  – Devices: To devices cgroup σύστημα είναι ένας τρόπος ελέγχου του ποιες συσκευές είναι προσπελάσιμες από κάποιες διεργασίες. Η προκαθορισμένη τιμή είναι all που σημαίνει ότι η αντίστοιχη διεργασία έχει πρόσβαση σε όλες τις συσκευές.
  – Network: Το Network cgroup παρέχει τρόπο ώστε να μπορεί κανείς να προσδώσει προτεραιότητα σε κάποιον τύπο πακέτων δικτύου έναντι άλλων και να εκτελέσει λειτουργίες σε αυτά τα πακέτα.
  – Freezer: Το Freezer cgroup επιτρέπει σε κάποιες διεργασίες να παγώσουν όταν ζητηθεί και να συνεχίσουν αργότερα την εκτέλεση τους με αντίστοιχη εντολή. Η διαχείριση των διεργασιών πραγματοποιείται σε σήματα (SIGSTOP, SIGCONT).

- **Δυνατότητες υπερχρήστη (Root Capabilities)**
  – Βοηθούν στο να επιβάλεις χώρους ονομάτων σε προνομιούχους (privileged) containers, περιορίζοντας τις δυνατότητες του υπερχρήστη.

- **Pivot_root**
  – είναι μία κλήση συστήματος που αλλάζει το τι βλέπει η καλούσα διεργασία ως κατάλογο ρίζα του συστήματος αρχείων.
  – Mandatory Access control (MAC) είναι μηχανισμοί όπως το AppArmor και το SELinux και δεν απαιτούνται για την δημιουργία των containers. Ωστόσο συχνά χρησιμοποιούνται γιατί είναι κρίσιμοι για την ασφάλεια.

Κάποια από τα παραπάνω εργαλεία απαιτούν ο πυρήνας να είναι ρυθμισμένος κατάλληλα. Συγκεκριμένα, για τους χώρους ονομάτων πρέπει ο πυρήνας να έχει ρυθμιστεί με CONFIG_<NAMESPACE_OPTION>_NS ενώ το SELinux χρειάζεται την ρύθμιση CONFIG_SECURITY_SELINUX.

Επιπλέον των παραπάνω εργαλείων για την δημιουργία ενός Linux Container κρίσιμες είναι και οι ακόλουθες κλήσεις συστήματος: clone(), netns(), unshare(). Η κλήση συστήματος clone() έχει παρόμοια λειτουργικότητα με την fork() με την διαφορά ότι το clone() επιτρέπει απομόνωση από τους χώρους ονομάτων που αναφέραμε νωρίτερα. Η κλήση συστήματος setns() επιτρέπει στην καλούσα

διεργασία να συμμετάσχει σε κάποιον υπάρχον χώρο ονομάτων, ενώ αντίθετο το unshare() επιτρέπει στην καλούσα διεργασία να δημιουργήσει ένα νέα χώρο ονομάτων για κάποια από τις προαναφερθέντες παραμέτρους.

2.4 Βασική δομή του CRIU

Το CRIU είναι ένα πρόγραμμα που υλοποιεί τις λειτουργίες Checkpoint/Restore σε χώρο χρήστη. Το Checkpoint είναι η διαδικασία της αποθήκευσης της κατάστασης εφαρμογών που τρέχουν κάτι που περιλαμβάνει τον εικονικό χώρο διευθύνσεων, περιγραφητές αρχείων και άλλες βασικές παραμέτρους όπως καταχωρητές. Χρησιμοποιώντας αυτό το εργαλείο καθίσταται δυνατό να παγώσει κανείς μία τρέχουσα εφαρμογή, να αποθηκεύσει την κατάστασή της σε ένα σύνολο αρχείων και κατά το δοκούν να την επανεκκινήσει απο το σημείο στο οποίο πάγωσε (Restore). Το βασικό χαρακτηριστικό που ξεχωρίζει το πρόγραμμα CRIU από αντίστοιχα προγράμματα με παρόμοια λειτουργικότητα είναι το γεγονός ότι είναι υλοποιημένο σε χώρο χρήστη ενώ παράλληλα είναι διαφανές στις εφαρμογές που αποθηκεύει. Σε αυτό το Κεφάλαιο, θα μελετήσουμε το πώς είναι υλοποιημένο το πρόγραμμα CRIU καθώς και κάποια παραδείγματα στα οποία η χρήση του αποβαίνει ιδιαίτερα σημαντική. Ακολουθούν ορισμένα πλεονεκτήματα του CRIU σε σχέση με άλλα προγράμματα που προσφέρουν παρόμοια λειτουργικότητα ([14] [15] [16] [7] [17]).

- Υποστηρίζει containers (LXC, OpenVZ, Docker) αφού καταγράφει την ύπαρξη χώρων ονομάτων, cgroups, ψευδο-τερματικά κ.α. που είναι κάποιες από τις τεχνολογίες που χρησιμοποιούνται στους containers.

- Οι εφαρμογές που ελέγχονται από το πρόγραμμα CRIU δεν καταλαβαίνουν ότι αποθηκεύεται η κατάστασης τους και ως εκ τούτου δεν χρειάζεται τροποποίηση στον κώδικά τους.

- Δεν υπάρχει επιβάρυνση και γενικά κάποια διαφοροποίηση στην εκτέλεση της εφαρμογής που έχει επανακκινηθεί απο αποθηκευμένη κατάσταση σε αρχεία.

- Υποστηρίζεται η αποθήκευση και αποκατάσταση συνδέσεων δικτύου.

- Χρησιμοποιεί μη τροποποιημένο πυρήνα Linux αλλά απαιτεί σχετικά καινούριο (>3.11)

- Δεν χρειάζεται η προφόρτωση βιβλιοθηκών πριν ξεκινήσει η εφαρμογή.

- Δεν απαιτεί δικαιώματα υπερχρήστη.

Το CRIU, για να πραγματοποιήσει την λειτουργία της αποθήκευσης και επαναφοράς χρησιμοποιεί εκτεταμένα τις κλήσεις συστήματος mmap και ptrace με τις οποίες υλοποιεί το λεγόμενο Paracite Call Injection ή έγχυση παρασιτικού κώδικα. Με αυτή την μέθοδο είναι δυνατόν να τρέξει το CRIU μία κλήση συστήματος σε μία ξένη διεργασία, κάνοντας mmap στον αντίστοιχο κώδικα στην μνήμη της διεργασίας και ύστερα με την εντολή ptrace να τον τρέξει. Με το Paracite Call Injection το CRIU παίρνει τα περιεχόμενα της μνήμης που σχετίζονται με της διεργασίες που αποθηκεύει. Επιπλέον, το CRIU αντλεί πολλές από τις απαιτούμενες πληροφορίες για να έχει γνώση της κατάστασης της διεργασίας που αποθηκεύει από το /proc σύστημα αρχείων. Το συγκεκριμένο σύστημα αρχείων παρέχει μία διεπαφή για τις δομές δεδομένων του πυρήνα. Για παράδειγμα, το CRIU χρησιμοποιεί το **/proc/<PID>/children** για να μάθει πληροφορίες για το δέντρο διεργασιών, το **/proc/<PID>/pagemap** το οποίο δείχνει τις αντιστοιχίσεις για κάθε διεργασία από εικονικές σελίδες στην μνήμη σε φυσικές. Τέλος, για τον έλεγχο ανοιχτών περιγραφητών αρχείων χρησιμοποιεί το **/proc/<PID>/fd** ενώ πληροφορίες για τα σημεία προσάρτησης βρίσκει στο **/proc/<PID>/mounts**.

Σε αυτό το σημείο μπορούμε να δούμε κάποιες γενικές πληροφορίες για την διαδικασία αποθήκευσης (Checkpoint) για την περίπτωση ενός Linux Container:

- Αρχικά συμβαίνει το πάγωμα των διεργασιών του container και η επαναφορά τους στην πιο κοντινή γνωστή κατάσταση. Επιπλέον απενεργοποιούνται οι συνδέσεις με το δίκτυο ώστε να μην αλλάξει χωρίς να γνωρίζουμε η κατάσταση κάποιας διεργασίας. Για το πάγωμα των διεργασιών το CRIU χρησιμοποιεί το cgroup freezer ή το PTRACE_SEIZE της κλήσης συστήματος ptrace.
- Ύστερα αποθηκεύεται η κατάσταση του container σε ένα σύνολο αρχείων. Για να πετύχει αυτό το CRIU είτε διαβάζει το proc σύστημα αρχείων είτε τρέχει παρασιτικό κώδικα μέσα στις διεργασίας ώστε να πάρει το περιεχόμενο της μνήμης, σήματα και άλλα αναγνωριστικά όπως PPID, session ID, Process Group ID κ.α.. Οι πληροφορίες που συγκεντρώνονται αποθηκεύονται σε αρχεία σε μορφή protocol buffer.
- Τέλος προαιρετικά σταματάει η εκτέλεση του container σκοτώνοντας όλες τις συσχετισμένες διεργασίες και αποπροσαρτίζοντας (umount) το σύστημα αρχείων του container.

Η διαδικασία της επαναφοράς (Restore) του container σε τρέχουσα κατάσταση γίνεται ως εξής:

- Επαναφορά της κατάστασης του container από την κατάσταση που έχει αποθηκευτεί στα αρχεία. Πρακτικά, αυτό που κάνει το CRIU είναι να δημιουργεί παιδιά με την χρήση του fork() εξαναγκάζοντας τα εικονικά PIDs να είναι ίδια με τα PIDs που είχαν οι διεργασίες όταν αποθηκεύτηκαν. Στην συνέχεια το CRIU επαναφέρει την διαμοιρασμένη μνήμη, τα cgroups, τους χώρους ονομάτων, τα αναγνωριστικά GIS/SID και άλλα. Κατά κύριο λόγο οι κλήσεις συστήματος που χρησιμοποιούνται είναι οι clone(), setns(), unshare() και pivot_root(). Σε αυτό το σημείο όλες οι διεργασίες θα πρέπει να είναι παγωμένες.
- Επαναφορά των διεργασιών σε τρέχουσα κατάσταση σε απεμπλοκή του δικτύου.

Η πιο σημαντική απαίτηση από το CRIU είναι η ρύθμιση CONFIG_CHECKPOINT_RESTORE από τον πυρήνα. Με αυτή την ρύθμιση το CRIU έχει πρόσβαση σε πολλές παραμέτρους του πυρήνα από χώρο χρήστη με την χρήση API. Επίσης για την παρακολούθηση γεγονότων σχετικά με τα συστήματα αρχείων χρειάζεται την εντολή inotify που επίσης πρέπει να είναι ενεργοποιημένη από τον πυρήνα.

