UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Ironman: Open Source Containers and Virtualization in bare metal

Ye Yang

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Trabalho de Projeto orientado por:
Prof. Doutor Hugo Alexandre Tavares Miranda

2021

Acknowledgments

This project would not be possible without a structured organization and clear definition of objectives to be achieved. For this I would like to thank my work supervisor at Solid Angle, Gustavo Homem, for not only paving a clear road to follow during the development of the project, but also for giving valuable insights regarding the writing of this thesis.

I would also like to thank my thesis coordinator, Professor Hugo Alexandre Tavares Miranda, for reviewing and enriching the contents of this thesis.

Finally I would like to thank my family for supporting me throughout the development of this project.

Dedicatória.

Resumo

A melhoria no desempenho de componentes físicos tornou possível a virtualização e partilha de recursos computacionais por várias instâncias de sistemas operativos executadas num único computador físico. Esta capacidade computacional permite que um servidor físico consiga executar dezenas a centenas de instâncias isoladas entre si a funcionar de forma independente.

Este projeto tem como principal objetivo explorar o uso profissional da ferramenta de gestão de instâncias virtuais *LXD*, aplicando-a num ambiente *bare-metal* (ambiente com servidores físicos) gerido com metodologia *IaC* (*Infrastructure as Code*). Pretende-se também que o sistema dê resposta às necessidades operacionais de cenários diversos incluindo a possibilidade de gestão via linha de comando, interface web e declarações Puppet e pretende-se a execução de instâncias Linux e Windows Server. O resultado final deverá permitir a configuração de servidores LXD independentes e de conjuntos de servidores a operar em grupo (*cluster*). As instâncias geridas pelo LXD poderão ser baseadas nas tradicionais máquinas virtuais (VMs) ou uma tecnologia mais recente de virtualização, que é mais eficiente na gestão de recursos; *containers Linux* (LXC). A configuração será, tanto quanto possível, automática e baseada em código declarativo.

Embora estejam disponiveis várias ferramentas de virtualização para uso profissional, algumas requerem um pagamento para serem utilizadas na sua totalidade e outras não se adequam aos cenários operacionais apresentados no projeto. O LXD foi designado como o melhor candidato de virtualização pois satisfaz os requisitos principais do sistema a desenvolver, sendo estes:

- Possibilidade de configuração através do *Puppet* permitindo a automatização de criação e gestão de recursos virtuais
- Repositório de imagens de sistemas operativos Linux, nomeadamente o Ubuntu 18.04 e 20.04
- Suporte para Windows
- Suporte para configuração em agrupamentos de servidores (*clustering*)

A criação e gestão de containers Linux é o objectivo inicial do projeto LXD, sendo que a execução de Linux em máquinas virtuais via *KVM* foi introduzida posteriormente.

A introdução do suporte ao KVM possibilitou também a execução de instâncias Windows Server, sendo um dos cenários cobertos por este projeto.

O cenário mais simples contemplado por este projeto é o servidor LXD individual, onde se podem gerir VMs Linux e Windows e containers Linux. Outro cenário explorado no projeto é a utilização do LXD num ambiente *clustered.* Entende-se por cluster o agrupamento de vários computadores físicos que funcionam como um único computador, partilhando os recursos entre si. Num cluster LXD o armazenamento é partilhado através do sistema *Ceph* que protege o conjunto contra perdas de dados. O sistema LXD assegura a mobilidade da execução entre os diferentes computadores do cluster. As instâncias criadas num cluster terão por isso uma tolerância a falhas superior às instâncias criadas num servidor individual.

Tendo em conta que as instâncias geridas pelo LXD trabalham como computadores independentes que desempenham diferentes funções é necessário garantir o bom funcionamento de diferentes tipos de configuração de rede, incluindo rede pública para comunicação externa e rede privada para comunicação dentro do servidor LXD, com alocações de endereço IP estáticas ou via um servidor DHCP. É ainda necessária uma terceira rede - física e privada - que assegura a comunicação CEPH e LXD entre elementos de um cluster.