2.5 Έμβια και ψυχρή μεταφορά

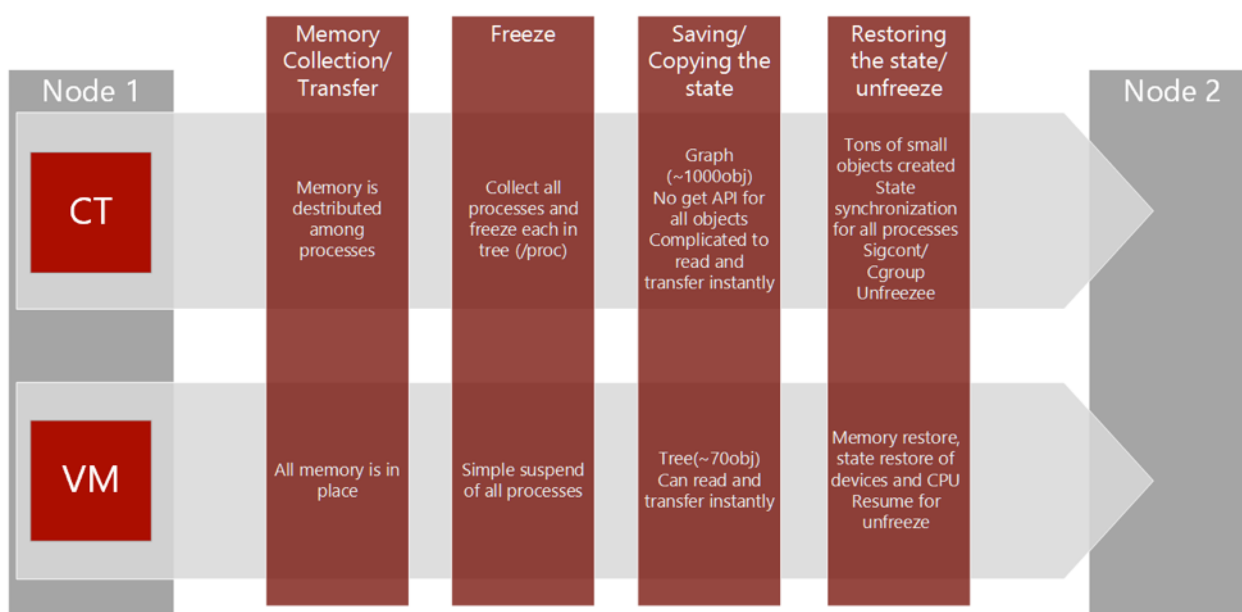
Η μεταφορά στο πλαίσιο της εικονικοποίησης αναφέρεται στην διαδικασία της μεταφοράς μίας τρέχουσα εφαρμογής, ενός εικονικού μηχανήματος ή ενός container μεταξύ φυσικών μηχανημάτων. Από εδώ και στο εξής θα αναφερόμαστε σε μεταφορά containers αν και οι γενική ιδέα παραμένει η ίδια και στα VMs.

Η τετριμμένη μορφή μεταφοράς είναι η ψυχρή (offline), στην οποία πρώτα διακόπτουμε την εκτέλεση του container και και ύστερα τον επανεκτελούμε στον κόμβο προορισμού. Σε κάθε περίπτωση το σύστημα αρχείων του container πρέπει να είναι προσβάσιμο και από το αρχικό και τον τελικό μηχάνημα, κάτι το οποίο λύνεται με κοινόχρηστο storage. Ωστόσο είναι ήδη εμφανές ότι η ψυχρή μεταφορά δεν είναι ιδιαίτερα χρήσιμη αφού προαπαιτεί τον τερματισμό λειτουργίας του αντίστοιχου container. Το πρόβλημα έγκειται στο γεγονός ότι σε ορισμένες περιπτώσεις, δεν είναι δυνατή η διακοπή της λειτουργίας τον εφαρμογών που τρέχουν μέσα στον container. Για παράδειγμα μπορεί ο container να είναι το περιβάλλον εκτέλεσης μίας εφαρμογής ανάλυσης DNA που θα εκτελείται για μεγάλο διάστημα και αν τερματιστεί η λειτουργία της θα χάσει την πρόοδο της.

Επομένως, για περιπτώσεις όπου δεν είναι δυνατό το σενάριο επανεκκίνησης του container, υπάρχει η τεχνική της ζωντανής μεταφοράς. Η ζωντανή μεταφορά αναφέρεται την διαδικασία της μεταφοράς ενός ζωντανού container μεταξύ δύο διαφορετικών φυσικών μηχανημάτων. Η μνήμη και η σύνδεση με το δίκτυο μεταφέρονται όλα από τον αρχικό κόμβο στον τελικό.

Οι λόγοι που καθιστούν την μεταφορά χρήσιμη είναι πολλοί: αρχικά, βοηθάει στην συντήρηση των servers καθώς μπορεί κάποιος να μεταφέρει τους φιλοξενούμενους containers ενός φυσικού μηχανήματος σε ένα άλλο σε περίπτωση που υπάρξει κάποια βλάβη στο αρχικό μηχάνημα ή για κάποιον

Figure 2.2: Μετανάστευση σε VM και CT



λόγω πρέπει να τεθεί εκτός λειτουργίας. Επίσης, είναι χρήσιμη για τον καταμερισμό φορτίου σε περιβάλλοντα Cluster: αυτό συμβάλει στο να μην υπάρχουν υποχρησιμοποιούμενα μηχανήματα ενώ άλλα είναι υπερφορτωμένα. Με την ζωντανή μεταφορά, είναι δυνατόν να μετακινεί ο διαχειριστής του Server ή ακόμα και κάποιο αυτοματοποιημένο πρόγραμμα τους τρέχοντες containers ώστε να επιτευχθεί βέλτιστη χρησιμοποίηση των διαθέσιμων πόρων.

Στην πραγματικότητα, ενώ η ιδέα της ζωντανής μεταφοράς είναι παρεμφερής και στα VM και στους containers, η υλοποίηση της μεταφοράς στους containers είναι πιο πολύπλοκη διαδικασία από ότι στα VMs. Κάποιες διαφορές μεταξύ containers και VMs που επηρεάζουν της διαδικασία της μεταφοράς βλέπουμε στην Εικόνα 2.2. Η βασική ιδέα πίσω από αυτό, είναι ότι τα VM's μπορούμε να τα αντιμετωπίσουμε σαν ένα μαύρο κουτί με όλη την μνήμη συγκεντρωμένη σε αυτό το κουτί, ενώ οι containers είναι μία πληθώρα από διεργασίες και η μνήμη είναι κατανεμημένη σε αυτές. Επίσης δεδομένου ότι ο container αποτελείται από ένα σύνολο διεργασιών, το πάγωμα του πρέπει να γίνει ξεχωριστά για κάθε διεργασία και όχι με την μία όπως στα VMs. Το ίδιο ισχύει και κατά την φάση επαναφοράς (restore) όπου στον container κάθε μία διεργασία πρέπει να δημιουργηθεί ξεχωριστά και να υπάρξει συγχρονισμός των διεργασιών κατά την εκκίνησή τους. Αντίθετα, στο VM η επαναφορά είναι απλά υπόθεση της επαναφοράς της συνολικής μνήμης του VM, την δημιουργία των συσκευών και την επαναφορά κατάστασης της CPU.

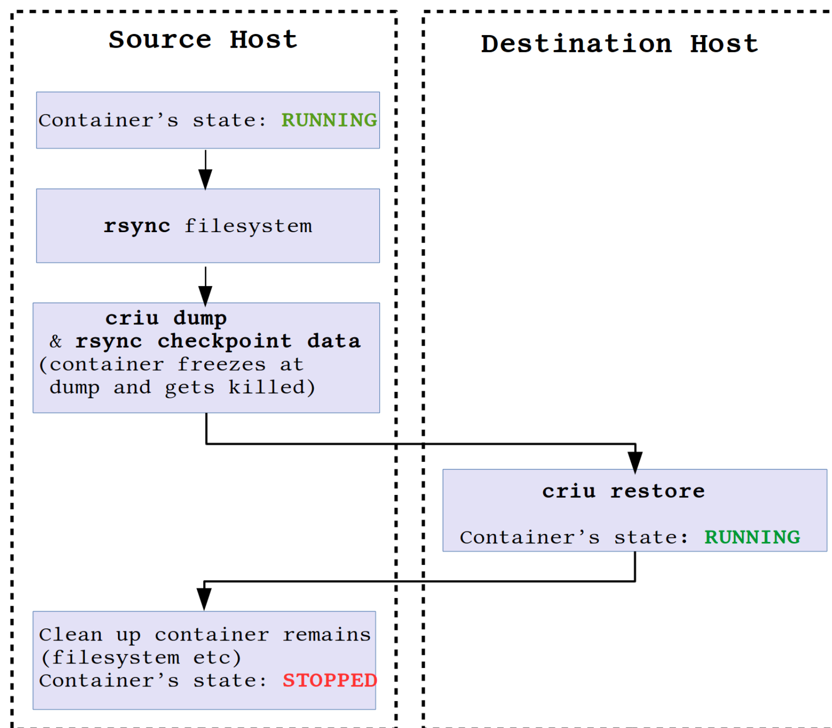
Παρακάτω θα μελετήσουμε τρεις βασικές υλοποιήσεις ζωντανής μεταφοράς ενός Linux Container με το εργαλείο CRIU, ανάλογα με τον διαδικασία μεταφοράς την μνήμης του container από τον κόμβο εκκίνησης στον κόσμο προορισμού:

2.5.1 Αφελής υλοποίηση ζωντανής μεταφοράς LXC

Όπως αναφέραμε και στην Ενότητα 2.4, με το εργαλείο αυτό είναι δυνατόν να αποθηκεύσει κανείς την κατάσταση ενός container που τρέχει σε μία συλλογή αρχείων. Τα αρχεία αυτά μπορούν να χρησιμοποιηθούν στην συνέχεια για την επαναφορά του container στην κατάσταση που αποθηκεύτηκε. Αν αυτά τα αρχεία που παρήγαγε το CRIU μεταφερθούν από τον κόμβο εκκίνησης στον κόμβο προορισμού ώστε η επαναφορά να γίνει στον κόμβο προορισμού και επίσης και οι δύο κόμβοι έχουν πρόσβαση στο σύστημα αρχείων του container τότε έχουμε την πιο απλή δυνατή υλοποίηση ζωντανής μεταφοράς ενός Linux Container χρησιμοποιώντας το εργαλείο CRIU. Η παραπάνω διαδικασία φαίνεται στην Εικόνα 2.3.

Ωστόσο αυτή η τεχνική μεταφοράς δεν είναι ικανοποιητική για περιπτώσεις όπου οι containers

Figure 2.3: Απλή ζωντανή μεταφορά LXC



φιλοξενούν εφαρμογές που έχουν γράψει σε πολλές σελίδες της μνήμης. Ο περιοριστικός παράγοντας σε αυτή την περίπτωση είναι ο χρόνος μεταφοράς των αρχείων από το ένα φυσικό μηχάνημα στο άλλο, και συνεπώς αυτό επηρεάζεται από τον τρόπο και υλικό διασύνδεσης των δύο μηχανημάτων.

2.5.2 Υλοποίηση ζωντανής μεταφοράς LXC με precopy τεχνική αντιγραφής την μνήμης

Μία τεχνική μείωσης της διάρκειας της κατάστασης κατά την οποία ο container είναι μη αποκρίσιμος είναι η precopy τεχνική. Η διαφορά αυτής της τεχνικής με την προηγούμενη αντιμετώπιση είναι ότι η μνήμη αντιγράφεται από τον κόμβο εκκίνησης στον κόμβο μεταφοράς σε πολλαπλά βήματα. Σε κάθε βήμα αντιγράφονται μόνο οι σελίδες της μνήμης που έχουν αλλάξει από το προηγούμενο βήμα. Αυτή η τεχνική μπορεί να έχει πολύ θετικά αποτελέσματα ειδικά σε περιπτώσεις που έχουμε τον container να γράφει σε πολλές σελίδες την μνήμης, ωστόσο η ταχύτητα μεταφοράς δεδομένων μεταξύ των δύο εμπλεκόμενων μηχανημάτων να είναι μεγαλύτερη από την ταχύτητα που η μνήμη αλλάζει.

Η τεχνική precopy υλοποιείται με το εργαλείο CRIU με την λεγόμενη φάση predump και χρησιμοποιώντας "memory changes tracking" [18], μία τεχνική που γίνεται εφικτή σε καινούριους πυρήνες του Linux. Η παραπάνω τεχνική είναι πολύ χρήσιμη ιδιαίτερα αν η μνήμη που χρησιμοποιείται από τον container είναι αρκετά μεγάλη. Αναφέραμε την τεχνική "memory changes tracking" αλλά για να καταλάβουμε το πως λειτουργεί είναι χρήσιμο να παρουσιάσουμε δύο σχετικά πρόσφατα patches στον πυρήνα του Linux που εξάγουν τα παρακάτω δύο αρχεία στο /proc pseudo-filesystem:

- /proc/$pid/clear_refs: Γράφοντας τον αριθμό 4 σε αυτό το αρχείο ζητάμε στον πυρήνα να κάνει 0 στο soft-diry bit για όλες τις σελίδες της διεργασίες με το δοθέν PID. Αυτό χρησιμοποιείται μαζί με το /proc/[pid]/pagemap από το CRIU για να ελέγξει ποιες σελίδες την μνήμης έχουν αλλάξει από τότε που γράφτηκε το αρχείο /proc/[pid]/clear_refs.