Num ambiente em cluster, a comunicação em rede realizada pelas ferramentas Ceph e LXD pode ser omitida da rede pública dos servidores físicos. Esta omissão de dados é necessária visto que num cenário de produção o tráfego na rede pública entre os servidores ser pago. De modo a diminuir os custos e garantir o bom desempenho do Ceph nas suas operações sobre os dados em disco das instâncias virtuais, foi necessária a configuração de uma segunda rede física de uso exclusivo pelas ferramentas Ceph e LXD.

O sistema, consoante o ambiente em que é implementado, será operado por utilizadores com diferentes níveis de conhecimento. Para garantir a usabilidade do sistema para variados tipos de utilizadores é necessário disponibilizar diferentes formas de interacção, nomeadamente através de:

- Linha de comandos
- Interface Web
- Código Puppet (IaC)

As formas de interação foram devidamente documentadas de modo a facilitar a operação do sistema.

O *LXDUI* foi a interface web escolhida para interagir com o LXD. Esta interface foi escolhida não só pela simplicidade visual e funcional, mas também por ser escrita na linguagem Python, com a qual a equipa está familiarizada. Foram realizadas várias alterações ao LXDUI para melhorar os níveis de integração, adicionando ou modificando funcionalidades para dar resposta aos casos de uso existentes.

Para o ambiente clustered foi investigada extensivamente a utilização do *Ceph*. A junção do LXD com o Ceph permite tolerância a falhas dos servidores físicos pertencentes a um cluster por via da replicação de dados entre testes. No âmbito do funcionamento em cluster foram investigados e documentados diversos cenários de recuperação de falhas. Foram também investigados alguns indicadores de desempenho cujos valores foram comparadados com os obtidos em servidores individuais. Para o projeto foi testado e configurado um cluster de três servidores físicos, todos configurados através do Puppet. Um cluster com esta dimensão permite a falha de um servidor até a tolerância a falhas não ser garantida, ou seja, se falharem dois servidores em simultâneo num conjunto de três servidores, o cluster irá parar de funcionar corretamente.

A introdução de clustering ao sistema melhora significativamente a disponibilidade das instâncias virtuais, pois se um servidor falhar, basta mover as instâncias a executar no mesmo para outro servidor funcional no cluster a partir do qual as instâncias podem retomar ao seu funcionamento normal. No caso de ser um servidor único de virtualização, se o servidor falhar, as instâncias mantém-se inacessíveis até o servidor retomar o funcionamento normal.

A configuração do sistema em ambos os ambientes - com e sem cluster - é extensa e muito detalhada. Se esta fosse realizada manualmente estaríamos a expor o sistema a possíveis erros humanos durante a configuração, o que poderia conduzir a falhas na instalação inicial do sistema e falta de uniformidade entre diferentes sistemas instalados. Para evitar estas situações e garantir alinhamento com as melhores práticas de IaC, automatizou-se a configuração do sistema utilizando a ferramenta *Puppet*. O Puppet permite criar e gerir recursos de infraestrutura (ficheiros, programas, configurações de rede, entre outros) de forma declarativa utilizando a linguagem própria da ferramenta. Com esta ferramenta é possível configurar de raíz um sistema LXD, fornecendo apenas um ficheiro de configuração com as variáveis chave (devidamente documentadas), sendo o restante assegurado por automatismos. A aplicação do Puppet pode ser usada nos níveis de:

- configurações de servidores LXD
- declaração de instâncias (containers ou VMs)
- configuração dos sistemas operativos das próprias instâncias

Neste projeto foram criados e documentados módulos de Puppet que gerem os aspetos anteriormente referidos. Isto permite a definição de vários modelos de declaração de recursos ou configurações de sistemas. Estes modelos são documentados de modo a poderem ser copiados e aplicados a qualquer máquina pelo utilizador, alterando apenas os parâmetros configuráveis.

Com os resultados apresentados, é possível concluir que a utilização de containers num ambiente de produção introduz melhorias no desempenho, quer no tempo de criação e execução das instâncias, quer no desempenho geral de computação em relação às VMs

tradicionais. O sistema desenvolvido permite a gestão de ambos os tipos de instâncias sendo flexível para qualquer caso de uso.

Com os testes realizados em ambiente clustered com Ceph pode-se também concluir que a introdução do clustering traz benefícios significativos na disponibilidade das instâncias e na tolerâncias a falhas dos servidores físicos. Com a replicação dos dados por entre os vários servidores num cluster conseguimos manter o sistema funcional até um certo limite de falhas, que irá depender do número total de servidores pertencentes ao cluster.

Deste projeto resulta a possibilidade de aplicar tecnologias maduras e robustas como Ubuntu e LXD tanto num ambiente sofisticado de *self-hosted* DevOps como num ambiente IT mais tradicional. Resulta também a possibilidade de aplicar estas tecnologias num ambiente empresarial de produção. De forma simples e eficaz, o projecto permitirá a configuração de raiz de um único servidor de virtualização ou de um cluster de servidores de virtualização. Com o apoio de documentação produzida e testada ao longo do projeto, o sistema final será intuitivo e fácil de utilizar.

Palavras-chave: código declarativo, infraestrutura, sistemas operativos, virtualização

Abstract

Computer virtualization has become prevalent throughout the years for both business and personal use. It allows for hosting new machines, on computational resources that are left unused, running as independent computers. Apart from the traditional *virtual machines*, a more recent form of virtualization was introduced and will be explored in this project, *containers*, more specifically *Linux Containers*.

While multiple virtualization tools are available, some of them require a premium payment, while others do not support container virtualization. For this project, *LXD*, an open source virtual instance manager, will be used to manage both virtual machines and containers.

For added service availability, *clustering* support will also be developed. Clustering will enable multiple physical computers to host virtual instances as if they were a single machine. Coupled with the *Ceph* storage back end it allows for data to be replicated across all computers in the same cluster, enabling instance recovery when a computer from the cluster is faulty.

The infrastructure deployment tool *Puppet* will be used to automate the installation and configuration of an LXD virtualization system for both a clustered and non clustered environment. This allows for simple and automatic physical host configuration limiting the required user input and thus decreasing the possibilities of system misconfiguration.

LXD was tested for both environments and ultimately considered an effective virtualization tool, which when configured accordingly can be productized for a production environment.

Keywords: clustering, containers, LXD, Puppet, virtual machine

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer virtualization allows for the execution of multiple virtual instances on a single hardware server, working independently as separate hosts. This technology has been widely used as a way to maximize the usage of a server's hardware. By allowing multiple independent computers to run on a single server, it lowers costs associated with acquiring physical hardware until the server's computational capacity is reached and only then is a new server required to host further virtual instances.

Other than the traditional *virtual machine* (VM) virtualization, a more recent and lightweight virtualization method will be explored in the development of this project: *container* virtualization, specifically *Linux Containers* (LXC). Containers allow for a fully virtualized system similar to virtual machines, all the while being more lightweight as they share the main software component with the physical hosts operating system responsible for managing the allocation of computational resources, the *kernel* [1]. With lower resource demand, one can host a larger amount of virtualized container instances when compared to VM instances.

To manage both types of instances, *LXD* was used and researched throughout the development of this project. This tool was proposed as it met many of the requirements set for the project, including but not limited to:

- Availability of pre prepared Linux images
- Use of open source and free of charge software
- Instance management for both container and VM instances

Since the system under development would be configured and used in a production environment, the configuration process had to be streamlined to avoid any possible configuration errors. Manual system configuration is always prone to human error. Therefore the process would have to be automated to decrease the amount of human interaction during the setup process and system usage. *Puppet* was the proposed tool for automating the process as it is used internally by *Solid Angle*, the host institution, to setup and configure existing servers.