- /proc/$pid/pagemap: Αυτό το αρχείο δείχνει την αντιστοίχηση από εικονικές σελίδες για κάθε

διεργασία σε φυσικές σελίδες.

Το CRIU χρησιμοποιεί τα παραπάνω δύο αρχεία ώστε να υποστηρίξει μία πιο αποδοτική σταδιακή φάση αποθήκευσης της μνήμης. Την παραπάνω λειτουργία μπορεί κανείς να την υλοποιήσει ως εξής:

```
1 for i in {1..MAX_ITER}; do
2     criu pre-dump --track-mem --prev-images-dir $checkpointdir/$((i-1))
3                               --images-dir $checkpointdir/$i [options]
4     rsync [OPTIONS] $checkpointdir/$i/ $dest_host:/$checkpointdir/$i/
5 done
6 criu dump --prev-images-dir $checkpointdir/$((MAX_ITER))
7           --images-dir $checkpointdir/$((MAX_ITER+1)) [options]
8 rsync [OPTIONS] $checkpointdir/$((MAX_ITER+1))/
9                 $dest_host:/$checkpointdir/$((MAX_ITER+1))/
```

Χρησιμοποιώντας την επιλογή `-prev-images-dir` , ο χρήστης ζητάει από το CRIU να ψάξει σελίδες που είναι ίδιες με την προηγούμενη φάση αποθήκευσης. Αν αυτό επιτύχει, δηλαδή το CRIU βρει σελίδες που δεν έχουν γραφτεί ξανά, δεν θα τις καταγράψει. Με την επιλογή `-track-mem option` ζητάμε από το CRIU να αρχικοποιήσει τον ελεγκτή αλλαγών για τις σελίδες. Αν γίνει αυτό, στην επόμενη φάση αποθήκευσης της μνήμης του container υπάρχουν πιθανότητες το CRIU να βρει μη αλλαγμένες σελίδες σε σχέση με την προηγούμενη επανάληψη. Η Εικόνα 2.6 σχηματοποιεί την διαδικασία μίας ζωντανής μεταφοράς ενός Linux Container με τον εργαλείο CRIU.

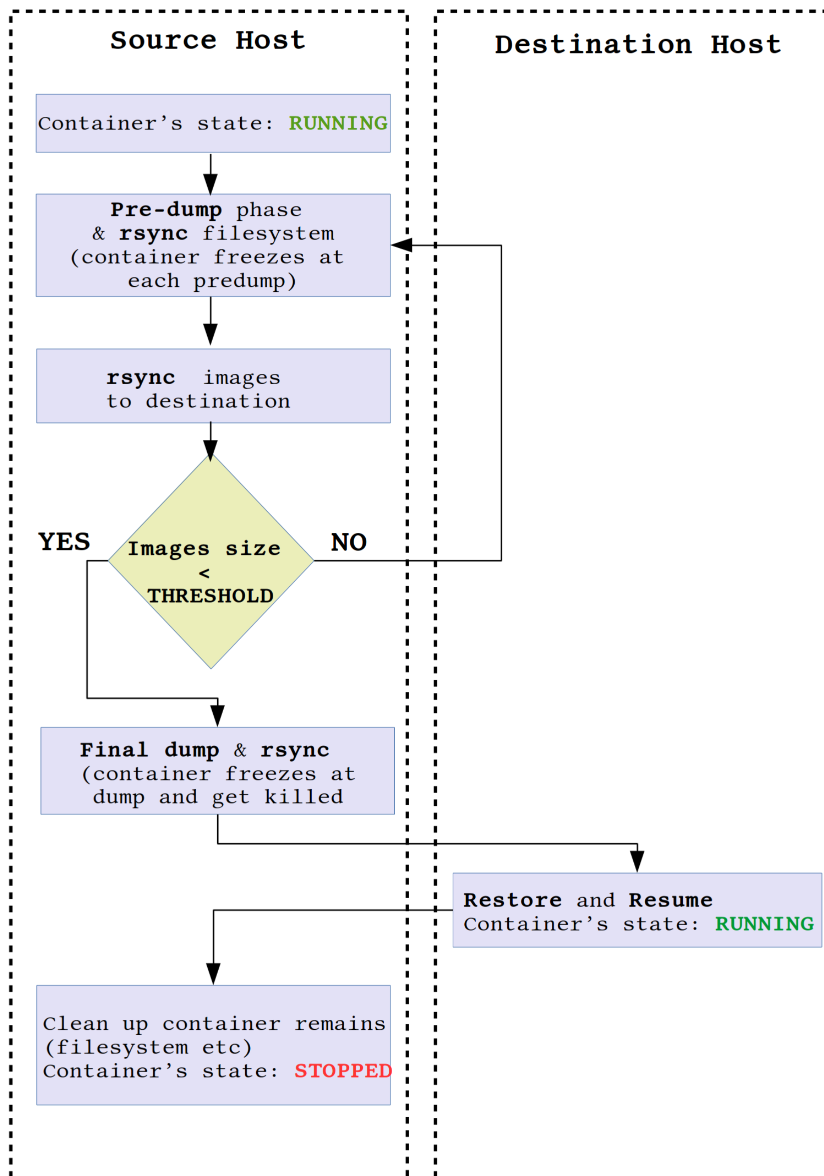
2.5.3 Υλοποίηση ζωντανής μεταφοράς LXC με postcopy τεχνική αντιγραφής την μνήμης

Μία επιπλέον υλοποίηση για ζωντανή μεταφορά Containers είναι η postcopy τεχνική και η ιδέα είναι πανομοιότυπη με τις εικονικές μηχανές. Αυτή η τεχνική προσπαθεί να επανεκκινήσει τον container στον κόμβο προορισμού χωρίς να έχει πρώτα μεταφέρει τις αντίστοιχες σελίδες την μνήμης του container. Οι σελίδες αντίθετα μεταφέρονται όταν ζητηθούν, δηλαδή όταν ο container πάει να διαβάσει ή να γράψει σε κάποια σελίδα που βρίσκεται ακόμα στον κόμβο εκκίνησης. Σε αυτή την περίπτωση, ο container κάνει μία μικρή παύση περιμένοντας για την σελίδα να έρθει και συνεχίζει κανονικά την εκτέλεση του μέχρι να τύχει το επόμενο pagefault.

Το project CRIU, πρόσφατα (Ιούνιο 2016) εισήγαγε μία τεχνική [19] που επιτρέπει την υλοποίηση την ζωντανής μεταφοράς containers με την postcopy τεχνική. Η βασική ιδέα πίσω από αυτή την υλοποίηση είναι η τεχνική του ”memory externalization” και η κλήση συστήματος userfaultfd() [20]. Η κλήση αυτή παρέχει σε χώρο χρήστη την δυνατότητα να ελέγχει κανείς τα pagefaults με τρόπο διαφανή στις εμπλεκόμενες διεργασίες. Η χρηστικότητα που παρέχει το usefaultfd είναι ότι μπορεί πλέον κάποιο πρόγραμμα να τρέχει σε έναν κόμβο ενώ μέρος της μνήμης του να βρίσκεται σε κάποιο άλλο φυσικό μηχάνημα. Η τεχνική που εισήγαγε το CRIU ονομάζεται lazy-pages. Ακολουθεί μία σύντομη περιγραφή της λειτουργίας της:

- Οι σελίδες της μνήμης των διεργασιών χωρίζονται σε δύο κατηγορίες, τις lazy pages και της υπόλοιπες.
- Αρχικά με την εντολή criu-dump με ενεργοποιημένη την επιλογή για τα lazy pages γίνεται καταγραφή της μνήμης των διεργασιών και οι σελίδες που δεν είναι lazy pages αποθηκεύονται σε αρχεία.
- Όταν τελειώσει η φάση καταγραφής (dump phase) το criu ξεκινά έναν TCP server ο οποίος τα χειρίζεται τα αιτήματα για σελίδες από την μεριά που εκτελείται το criu restore.

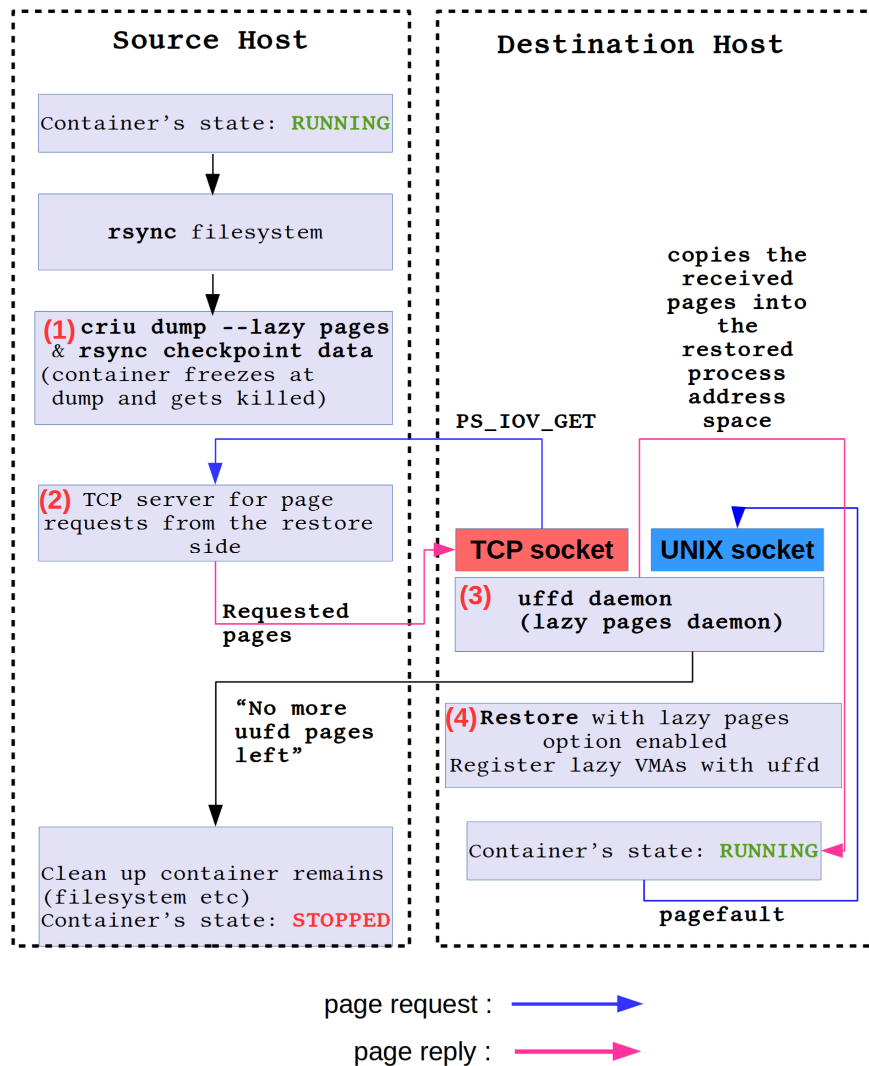
Figure 2.4: Έμβια μεταφορά LXC με precopy μεταφορά μνήμης



- Οι σελίδες που δεν είναι lazy αντιγράφονται μαζί με τα υπόλοιπα .img αρχεία στον κόμβο προορισμού.
- Στον κόμβο προορισμού ξεκινάμε μία διεργασία δαίμονα (uffd daemon) η οποία τα παίρνει τα αιτήματα για σελίδες που λείπουν και θα στέλνει αιτήματα στον TCP server του dump.
- Το criu dump βρίσκει τις ζητούμενες σελίδες και τις στέλνει στον TCP server.
- Η διεργασία uffd αντιγράφει τις σελίδες που ζητήθηκαν στον χώρο διευθύνσεων του container.
- Παράλληλα η διεργασία uufd μεταφέρει όλες τις σελίδες του container, ακόμα και αν δεν έχουν ζητηθεί ακόμα ώστε να τερματίσει την λειτουργία της όταν ολοκληρωθεί η μεταφορά.