Clustering is another aspect that will be explored in the development of this project. It involves the use of multiple physical servers that work together as one, distributing computational power among each other to host LXD managed instances. Coupled with LXD, the *Ceph* [1] storage backend will be used to store instance data. Ceph will guarantee data availability and safety by replicating it across all physical servers in the cluster. This introduces fault tolerance to the cluster as it will be possible to recover data of instances hosted on faulty servers that have stopped operating normally due to either software or hardware issues.

The project aims to develop a container and VM virtualization system, both for a single host and clustered environment that is both easily configurable and usable for a wide variety of application scenarios. This will be achieved by documenting usage instructions for both the command line interface and a graphical user interface to accommodate different utilization patterns. The final system should be ready to be deployed in a production environment for both a single server and clustered environment, with both the installation of dependencies and system management done through Puppet, automating the process and decreasing human interaction which in turn decreases any human induced errors when configuring and managing the system.

1.1 Motivation

Many of the virtualization tools available imply some form of premium payment if all of the services offered by the tool were to be used. Others do not support container instances but rather the traditional fully virtualized virtual machines. This apparent gap in available virtualization tools that include both a fully free of charge service and lightweight virtualization option for server management motivated the research and development of this project.

One of the main motivating points was having the possibility of developing and researching a fully open source software based system to virtualize instances in a production environment. This reinforces the use of free software with a robust technological basis that can be used in a production environment, straying further from proprietary software that often times does not provide the user with the freedom to adjust the tools according to their needs.

Cloudmin [2] was the tool used previously at the host institution. It is a user interface designed for *Webmin*, which manages virtual instances. However, this tool proved to be inadequate for a production setup due to the following drawbacks:

- Long virtual disk allocation time
- Unintuitive user interface, exposing too many configurable options to the user

[1]`https://ceph.io/` (Accessed on: April 24, 2021)

[2]`https://www.webmin.com/cloudmin.html` (Accessed on: August 5, 2021)

- Lack of default values on parameters which need to be manually configured

The need to develop a virtualization system that overcomes the previously mentioned drawbacks is also a main motivating point.

1.2 Main objectives

The project's main objective is the development of a production ready system that configures physical servers as a single host for virtualizing instances or as part of a cluster of hosts to distribute the hosts' computational resources and storage space to instances.

A graphical user interface (GUI) will also be adapted for the project's use case guaranteeing system accessibility to users with different backgrounds, mainly regarding their expertise using Linux shells.

The main focus will be on developing Puppet modules to automate the configuration and management process for both environment setups so user input is limited, leading to less potential errors during system setup and usage.

The developed system will be tested and all data collected during the testing phase will be used to produce documentation present on the project's official source code repository's *Wiki* page [2]. This data may be used as reference for future users and further system improvements.

1.3 Host institution

Solid Angle is a company that provides, among others, Linux based server management services[3]. The team behind it is comprised of 10 members, each with a distinct knowledge background which allows the company to offer a varied set of services, from web development to infrastructure planning and management.

The project supervisor within the company, Gustavo Homem, is a co-founder of Solid Angle and project coordinator. He was responsible for organizing the development of the project in *sprints*, each containing a set of main goals to achieve. After each sprint a coherent and well documented aspect of the developed system should be available. The goals within each sprint would change according to the methodology employed and the features provided by the tools that were used during the development of the system, as these would either need the changing of an existing feature or the addition of new features to best suit the developed system.

The company has a strong emphasis on the use of open source software, which was followed through during the development of the project by using only software of the same nature.

[3]`https://www.solidangle.eu/`(Accessed on: October 31, 2020)

With the exploitation of LXD, and the adaption of a web interface, the goal was to develop a central virtualization server that allows for the creation of multiple virtual instances on physical servers residing within the premises of the institution. The developed system will standardize the creation of virtual resources, simplifying its use and management in a production environment, replacing the virtualization system previously used.

1.4 Contributions

To automate system setup and management, Puppet modules were developed to handle the dependencies on both clustered and non clustered environments. Following are the two main Puppet modules developed from root to configure a system according to the project's use case:

- `is_lxd` - a Puppet module used to configure a single host LXD virtualization server, handling among other tasks, network configurations, software package dependency installation and LXD configuration for a production environment.
- `is_ceph_lxd` - a Puppet module used to configure nodes in a clustered environment. This module installs both Ceph and LXD dependencies in the form of software packages and configures a Ceph and LXD cluster from root.

The following Puppet modules were adapted from their original version to include functionalities needed for the project:

- `lxd-puppet-module` by *OVH* [4] - this Puppet module originally developed by OVH only allowed for container instance creation. Improvements were made for the module, mainly the introduction of LXD VM instance creation and image download from LXD's official image repository.
- `puppet-snapd` by *ethanhs* [5] - a module used for installing and managing software packages from the *Snapcraft* [6] software package manager. The original version allowed for installation given the software's version but did not allow for version upgrade which was implemented in the modified Puppet module.

A modified version of a GUI for LXD instance management, *LXDUI*, was developed for the project's usage scenarios. Adaptations had to be made so the GUI has both clustering support and intercompatibility between a clustered and non clustered setup.

Documentation for system installation, configuration and usage for both the command line interface (CLI) and GUI was also one of the main deliveries. This simplifies future system development or management with documented instructions that have been tried and tested throughout the development of this project.

[4]`https://github.com/ovh/lxd-puppet-module` (Accessed on: May 18, 2020)
[5]`https://github.com/ethanhs/puppet-snapd` (Accessed on: May 18, 2020)
[6]`https://snapcraft.io/`(Accessed on: April 2, 2021)

1.5 Document structure

The document is organized as follows:

Chapter 2 introduces the tools used for the development of the project and associated concepts.

Chapter 3 details the single server virtualization system.

Chapter 4 details the clustered virtualization system.

Chapter 5 presents performance benchmarks for both developed systems.

Chapter 6 discusses pros and cons regarding a setup of both a single and clustered virtualization system.

Chapter 7 gives an overview of the developed project.

Chapter 2

Concepts and Tools

The development of this project required the use of various tools for software provisioning, network management, among other crucial aspects. These tools will be introduced along with specific terminology related to them.

2.1 Open-source

The term *open-source* refers to the public nature of the source code of some digital product [3], typically associated with a license, for example *Apache 2.0*[1]. All the tools used in the development of the project were open-source by nature, which greatly benefits the development process. One can not only add new functionalities based on specified needs, but also fix any issues found when using the tools.

2.2 Virtual Machines and Containers

Virtual machines and Containers are the two types of virtualization technology used during the development of the project. One of the main differences between both technologies is the *kernel* used by each. The kernel is software that connects the computer's physical components and its processes. It is mainly responsible for allocating resources (for example CPU and memory) to every running process on a computer, whether it be a physical machine or a virtual one [1].

Containers share the kernel with the host computer, meaning that it is managed alongside the host's other running processes. A VM (Virtual Machine) on the other hand virtualizes its own kernel and the resources (having a slight overhead in regards to resource consumption of the host machine) meaning that its kernel version can be different from the host's. This allows for the creation of virtual machines based on any operating system, for example Windows, as will be discussed further.

[1]`https://www.apache.org/licenses/LICENSE-2.0` (Accessed on: October 30, 2020)

Containers can be categorized in two types, *system containers* and *process containers*. Process containers are simpler when compared to system containers as they only host a single application with all the dependencies needed for it to function correctly. System containers on the other hand function similarly to virtual machines, in which multiple applications can be hosted. They can also run on a different Operating System from the host, as long as the kernel used is the same [4]. An Operating System is a piece of software that controls the execution of applications and acts as a mediator between the computer hardware and software, the Kernel being one of its main components [5]. In this project, container instances used will all be system containers.

Containers are less resource intensive to run in comparison to VMs as resource virtualization introduces an overhead in resource consumption on the host. As system containers only virtualize at the operating system level, it takes significantly less time to create and destroy a container instance in comparison to VMs.