Η Εικόνα 2.7 σχηματοποιεί την παραπάνω περιγραφή για την postcopy ζωντανή μεταφορά container με την χρήση των lazy pages.

Figure 2.5: Έμβια μεταφορά LXC με postcopy μεταφορά μνήμης



2.6 Συνεισφορά

Στην παρούσα διπλωματική εξετάζουμε την έννοια της ζωντανής μεταφοράς στους containers καθώς και της τεχνικές τις αποθήκευσης/Checkpoint και επαναφοράς/Restore την κατάστασης των containers που χρησιμοποιούμε για την υλοποίηση την ζωντανής μεταφοράς. Σε αυτό το πλαίσιο, αξιολογούμε διάφορα συστήματα αποθήκευσης και μεταφοράς αρχείων καθώς και δίκτυα διασύνδεσης υπολογιστών όσον αφορά τις επιδόσεις τους σε σενάρια ζωντανής μεταφοράς. Παρέχουμε μία υλοποίηση ζωντανής μεταφοράς με χρήση του εργαλείου CRIU παρέχοντας τις αντίστοιχες μετρήσεις για τους χρόνους μη αποκρισιμότητας των containers και εξηγούμε πώς είναι δυνατόν να βελτιωθούν τα αποτελέσματα. Τέλος, παρουσιάζουμε την συνεισφορά μας στο εργαλείο εικονικοποίησης libvirt, στο οποίο υλοποιήσαμε τις λειτουργίες του Checkpoint και Restore για Linux Containers. Η παραπάνω συνεισφορά είναι ο πρόδρομος για την υλοποίηση του ζωντανής μεταφοράς Linux Containers στην libvirt, κάτι που ωστόσο δεν είναι το πλαίσιο μελέτης της παρούσας εργασίας.

Κεφάλαιο 3

Αξιολόγηση της ζωντανής μεταφοράς LXC

Σε αυτό το Κεφάλαιο, θα παρουσιάσουμε κάποιες μετρήσεις που πήραμε δοκιμάζοντας την τεχνική της ζωντανής μεταφοράς σε Linux Containers. Θα αξιολογήσουμε το κόστος της ζωντανής μεταφοράς σχετικά με την χρησιμοποίηση διαφόρων filesystems και τρόπων μεταφοράς αρχείων, ώστε τα αρχεία που παράγει το CRIU στον κόμβο εκκίνησης να γίνουν προσβάσιμα και στον κόμβο προορισμού. Επίσης θα μελετήσουμε την επιρροή που έχουν δύο διαφορετικά πρότυπα τοπικής δικτύωσης υπολογιστών και συγκεκριμένα τα Gigabit Ethernet και Infiniband.

Για τις μετρήσεις που θα ακολουθήσουν θα μελετήσουμε τα παρακάτω χαρακτηριστικά μίας ζωντανής μεταφοράς:

- Τον χρόνο κατά τον οποίον ο container είναι παγωμένος κατά την διάρκεια που η κατάσταση του αποθηκεύεται σε αρχεία.
- Τον χρόνο που χρειάζεται ώστε τα αρχεία που παράγει το CRIU στον κόμβο εκκίνησης να γίνουν προσπελάσιμα από τον χρόνο προορισμού.
- Τον χρόνο κατά τον οποίον ο container είναι παγωμένος κατά την διαδικασία επανεκκίνησής του στον κόμβο προορισμού.

Στην συνέχεια θα παρουσιάσουμε την υλοποίηση της ζωντανής μεταφοράς των Linux Containers χρησιμοποιώντας το εργαλείο CRIU, θα παρουσιάσουμε τα διάφορα σενάρια ζωντανής μεταφοράς και θα τα μελετήσουμε.

3.1 Υλοποίηση ζωντανής μεταφοράς

Η υλοποιήσή μας που μπορεί να κανείς να δει στο https://github.com/KKoukiou/lxc-migration, είναι ένα κέλυφος γύρω από το CRIU project, ώστε να μπορεί να κανείς να μεταναστεύει Linux Containers παρέχοντας κάποιες βασικές ρυθμίσεις μέσω ενός αρχείου, το migration.conf. Ο κώδικας είναι γραμμένο σε bash. Οι επιλογές που είναι δυνατές για έναν χρήστη είναι οι εξής:

- Το όνομα του container όπως το βλέπουν τα LXC tools.
- Η IP/hostname του κόμβου προορισμού.
- Μέγιστος αριθμός επαναλήψεων για την φάση προεργασίας της αποθήκευσης σελίδων κατά το checkpoint. Αυτή η παράμετρος είναι ένα ανώτατο όριο επαναλήψεων. Ωστόσο αν το πλήθος τον αλλαγμένων σελίδων γίνει αρκετά μικρό αυθαίρετα σταματάμε την φάση την προεργασίας και εξαναγκάζουμε την εκκίνηση μίας τελικής φάσης αποθήκευσης.
- Έλεγχος αποθήκευσης των αρχείων στην μνήμη ή στον δίσκο καθώς και μεταφορά τους με rsync ή κοινόχρηστο NFS σύστημα αρχείων.

Ακολουθεί ένα παράδειγμα ενός configuration αρχείου για την υλοποίηση μας:

```
1  cat migration.conf
2  #Checkpoint data can be stored and tranferred from source to destination
3  #with the following options:
4  # 1: checkpoint on disk,rsync,restore on disk
5  # 2: checkpoint on disk. Destination host shares the checkpoint directory
6  over nfs, so no rsync is needed.
7  # 3: checkpoint on tmpfs, rsync restore on tmpfs
8  # 4: checkpoint on tmpfs. Destination host has nfs over tmpsf on the
9  # checkpoint directory, so no rsync is needed.
10 # Options above should be one of the above {1,2,3,4}
11 # In cases 2 and 4 the nfs server is the destination host
12 Containers_Name: u1
13 Destination_Host: root@xenon8.cslab.ece.ntua.gr
14 Max_Iterations: 3
15 Checkpoint_Option: 3
```

3.2 Test: ext4 έναντι tmpfs σύστημα αρχείων με την χρήση rsync για μεταφορά των δεδομένων σε εφαρμογή με 250MB μνήμη

Σε αυτό το test θα δοκιμάσουμε πώς επηρεάζονται οι χρόνοι της ζωντανής μεταφοράς αν τα αρχεία που παράγει το CRIU αποθηκεύονται στον δίσκο σε ext4 ή στην μνήμη σε tmpfs σύστημα αρχείων. Συγκεκριμένα θα μελετήσουμε τα ακόλουθα σενάρια:

- Τα δεδομένα από το criu dump αποθηκεύονται σε ext4 σύστημα αρχείων στον δίσκο στον κόμβο εκκίνησης. Στην συνέχεια χρησιμοποιούμε rsync για να μεταφέρουμε τα αρχεία από τον αρχικό κόμβο στον τελικό κόμβο, όπου αποθηκεύονται στον δίσκο σε ext4 σύστημα αρχείων από όπου θα χρησιμοποιηθούν στην φάση της επαναφοράς (restore phase)

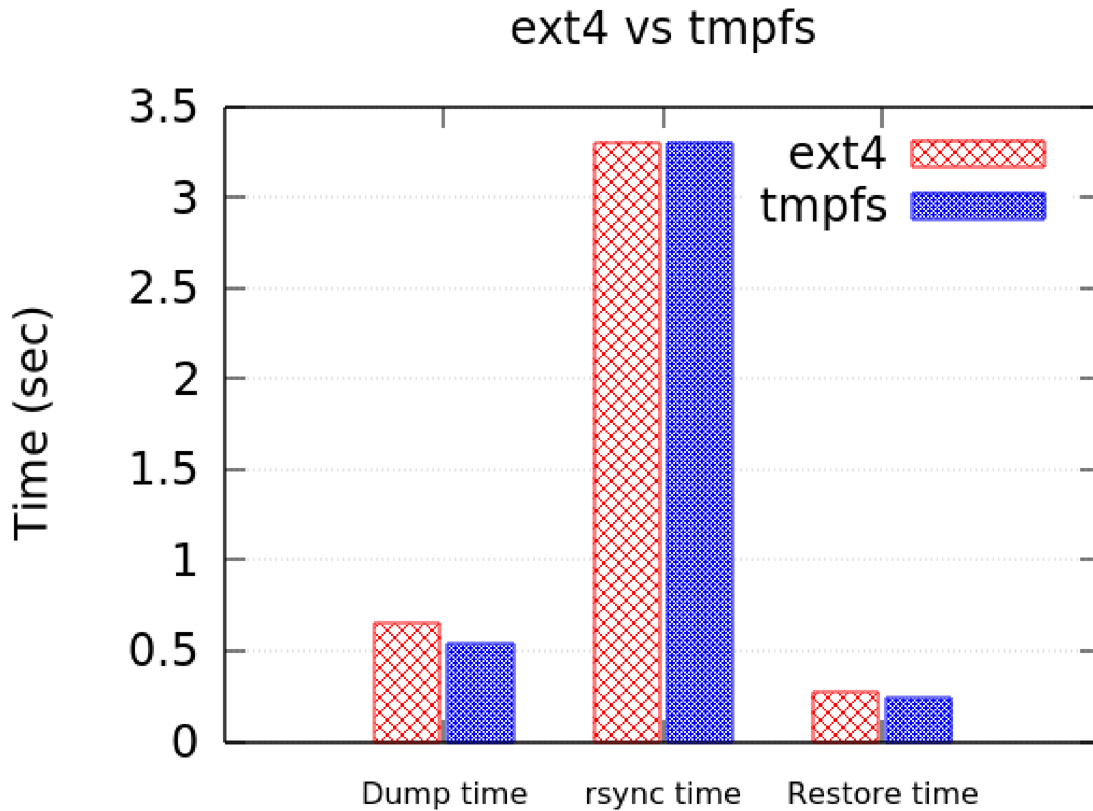
- Τα δεδομένα από το criu dump αποθηκεύονται σε tmpfs σύστημα αρχείων στην μνήμη στον κόμβο εκκίνησης. Στην συνέχεια χρησιμοποιούμε rsync για να μεταφέρουμε τα αρχεία από τον αρχικό κόμβο στον τελικό κόμβο, όπου αποθηκεύονται και πάλι στην μνήμη από όπου θα χρησιμοποιηθούν στην φάση της επαναφοράς (restore phase).

Η Εικόνα 3.1 δείχνει τις τρεις φάσεις της μεταφοράς για τα δύο παραπάνω σενάρια. Για το συγκεκριμένο test χρησιμοποιήσαμε μία επανάληψη της φάσης καταγραφής της μνήμης.

Είναι προφανές ότι τον βασικότερα ρόλο για τις καθυστερήσεις σε αυτό το σενάριο παίζει η φάση της μεταφοράς των αρχείων και ύστερα η φάση καταγραφής της κατάστασης και επαναφοράς. Η μεταφορά των αρχείων επηρεάζεται κυρίως από τη ταχύτητα μεταφοράς δεδομένων μεταξύ τον δύο μηχανημάτων που εξαρτάται από την τεχνολογία διασύνδεσης (40GbE, 100GbE, Infiniband). Ωστόσο, σχετικά τα πειράματα έδειξαν ότι δεν έχουμε βελτιώσεις με την χρήση του Infiniband και αυτό συμβαίνει λόγω του γεγονότος ότι η απόδοση του rsync στα μηχανήματά μας είναι φραγμένη από την συχνότητα της CPU και όχι από την ταχύτητα μεταφοράς των δεδομένων.