VMs while requiring more resources to start up and execute, are more isolated in comparison to containers, as they virtualize their own kernel, separate from the host. In this sense, the resources used by all the processes inside a VM are unique to it, isolated from the processes running on the host.

VMs are also useful for systems where kernel dependency is important, as if that system requires a kernel update, it can be easily done within the VM since it is isolated. If installed on a container, upgrading its kernel would mean upgrading the kernel of the host machine which could lead to many unwanted side effects, including kernel incompatibility with pre existing services and service downtime for when the host machine is restarted to install the updated kernel [4].

2.3 Netplan

Netplan[2] is a network configuration tool that simplifies the configuration through easily configurable YAML files. These files are automatically generated by Netplan with basic network settings which can be configured to better suit a specific use case.

For the project's production scenario, virtualized instances need to be accessed by physical hosts on the same local area network. Since all virtualization servers will be based on Ubuntu, Netplan was the chosen tool to configure the server's network to enable virtual instance connectivity to the local area network as it was officially supported starting from Ubuntu 17.10 [6].

[2]`https://netplan.io/` (Accessed on: October 6, 2020)

2.4 Puppet

Puppet[3] is a tool that automates the installation and maintenance of software of both physical and virtual machines and is used internally by Solid Angle.

The main concept of Puppet revolves around the existence of a primary server node which will store all the modules and configuration files for specific machines that target specific parts of a system. A machine can join the primary server through the use of a *Puppet agent* after which it will be called a *Puppet node*. The nodes will fetch their own configuration files remotely from the primary server and apply them locally.

This tool will be used to setup physical servers by installing and configuring LXD, applying network configurations, among other crucial aspects so the server is automatically configured in a working state. This will simplify configuration changes related to LXD on any machine as the process will be automatic and require little to no user input, which can always be error-prone.

2.5 Hypervisors

The project's main focus is virtualizing hardware components into multiple virtual instances in order to fully take advantage of the computing capabilities. To create and manage said virtual instances a *hypervisor* will be used.

A hypervisor allows for the creation and management of virtual instances, both containers and VMs. The chosen tool for this project was *LXD*, which met the defined requirements as will be further discussed.

In this section, a brief comparison will be made between LXD and another popular container instance hypervisor *Kubernetes* as well as the main container technology used by it, *Docker* containers.

2.5.1 LXD

LXD is the main tool explored and used during the development of the project. At its core LXD is a REST API built on top of LXC (Linux Containers)[4] and KVM[5], providing a unified and improved command set which can be executed through the *CLI* (Command-Line Interface) to create and manage both LXC and KVM VMs [7].

LXD uses *QEMU*[6] to create its VM instances, which in turn uses KVM as its virtual driver.

[3]`https://puppet.com/` (Accessed on: October 7, 2020)
[4]`https://linuxcontainers.org/` (Accessed on: 20 October 2020)
[5]`https://www.linux-kvm.org/page/Main_Page` (Accessed on: 20 October, 2020)
[6]`https://www.qemu.org/` (Accessed on: October 20, 2020)

Instance creation is based on an official LXD image repository which has a large variety of Linux distributions [7]. These images are automatically transferred whenever an instance is created with an image that is not locally present. Images transferred from other sources can also be used with LXD. The availability of this official repository makes the instance creation process more versatile as images to be used for an instance can be chosen at creation time without any prior preparation. The two Linux distributions used in this project are *Ubuntu* 18.04 and 20.04.

For the project's particular use case, the creation of VMs that use Windows Server 2019 as a base image was also required. The creation of these instances through LXD is not supported by default and the process is more manual when compared to Linux based instances [8].