Όσον αφορά στην φάση καταγραφής (dump phase), παρατηρούμε ότι το tmpfs και το ext4 έχουν ίδιες επιδόσεις στο συγκεκριμένο setup. Αυτό συμβαίνει γιατί όλα τα δεδομένα που παράγει το CRIU χωράνε στην page cache (RAM) και συνεπώς η ταχύτητα προσπέλασής τους είναι ίδια και στα δύο σενάριο. Ωστόσο στην περίπτωση που τα δεδομένα στην περίπτωση του ext4 δεν έμεναν συνεχώς στην RAM λόγω πιθανού ανταγωνισμού με άλλα προγράμματα, αυτό δεν θα ίσχυε αλλά το ext4 σύστημα αρχείων θα προκαλούσε καθυστερήσεις λόγω swapping. Η εξήγηση για αυτό είναι ότι το γράψιμο και το διάβασμα από το δίσκο είναι πολύ πιο αργό από την μνήμη. Επιπλέον, είναι γνωστό ότι το ext4 είναι ένα σύστημα αρχείο που κρατάει ημερολόγιο (Journal) των αλλαγών ώστε αν υπάρξουν προβλήματα (λόγω ενός βεβιασμένου τερματισμού) να μπορέσει να το επιδιορθώσει. Αυτή η

Figure 3.1: tmpfs έναντι ext4 με rsync σε εφαρμογή με 250MB μνήμη



αποθήκευση του Journal προσδίδει επιπλέον κόστος δίνοντας ωστόσο τα πλεονεκτήματα που προαναφέραμε. Όσον αφορά στην φάση του restore, αφού τα δεδομένα είναι όλα στην RAM δεν έχουμε διαφορές.

Ομοίως, λόγω του γεγονότος ότι τα δεδομένα βρίσκονται όλα στην RAM από την φάση καταγραφής, η φάση μεταφοράς των αρχείων δεν έχει διαφορές στο συγκεκριμένο πείραμα, αν χρησιμοποιούμε ext4 ή tmpfs για την αποθήκευση των αρχείων.

Το ίδιο ακριβώς ισχύει και στην φάση επαναφοράς, όπου τα δεδομένα βρίσκονται εξ'ολοκλήρου στην τοπική RAM.

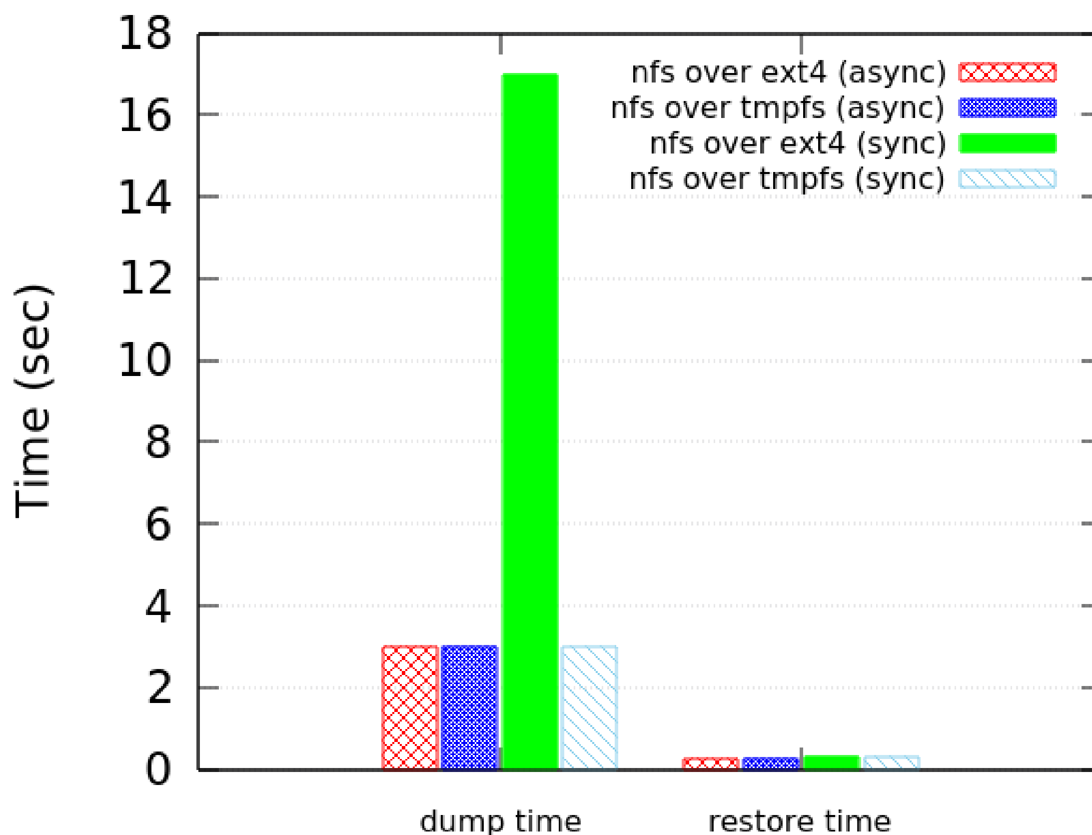
3.3 Test: tmpfs έναντι ext4 με NFS για την κοινή χρήση των δεδομένων από το CRIU σε εφαρμογή με 250MB μνήμη

Σε αυτό το σενάριο, ο κόμβος προορισμού ενεργεί ως NFS server για τα αρχεία από την φάση του checkpoint. Ο κόμβος εκκίνησης, που είναι ο NFS client, ζητάει πρόσβαση εκτελώντας την εντολή mount για τον κατάλογο που βρίσκεται στον κόμβο προορισμού, ώστε να γράψει τα αρχεία του checkpoint. Ύστερα, κάθε απόπειρα εγγραφής από τον κόμβο εκκίνησης θα ανανεώνει τα αρχεία του checkpoint και στον κόμβο προορισμού, ώστε στο τέλος όλα τα αρχεία να είναι διαθέσιμα στον κόμβο προορισμού. Για τα tests θα χρησιμοποιήσουμε NFS v3 με UDP πρωτόκολλο μεταφοράς.

Στην Εικόνα 3.2 βλέπουμε τα κόστη για την φάση dump και restore όταν χρησιμοποιούμε NFS για την εξαγωγή των δεδομένων στον κόμβο προορισμού. Υπάρχουν δύο επιλογές για το πώς ο NFS server αντιμετωπίζει το export, και επίσης δύο για το πως ο NFS client εκτελεί το mount για τον κατάλογο που θα γράψει τα δεδομένα από το checkpoint. Συγκεκριμένα:

NFS server:

Figure 3.2: tmpfs έναντι ext4 με NFS σε εφαρμογή με 250MB μνήμη



- synchronous: O NFS server απαντάει ότι κάποια εγγραφή εκτελέστηκε μόνο όταν οι εγγραφές αυτές πραγματοποιηθούν στην συσκευή αποθήκευσης (πχ σκληρός δίσκος). Αυτό σημαίνει ότι ύστερα από κάθε εκτέλεση εντολής εγγραφής, η εφαρμογή μπλοκάρει μέχρι να πάρει επιβεβαίωση ότι έχει πραγματοποιηθεί η εγγραφή.

- asynchronous: O NFS server δίνει επιβεβαίωση για τις εγγραφές χωρίς να περιμένει να πραγματοποιηθούν στην συσκευή αποθήκευσης.

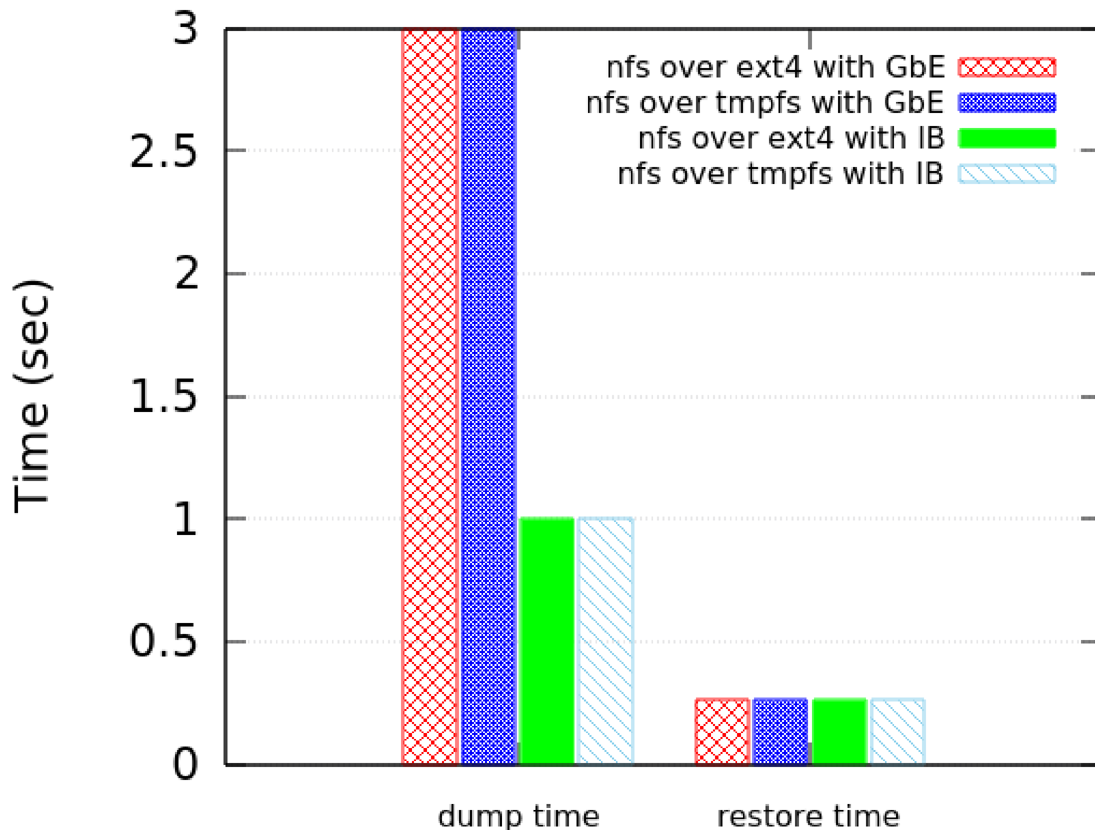
NFS Client:

- synchronous: Όλες οι αλλαγές που γίνονται στο σύστημα αρχείων από τον client γίνονται κατευθείαν commit στον server. Μέχρι να ολοκληρωθεί κάποια εγγραφή οι εγγραφές που ακολουθούν μπαίνουν σε αναμονή. Για καταλόγους που βρίσκονται πάνω σε σκληρούς δίσκους HDD που είναι μηχανικοί αυτό οδηγεί σε πολύ μεγάλες καθυστερήσεις, καθώς το κάθε γράψιμο οδηγεί στην μηχανική κίνηση της κεφαλής του δίσκου ενώ οι υπόλοιπες εντολές εγγραφής βρίσκονται σε αναμονή.

- asynchronous: Με αυτή την επιλογή οι εντολές εγγραφής αποθηκεύονται σε buffer και βελτιστοποιούνται ώστε να επιτύχουν τις ελάχιστες δυνατές μηχανικές κινήσεις στον δίσκο, αναδιατάσσοντας την σειρά που πραγματοποιούνται οι εγγραφές. Με αυτή την επιλογή επομένως, οι εγγραφές δεν μπλοκάρονται και οι εφαρμογή συνεχίζει την εκτέλεση της κανονικά.

Για το test που ακολουθεί μελετούμε την async επιλογή για τον client ενώ για τον server και τις δύο επιλογές. Επίσης, χρησιμοποιήσαμε μία επανάληψη της φάσης καταγραφής της μνήμης, δηλαδή δεν υπάρχει φάση predump.

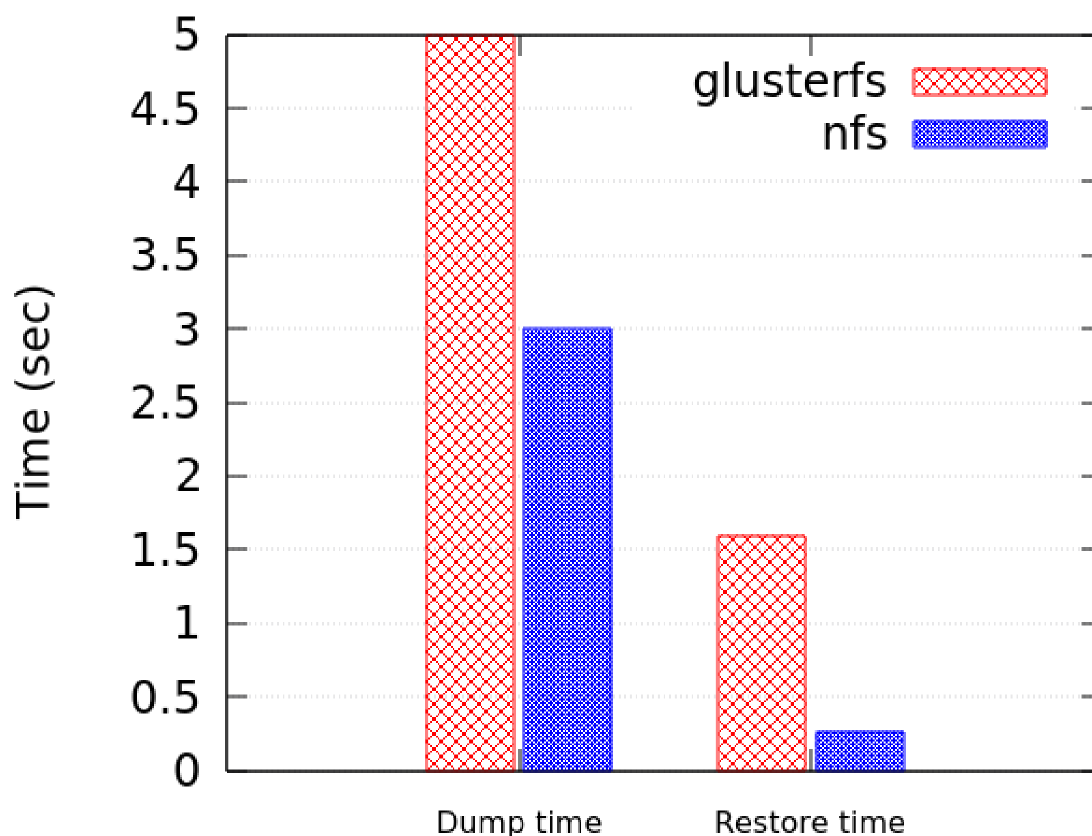
Figure 3.3: tmpfs έναντι ext4 με async NFS export με 250MB μνήμη και GbE έναντι IB



Όπως είναι αναμενόμενο, η sync επιλογή για τον server προκαλεί μεγάλες καθυστερήσεις στην φάση καταγραφής, ειδικά όταν η αποθήκευση γίνεται σε δίσκο. Αυτό συμβαίνει διότι κάθε εγγραφή από τον client μπλοκάρει τις επόμενες εγγραφές, καθώς αναμένει την επιβεβαίωση από τον server ότι η εγγραφή πραγματοποιήθηκε. Κατά την φάση restore δεν υπάρχουν μεγάλες διαφορές καθώς δεν γίνονται πολλές εγγραφές αλλά κυρίως αναγνώσεις από τον server. Η χρήση tmpfs για αποθήκευση των αρχείων στην μνήμη αποβαίνει σε αυτό το παράδειγμα πολύ χρήσιμη στην περίπτωση που χρησιμοποιήσουμε sync επιλογή στον NFS server. Ωστόσο, όπως και στο προηγούμενο πείραμα, σε όλες τις υπόλοιπες περιπτώσεις το tmpfs σύστημα αρχείων έχει τις ίδιες επιδόσεις με το ext4 και αυτό γιατί τα δεδομένα σε όλη την διάρκεια του πειράματος μένουν στην RAM.

Σε αυτό το σημείο, θα μελετήσουμε το κατά πόσο στο συγκεκριμένο πείραμα έχουμε καλύτερους χρόνους αν χρησιμοποιήσουμε Infiniband αντί για GbE σαν δίκτυο διασύνδεσης. Η Εικόνα 3.3 δείχνει τα αποτελέσματα σύγκρισης των δύο αυτών μεθόδων διασύνδεσης για την περίπτωση που ο κατάλογος που θα αποθηκευτούν τα δεδομένα είναι διαθέσιμος μέσω NFS και έχει χρησιμοποιηθεί η επιλογή async για τον NFS server. Είναι εμφανές ότι στην φάση της αποθήκευσης κατάστασης έχουμε πολύ μεγάλη βελτίωση με την χρήση Infiniband. Αυτό σημαίνει ότι ο περιοριστικός παράγοντας εδώ ήταν η ταχύτητα μεταφοράς των δεδομένων, καθώς έχουμε 70% μείωση στους χρόνους κατά την φάση αποθήκευσης. Αναμενόμενο είναι ότι στην φάση επαναφοράς δεν έχουμε καμία βελτίωση, καθώς τα το δίκτυο δεν εμπλέκεται σε αυτό το σημείο αλλά τα αρχεία ανακτώνται τοπικά.

Figure 3.4: GlusterFS έναντι NFS σε εφαρμογή με 250MB μνήμη



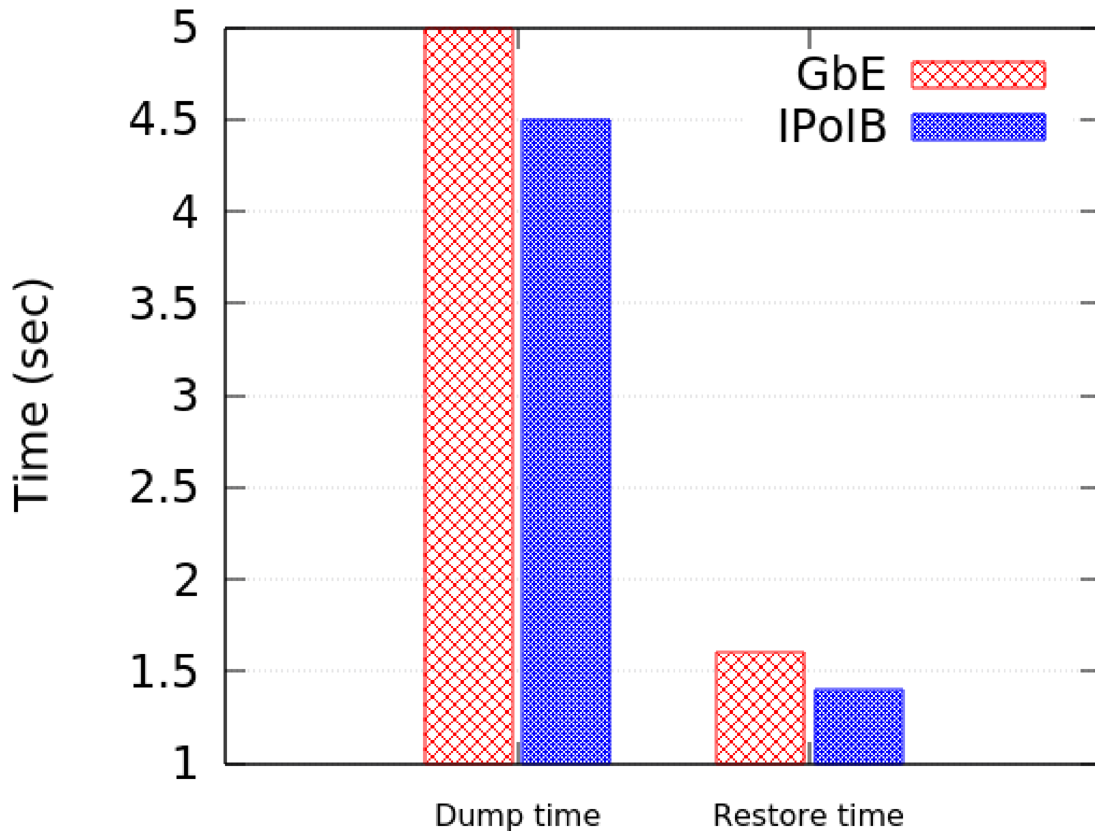
3.4 Test: GlusterFS με ext4 έναντι NFS με ext4 σε εφαρμογή με 250MB μνήμη

Το GlusterFS είναι ένα σύστημα αρχείων υλοποιημένο σε επίπεδο χώρου χρήστη (FUSE). Όταν χρησιμοποιεί κανείς GlusterFS τα αρχεία αποθηκεύονται σε κανονικό σύστημα αρχείων όπως ext4, xfs κα. Για το test που ακολουθεί η αποθήκευση γίνεται σε ext4 και πάνω από αυτό λειτουργεί το GlusterFS για τον συγχρονισμό των αρχείων στα δύο μηχανήματα. Τα αποτελέσματα δείχνουν ότι το NFS έχει καλύτερες επιδόσεις από το GlusterFS στο σενάριό μας, και όσον αφορά την φάση dump αλλά και για την φάση restore. Αυτό συμβαίνει, διότι το NFS στην υλοποίησή μας χρησιμοποιεί UDP πρωτόκολλο για την μεταφορά των αρχείων μεταξύ των δύο κόμβων ενώ το GlusterFS χρησιμοποιεί TCP. Υπάρχουν κάποιες βασικές διαφορές μεταξύ των δύο πρωτοκόλλων που καθιστούν το UDP πιο γρήγορο. Θα δούμε μερικές από αυτές παρακάτω:

- Το TCP πραγματοποιεί τριπλή χειραψία (SYN, SYN_ACK, ACK) κάτι που εξασφαλίζει την αξιοπιστία των συνδέσεων. Το UDP δεν πραγματοποιεί καμία χειραψία.
- Το TCP πραγματοποιεί έλεγχο σειράς των πακέτων, και να αναδιατάσει ανάλογα ώστε να φτάσουν στον παραλήπτη με την ίδια σειρά με την οποία στάλθηκαν. Στο UDP δεν υπάρχει συγκεκριμένη σειρά προσέλευσης των πακέτων.
- Το TCP θεωρείται ιδιαίτερα βαρύ διότι χρειάζονται τουλάχιστον 3 πακέτα για να εγκαθιδρυθεί η σύνδεση.

Στην συγκεκριμένη περίπτωση παρατηρούμε ότι το UDP πρωτόκολλο μας έδωσε πολύ καλά αποτελέσματα. Ωστόσο σε ένα περιβάλλον παραγωγής όπου ίσως θα ήταν σημαντικό να είμαστε σίγουροι για το ότι δεν έχουν υπάρξει χαμένα δεδομένα που θα οδηγήσουν σε αποτυχία στο στάδιο restore,

Figure 3.5: GbE έναντι IB σε GlusterFS για 250MB μνήμη



θα έπρεπε να επιλέξουμε πρωτόκολλο TCP που μας εξασφαλίζει αξιοπιστία. Όσον αφορά στην φάση restore, επίσης το GlusterFS δεν έχει καλές επιδόσεις και αυτό μπορεί να οφείλεται στο γεγονός το πώς λειτουργεί για το GlusterFS η εύρεση των αρχείων. Συγκεκριμένα, ο εντοπισμός των αρχείων γίνεται με αλγοριθμικό τρόπο με έναν αλγόριθμο Hashing κάτι το οποίο προκαλεί καθυστερήσεις στον εντοπισμό των αρχείων συγκριτικά με παραδοσιακές μεθόδους.

Σε αυτή την φάση θα μελετήσουμε το κατά πόσο θα είχαμε καλύτερες επιδόσεις στο GlusterFS αν η ανταλλαγή των δεδομένων γινόταν πάνω από Infiniband δίκτυο. Στην Εικόνα 3.5 παρατηρούμε ότι πράγματι υπάρχει μία βελτίωση της τάξης του 10%. Το βασικό κόστος ωστόσο δεν μειώνεται καθώς οφείλεται από άλλα στοιχεία του πρωτοκόλλου και όχι την ίδια την μεταφορά των δεδομένων. Τέλος, είναι εμφανές ότι το GlusterFS για το σκοπό μας δεν αποτελεί την βέλτιστη επιλογή.