LXD also allows for the simple configuration of instances through the use of *profiles*. An LXD profile contains a set of pre defined instance configuration options, the following being the main options to take into consideration:

- Usage of the host's physical resources (CPU cores used, memory and disk space used)
- Storage device
- Network configuration

A profile can be defined once and applied to any instance. All instances created with a certain profile will have the same initial configuration options applied, without the need for manual configuration of each instance. This is crucial from a system administrator's perspective as manually configuring instances that use the same resources (mainly network and storage pools) is time consuming and prone to human error. Having the resources already configured in a profile, prevents instance misconfiguration and saves time.

2.5.2 Docker containers

While *Docker*[8] itself is not an instance hypervisor, it is one of the most well known container virtualization technologies.

Docker allows for the deployment of multiple applications in the form of Docker containers, each having the required dependencies for the application to run properly [9]. They can be categorized as process containers as the main focus is the deployment of a single application or multiple services of a larger application [4]. There is also an emphasis on the intercompatibility of Docker containers with hosts running different operating systems so long as they support Docker [10].

[7]`https://uk.images.linuxcontainers.org/` (Accessed on: October 20, 2020)
[8]`https://www.docker.com/`(Accessed on: October 18, 2020)

2.5.3 Kubernetes

One of the main container orchestration tools is *Kubernetes* [11]. Similar to LXD, Kubernetes is used as a container orchestration framework; however, it is mainly used in a distributed clustered environment. It is most commonly used in a cloud environment making use of *nodes*, VMs deployed within the cloud platform [12], to host the *pods*, a group of one or more containers that share a namespace and run a specific service [13]. While other containerization tools are available, Kubernetes is most commonly coupled with the use of Docker containers.

Kubernetes can also be setup in a bare metal environment, but the process for configuring on a bare metal cluster is significantly more complex when compared to an LXD cluster. This is due to the need of configuring hosts in various ways according to its role in the cluster. An example would be control nodes that supervise worker nodes which in turn also have a different setup method [14].

2.5.4 Comparison

LXD vs Docker

Considering that all technologies have the use of container virtualization in common, we can draw a comparison between each tool and LXD which will be used to develop the project.

LXD, manages both LXC and KVM VM instances through a unified and improved command set. The introduction of VM creation through LXD made it possible to create Windows VMs, which is not possible through Docker. An official image repository hosting multiple Linux distributions also makes LXD more versatile when it comes to instance creation. Since all instances virtualize an operating system, one can also install and run dockerized applications within LXD's instances. Containers managed by LXD are system containers as opposed to Docker managed containers. In other words, LXD managed container instances simulate a working operating system: they are able to host multiple services and grant system access similar to a VM [4].

Therefore, one may conclude that Docker is more focused on an application level, installing and configuring a system and its dependencies inside process containers while LXD is more focused on being an instance hypervisor for both LXC and KVM VMs.

LXD vs Kubernetes

The most defining difference between both tools is their primary use case. While Kubernetes is mainly focused on hosting services in the form of applications, LXD focuses on simulating working Linux hosts that function and can be accessed as normal hosts, hosting multiple services, Docker based applications included.

While LXD also allows for setup in a clustered environment, it is not its main focus as it is mainly used to orchestrate both system containers and VMs of multiple Linux based distributions on a bare metal environment.

LXD's concept of clustering is simpler when compared to Kubernetes as it is based on the availability of LXD resources across different hosts. This is done through background LXD API calls sharing any resource updates across the network [15].

A clustered LXD setup, in comparison, is much simpler to setup and configure as all hosts within the cluster have the same role and access the same LXD resources [15]. The sole difference during host setup will be the bootstrap host as it will have to create both the network interface and the remote storage pool to be used by all cluster members. Subsequent members will have an identical cluster joining process.

Both LXD and Kubernetes have their own specific use cases and should be chosen according to its main usage. For this project Kubernetes was not an option as the focus was on virtualizing Linux hosts in container and VM instances which in turn will either host services themselves, or will be used for everyday tasks as a normal host, fitting LXD's primary use case.