Κεφάλαιο 4

Προσθήκη των λειτουργιών Checkpoint/Restore στην libvirt

Η libvirt είναι ένα σύνολο προγραμμάτων κυρίως υλοποιημένα σε C που παρέχει έναν εύκολο τρόπο διαχείρισης εικονικών μηχανών και άλλων λειτουργιών εικονικοποίησης, όπως containers, εικονικά δίκτυα και εικονική αποθήκευση. Αποτελείται από ένα βιβλιοθήκη που παρέχει API, έναν δαίμονα (libvirtd) και μία πρόγραμμα CLI το virsh. Ο βασικός σκοπός της Libvirt είναι να διαχειρίζεται με κοινό τρόπο πολλαπλούς παρόχους και περιβάλλοντα εικονικοποίησης. Για παράδειγμα η εντολή `virsh list -all` μπορεί να χρησιμοποιηθεί για να δείξει τα υπάρχοντα εικονικά μηχανήματα για όλους τους υποστηριζόμενους hypervisors (KVM, Xen, VMWare ESX etc). Η libvirt παρέχει πολλές εντολές ώστε να καλύπτονται όλες οι υποστηριζόμενες ενέργειες από κάθε πλατφόρμα εικονικοποίησης. Μερικές από αυτές είναι: start, stop, pause, save, restore και migrate. Αν κάποια λειτουργία δεν υποστηρίζεται από την αντίστοιχη τεχνολογία εικονικοποίησης η Libvirt προειδοποιεί με μηνύματα του τύπου ”this function is not supported by the connection driver”.

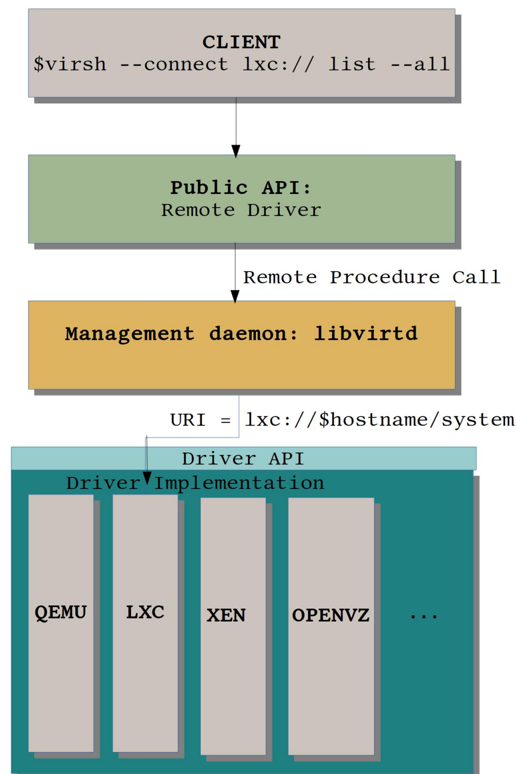
Εκτός από λειτουργίες σε εικονικοποιημένα περιβάλλονται η libvirt προσφέρει και άλλες λειτουργίες όπως έλεγχο συσκευών αποθήκευσης, διαχείριση πόρων δικτύου κα. Τέλος, με τη libvirt μπορεί κανείς να εκτελεί εντολές και σε απομακρυσμένους κόμβους αρκεί να τρέχει και σε αυτούς ο δαίμονας libvirtd.

4.1 Η αρχιτεκτονική της libvirt και ο libvirt-lxc driver

Όπως είπαμε, η libvirt παρέχει έναν γενικό και εύκολο τρόπο να διαχειρίζεται κανείς πολλές τεχνολογίες εικονικοποίησης. Για να υποστηρίζονται οι διάφοροι πάροχοι και οι αντίστοιχοι hypervisors με τις ανάλογες εξειδικευμένες λειτουργίες που υποστηρίζουν η libvirt εισήγαγε την ιδέα των drivers. Στην libvirt υποστηρίζονται οι παρακάτω τεχνολογίες εικονικοποίησης υπάρχει και άρα υπάρχουν οι αντίστοιχοι drivers:

- LXC - Linux Containers
- OpenVZ
- QEMU
- Test - Used for testing
- UML - User Mode Linux
- VirtualBox
- VMware ESX
- VMware Workstation/Player
- Xen
- Microsoft Hyper-V

Figure 4.1: libvirt drivers



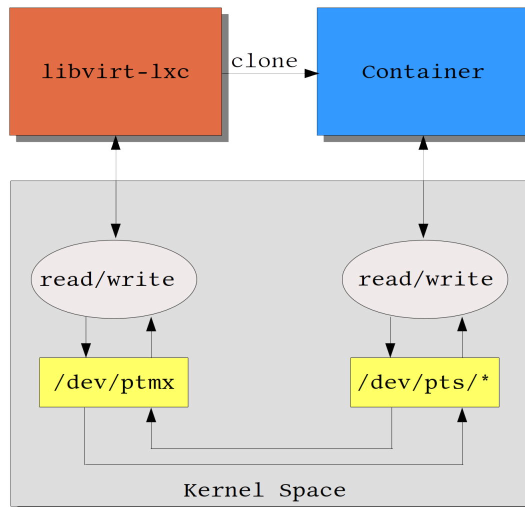
- IBM PowerVM (phyp)
- Virtuozzo
- Bhyve - The BSD Hypervisor

Κάθε driver είναι υπεύθυνος να εξάγει την λειτουργίες που υποστηρίζει η αντίστοιχη τεχνολογία ώστε να μπορούν ύστερα να χρησιμοποιηθούν από το δημόσιο API της libvirt. Η πρόσβαση σε κάθε έναν driver γίνεται ως εξής: Έστω ότι ο χρήστης χρησιμοποιεί το virsh σαν πρόγραμμα χειρισμού. Αρχικά το virsh θα συνδεθεί στον λεγόμενο ”remote driver” και θα τον ενημερώσει με ποιον από τους διαθέσιμους drivers θέλει να συνδεθεί. Ύστερα ο ”remote driver” προωθεί τις κλήσεις στο libvirtd δαίμονα μέσω μίας κλήσης απομακρυσμένης διαδικασίας (RPC). Το libvirtd στην συνέχεια ζητάει από τον ζητούμενο driver να εκτελέσει την εντολή. Ύστερα παίρνει την απάντηση, επιστρέφει τα αποτελέσματα πίσω στο virsh το οποίο θα αποφασίσει πως να τα παρουσιάσει. Η Εικόνα 4.1 σχηματοποιεί την παραπάνω διαδικασία.

Στο πλαίσιο της παρούσας διπλωματικής θα ασχοληθούμε αποκλειστικά με τον LXC driver, που είναι αυτός που χειρίζεται τους Linux Containers. O LXC Driver ή αλλιώς libvirt-lxc δεν έχει εξάρτηση από τα LXC tools, αλλά αντίθετα δημιουργεί τους containers εξαρχής χρησιμοποιώντας τις αντίστοιχες τεχνολογίες του Linux.

Από τεχνική άποψη, κάθε Linux Container που δημιουργείται με την libvirt έχει μία διεργασία που τον ελέγχει. Το εκτελέσιμο αυτής της διεργασίας δίνεται από τον χρήστη σε ένα XML configuration αρχείο που περιέχει γενικά χαρακτηριστικά του container. Σε γενικές γραμμές, η libvirt-lxc διεργασία έχει την ευθύνη να στήσει το περιβάλλον του container και να δημιουργήσει μία κονσόλα για πρόσβαση σε αυτόν. Θα περιγράψουμε παρακάτω περιληπτικά πως γίνεται η δημιουργία ενός Linux Container στον libvirt-lxc driver. Αρχικά εκκινούμε την controller διεργασία η οποία κάνει clone() και δημιουργεί ένα παιδί το οποίο ύστερα από τις κατάλληλες τροποποιήσεις στο περιβάλλον θα εκτελέσει την init διεργασία του container. Για να επιτευχθεί απομόνωση της διεργασίας που θα γίνει ο

Figure 4.2: Ψευδοτερματικά και libvirt-lxc



container από το υπόλοιπο σύστημα χρησιμοποιείται οι κατάλληλες παράμετροι στην κλήση συστήματος clone(). Συγκεκριμένα, CLONE_NEWPID για δημιουργία καινούριο χώρου ονομάτων για τα αναγνωριστικά PID, ώστε για παράδειγμα να μπορεί η init του container να βλέπει ότι έχει PID ίσο με μονάδα, ενώ παράλληλα και η init του host, επίσης να βλέπει το PID ίσο με μονάδα. Αντίστοιχα κάθε έναν από υπόλοιπους χώρους ονομάτων, όπως αυτοί παρουσιάστηκαν στο Κεφάλαιο 2 μπορούμε να τους απομονώσουμε χρησιμοποιώντας την κατάλληλη επιλογή από τις CLONE_NEWNS, CLONE_NEWUTS, CLONE_NEWUSER, CLONE_NEWIPC και CLONE_NEWNET. Σε αυτό το σημείο η libvirt-lxc διεργασία έχει δημιουργήσει κατάλληλο απομονωμένο περιβάλλον για τον container και πρέπει να αλλάξει το κατάλογο ρίζας του container σε περίπτωση που αυτός δεν είναι ο ίδιος με τον κατάλογο ρίζα του host. Αυτό το πετυχαίνει με την χρήση της κλήσης συστήματος pivot_root(). Επίσης ο libvirt-lxc controller, έχει χρέος να παρέχει στον container stdin, stdout, stderr. Για αυτό χρησιμοποιεί ψευδοτερματικά ή pseudoterminal interfaces (PTYs). Ένα ψευδοτερματικό είναι ένα ζεύγος από εικονικές συσκευές χαρακτήρων που παρέχουν ένα αμφίδρομο κανάλι επικοινωνίας. Το ένα άκρο του καναλιού λέγεται master και το άλλο λέγεται slave. Το slave άκρο του ψευδοτερματικού παρέχει μία διεπαφή που συμπεριφέρεται ακριβώς όπως ένα κανονικό τερματικό. Στον container παρέχεται το slave άκρο ψευδοτερματικού και στην controller διεργασία δίνεται το master άκρο. Με αυτόν τον τρόπο ο container έχει πλέον μία κονσόλα. Για τους Linux Containers το master άκρο βρίσκεται στον host στην συσκευή /dev/ptmx ενώ το slave άκρο βρίσκεται στον container στην συσκευή /dev/pts/* όπως φαίνεται και στη Εικόνα 4.2. Ένα επιπλέον βασικό βήμα στην δημιουργία του container είναι ο περιορισμός του όσον αφορά την χρησιμοποίηση των πόρων του συστήματος. Αυτό το πετυχαίνουμε με την χρήση των cgroups με την βοήθεια των οποίων μπορούμε να ελέγξουμε την χρήση της μνήμη, της CPU, των συσκευών αποθήκευσης περιορίζοντάς τις στα επιθυμητά κατώφλια όπως θα έχει οριστεί στο XML configuration αρχείο του container.

4.2 Χρήση των Linux Containers μέσω του εργαλείου virsh

Παρακάτω θα παρουσιάσουμε κάποιες συνηθισμένες εντολές για χειρισμό Linux Containers με το εργαλείο virsh. Όπως αναφέραμε και παραπάνω η ρύθμιση των παραμέτρων του container γίνεται από τον χρήστη μέσω ενός XML αρχείου όπως το παρακάτω.

```
1 cat sh\_container.xml
2 <domain type='lxc'>
3   <name>vm1</name>
```

Λειτουργία	Εντολή
Φόρτωμα του ορισμού του container στην libvirt (μόνο για μόνιμους containers)	`virsh -c lxc:/// define guest.xml`
Αφαίρεση του ορισμού του container από την libvirt (μόνο για μόνιμους containers)	`virsh -c lxc:/// undefine guest`
Εκκίνηση του container	`virsh -c lxc:/// start guest`
Τερματισμός του container	`virsh -c lxc:/// shutdown guest`
Εξαναγκασμένος τερματισμός του container	`virsh -c lxc:/// destroy guest.xml`
Δημιουργία του container (για προσωρινούς containers)	`virsh -c lxc:/// create guest.xml`
Εκκίνηση του container	`virsh -c lxc:/// start guest`
Σύνδεση με την κονσόλα του container	`virsh -c lxc:/// console guest`

Table 4.1: Χρήση virsh με LXC

```
4    <memory>500000</memory>
5    <os>
6      <type>exe</type>
7      <init>/bin/sh</init>
8    </os>
9    <vcpu>1</vcpu>
10   <clock offset='utc'/>
11   <on\_poweroff>destroy</on\_poweroff>
12   <on\_reboot>restart</on\_reboot>
13   <on\_crash>destroy</on\_crash>
14   <devices>
15     <emulator>/usr/libexec/libvirt\_lxc</emulator>
16     <interface type='network'>
17       <source network='default'/>
18     </interface>
19     <console type='pty' />
20   </devices>
21 </domain>
```

Στον Πίνακα 4.2 μπορεί να δει κανείς τι βασικές εντολές χειρισμού Linux Containers με το εργαλείο virsh.

4.3 Υλοποίηση συναρτήσεων Save and Restore για τον libvirt-lxc driver

Σε αυτή την ενότητα θα παρουσιάσουμε τα βήματα που ακολουθήσαμε για να υλοποιήσουμε τις λειτουργίες των Save και Restore για τον libvirt-lxc driver. Σκοπός να ήταν παρέχουμε στην libvirt τις ακόλουθες δύο νέες εντολές:

- `virsh -c lxc:/// save container_name checkpoint_directory`
- `virsh -c lxc:/// restore checkpoint_directory`

Οι παραπάνω εντολές θα έχουν την ίδια λειτουργία με τις αντίστοιχες δύο εντολές των LXC tools.

- `lxc-checkpoint -n container_name -D checkpoint_directory`
- `lxc-checkpoint -r -n container_name -D checkpoint_directory`

Με την εντολή `virsh save` θα μπορεί λοιπόν ο χρήστης Linux Containers που έχουν δημιουργηθεί με την libvirt να αποθηκεύει την κατάσταση των containers ενώ αυτοί τρέχουν. Όπως και τα

LXC tools η λειτουργία του checkpoint/save γίνεται με το εργαλείο CRIU που περιγράψαμε στο Κεφάλαιο 2. Αντίστοιχα με την εντολή `virsh restore` θα μπορεί ο χρήστης να επανεκκινήσει έναν container στην κατάσταση που ήταν όταν αποθηκεύτηκε με το `virsh save`. Και σε αυτή την περίπτωση η τεχνολογία που μας επιτρέπει την λειτουργία του restore είναι το CRIU.

Για την υλοποίηση των παραπάνω επεκτείναμε αρχικά τον κώδικα του libvirt-lxc driver που βρίσκεται στα αρχεία src/lxc/lxc_driver.{c, h} στον κατάλογο της libvirt [6]. Προσθέσαμε δύο νέες συναρτήσεις που θα υποστηρίζονται πλέον, τις lxcDomainSave και lxcDomainRestore.

Όταν εκτελείται η εντολή `virsh -c lxc:/// save` ο remote driver προωθεί την εντολή στο δαίμονα libvirtd ο οποίος ζητάει απο τον libvirt-lxc driver να εκτελέσει την συνάρτηση lxcDomainSave. Η συνάρτηση αυτή πριν εκκινήσει την αποθήκευση της κατάστασης του container πραγματοποιεί κάποιους ελέγχους για το αν η συνάρτηση καλείται με τις σωστές συνθήκες όπως για παράδειγμα ο container πρέπει να τρέχει. Επίσης εκτελεί κάποια κλειδώματα σε διαμοιραζόμενες δομές, ώστε να αποφευχθεί απροσδιόριστη συμπεριφορά από ταυτόχρονη πρόσβαση από άλλα νήματα. Σε αυτό το σημείο εισάγαμε μία αποτελεσματική αντιμετώπιση του συγχρονισμού της ταυτόχρονης εκτέλεσης εντολών πάνω στον ίδιο container [1] [3] [2] [4] [5] [6] [7] για τον libvirt-lxc driver. Στην συνέχεια, ο libvirt-lxc driver καλεί την συνάρτηση lxcCriuSave που είναι ορισμένη στο src/lxc/lxc_criu{c, h} που είναι ένας wrapper γύρω από το CRIU ώστε να γίνει το checkpoint του container.

Η συνάρτηση lxcDomainRestore, που ορίζεται στο src/lxc/lxc_driver.{c, h} καλείται αντίστοιχα όταν εκτελείται η εντολή `virsh -c lxc:/// restore`. Η υλοποίηση για αυτή την λειτουργία δεν είναι τόσο απλή όπως για την `virsh -c lxc:/// save`. Για να καταλάβουμε την υλοποίηση αυτής της λειτουργίας πρέπει να κατανοήσουμε καλύτερα τι έχουμε αποθηκεύσει στα αρχεία με την εντολή `virsh -c lxc:/// save`. Ο container που έχει δημιουργηθεί με την libvirt όπως είπαμε προηγουμένως αποτελείται από δύο μέρη. Την controller διεργασία libvirt-lxc και τις διεργασίες που αποτελούν τον container, δηλαδή το υποδέντρο που έχει σαν ρίζα την init διεργασία του container. Ωστόσο όταν αποθηκεύουμε την κατάσταση του container σε αρχεία με το CRIU αποθηκεύουμε μόνο τις πληροφορίες για το υποδέντρο του container και όχι για την libvirt-lxc διεργασία. Ο σχεδιασμός αυτός οφείλεται σε πολλούς λόγους, ένας εκ των οποίων είναι ότι η libvirt-lxc διεργασία και ο container ανήκουν σε διαφορετικά namespaces κάτι το οποίο το CRIU δεν το υποστηρίζει. Οπότε, όταν ξεκινούμε την επαναφορά (restore) του container πρέπει πρώτα να δημιουργήσουμε εξ'αρχής την διεργασία libvirt-lxc που θα αποτελέσει την controller διεργασία για τον container. Οπότε δημιουργούμε την διεργασία αυτή, και προετοιμάζουμε το περιβάλλον στο οποίο θα εκκινήσουμε τον container. Για παράδειγμα, πρέπει να δημιουργήσουμε τα master άκρα για τα ψευδοτερματικά καθώς στα αρχεία υπάρχουν μόνο πληροφορίες για τα άκρα των ψευδοτερματικών που βρίσκονταν μέσα στον container, δηλαδή τις συσκευές /dev/pts/*.

Κεφάλαιο 5

Σύνοψη

5.1 Συμπεράσματα

Οι υπάρχουσες υλοποιήσεις για Checkpoint/Restore σε Linux Containers δεν επιτρέπουν αποδοτική ζωντανή μεταφορά (`lxc move`, , `lxc-checkpoint [-r]`). Ωστόσο, το CRIU παρέχει τα απαραίτητα εργαλεία (predump, lazypages) ώστε να επιτύχουμε καλύτερους χρόνους ζωντανής μεταφοράς και συνεπώς είναι θέμα υλοποίησης να καταφέρει κανείς να μεταφέρει αποδοτικά containers. Όσον αφορά τα εργαλείο εικονικοποίησης libvirt, οι υπάρχουσες προσεγγίσεις [23] για Checkpoint/Restore/Live Migration δεν έχουν σαν σκοπό την ενσωμάτωση των λειτουργιών που υλοποιούν με το project. Αντίθετα η δική μας υλοποίηση για Save/Restore στον libvirt-lxc driver είναι φτιαγμένη με τέτοιο τρόπο ώστε να μπορεί να ενσωματωθεί στην libvirt.

5.2 Μελλοντικές Κατευθύνσεις

Έχοντας στείλει τα patches της υλοποίησης μας στην libvirt αναγνωρίσαμε ότι πρέπει να υπάρξουν κάποιες αλλαγές στην παρούσα υλοποίηση ώστε να είναι επεκτάσιμη και να μπορεί υποστηρίξει αργότερα ικανοποιητική ζωντανή μεταφορά. Συγκεκριμένα, η παρούσα τεχνική είναι δεσμευσμένη στο να αποθηκεύει τα αρχεία που παράγει το CRIU στον δίσκο ή στην μνήμη σε αρχεία πριν τα μεταφέρει στον κόμβο προορισμού σε περίπτωση ζωντανής μεταφοράς. Αυτό συμβαίνει διότι το CRIU δεν παρέχει κάποιον τρόπο ώστε τα αρχεία που παράγει να γίνονται stream κατευθείαν μέσω sockets στην εφαρμογή που τα ζητάει. Σαν συνέχεια της παρούσας δουλειάς, έχουμε σκοπό να επεκτείνουμε την αρχιτεκτονική του CRIU [24] ώστε να υποστηρίζει την προσωρινή αποθήκευση των αρχείων σε sockets μέχρι αυτά να ζητηθούν από την διεργασία παραλήπτη. Με αυτό τον τρόπο, γλιτώνουμε την περιττή διαδικασία αποθήκευσης αρχείων στον δίσκο που στην περίπτωση μίας ζωντανής μεταφοράς θα αποδειχθεί ιδαίτερα σημαντικό. Παράλληλα δουλεύουμε στην λειτουργια της ζωντανής μεταφοράς [25]. Τέλος υπάρχουν κάποιες βελτιώσεις που πρέπει να γίνουν στην παρούσα υλοποίηση για να έχει πλήρη λειτουργικότητα. Κάποιες από αυτές είναι:

- Χειρισμό των συνδέσεων δικτύου με το εργαλείο CRIU.
- Υποστήριξη πολλαπλών ttys στην περίπτωση που ο container είναι ρυθμισμένος να έχει και άλλες εκτός της tty1.
- Υποστήριξη containers με mounts τύπου efivars.

Βιβλιογραφία

[1] Kubernetes user's guide. http://kubernetes.io/docs/user-guide/.

[2] LXD hypervisor user's guid. https://help.ubuntu.com/lts/serverguide/lxd.html.

[3] Redhat's Openshift user's guide. https://access.redhat.com/documentation/en-US/OpenShift_Enterprise/2/html/User_Guide/.

[4] Linux Containers. https://linuxcontainers.org/, Initial release: August 6, 2008.

[5] Checkpoint/Restore In Userspace. https://criu.org/Main_Page, Initial release: July 23, 2012.

[6] read-only mirror of libvirt repository. https://github.com/libvirt/libvirt.

[7] OpenVZ. https://openvz.org/Main_Page.

[8] Virtuozzo. https://virtuozzo.com/.

[9] Process HAULer. https://criu.org/Main_Page.

[10] https://linuxplumbersconf.org/2015/ocw/sessions/3165.html.

[11] Docker. https://docs.docker.com/engine/userguide/.

[12] namespaces manpage. http://man7.org/linux/man-pages/man7/namespaces.7.html.

[13] Understanding and Hardening Linux Containers. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-10pdf/.

[14] Fault tolerance interface. https://github.com/leobago/fti.

[15] Berkeley Lab Checkpoint/Restart (BLCR) for LINUX user's Guide. https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Users_Guide.html.

[16] DMTCP: Distributed MultiThreaded CheckPointing. http://dmtcp.sourceforge.net/.

[17] Serge E. Hallyn Oren Laadan. Linux-CR: Transparent Application Checkpoint-Restart in Linux. 2010.

[18] mm: Ability to monitor task memory changes. http://lwn.net/Articles/546966/.

[19] Lazy pages support for CRIU. https://patchwork.criu.org/patch/885/.

[20] Memory externalization with userfaultfd. https://www.kernel.org/pub/linux/kernel/people/andrea/userfaultfd/userfaultfd-LSFMM-2015.pdf.

[21] Kernel: Transparent hugepage support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[22] Google summer of code 2016/lxc migration. http://wiki-libvirt.rhcloud.com/page/Google_Summer_of_Code_2016/lxc_migration.

[23] Migrating LinuX Containers Using CRIU. https://techblog.lankes.org/pdf/VHPC16a.pdf.

[24] CRIU: Process Migration using Sockets. https://lists.openvz.org/pipermail/criu/2016-August/030671.html.

[25] Migration support for libvirt-lxc. https://github.com/KKoukiou/my_libvirt_clone/.