Chapter 3

Single virtualization server development

During the introductory phase of the project there was a strong emphasis on the production of documentation for each executed and tested procedure. Documentation was organized on a *Wiki* which is publicly available at the *BitBucket*[2] repository. This documentation centralizes all the information regarding the development and usage of the system and can serve as reference for future use or development of new features. The focus on documenting the results accentuates the business environment in which the project is developed in, as the usage of the final product should be straightforward and intuitive for users with any backgrounds.

3.1 Scenario

The use case presented for single virtualization servers assumed the existence of multiple servers independent from one another in regard to instance storage. These servers would need to have LXD installed and configured equally among each other in order to prevent compatibility issues if and when instance transfer between servers is required, for example, if the host server requires maintenance and must be temporarily shutdown. Network communication would not only occur between the physical hosts and their instances, but also between other hosts' instances.

The server's configuration must be simple, intuitive and automatically configurable by editing a Puppet node declaration template as will be further discussed.

The final version of the system system should be easily accessible, with the respective documentation to guide new users through the process of managing containers and should also be foolproof for different types of users, with varying degrees of command line knowledge.

3.2 Usability

During the initial research and development phase of the project, the type of end user of the system came into question. Since users with different backgrounds will have to be taken into account, having multiple ways of using LXD would be required so that the system is usable for both users with less and more command line knowledge.

LXD does not officially support a GUI so there was a need to find and adapt an existing web interface for LXD.

After a thorough examination of existing GUIs, two were found to have the main functionalities that would be used: *LXDUI*[1] and *LXD Mosaic*[2]. A comparison between both interfaces can be found at Table 3.1.

	Pros	Cons
LXDUI	• Developed in Python, a language common to the team members at Solid Angle simplifying the process of implementing UI improvements • Basic and easy to understand interface that gives access to LXD's main functionalities	• The UI is not implemented by default to be installed inside a container • No clustering support • At the moment of writing (October 24, 2020), the project is not actively being developed or maintained by the authors
LXD Mosaic	• Default installation within a container, encapsulating the dependencies without affecting the host • Clustering support • Preview of host server resource consumption by LXD instances	• Developed in PHP, which is less common among team members at Solid Angle, making any future improvements to the UI harder to implement • UI instability issues when the container used to install the UI is restarted

Table 3.1: Comparison between LXD web interfaces

After thoroughly comparing both interfaces, LXDUI proved to be the best fit for project's the requirements. Although the functionalities provided are limited when com-

[1]`https://github.com/AdaptiveScale/lxdui` (Accessed on: October 8, 2020)
[2]`https://github.com/turtle0x1/LxdMosaic` (Accessed on: October 8, 2020)

pared to LXD Mosaic, the fact that it is written in Python simplifies the implementation of new functionalities that come along with the development of the project.

The next UI development phase introduced improvements for the chosen UI, so it could be installed within a container, one of the main requirements. This necessity came from the fact that the interface needed multiple other dependencies in the form of software packages in order to function properly. If the interface was to be installed on the host machine, these dependencies would be too numerous to be managed and could potentially generate vulnerabilities in the system. By installing the interface inside a container, all its dependencies would also be installed within, simplifying the UI's management process whenever needed (version upgrade, clean uninstallation, among others) without directly interfering with the host system.

Since the UI is hosted in a container, if it were to list its local containers, it would always display an empty list. In order for the container to retrieve information regarding its host's LXD resources it was necessary to add the host as the container's LXD remote server. LXD allows external hosts to access the current host's LXD resources (instances, profiles, images, among others) by setting up the current host as a remote server for the external host (which in this case is the LXDUI container). This can be setup by exposing LXD's default port 8443 to the network by binding it to the host's IP address. A security password is then required to be setup so only known hosts can access the local host remotely [16]. With a LXD remote setup, LXDUI is then able to display information regarding the remote physical host from inside a container.

Given that LXDUI is open-source, a series of *pull requests*[3] with all of the developed improvements were issued, with a production-ready version available at the public BitBucket repository[17].

Below is a list of the main improvements and changes, for the project's use case made for LXDUI:

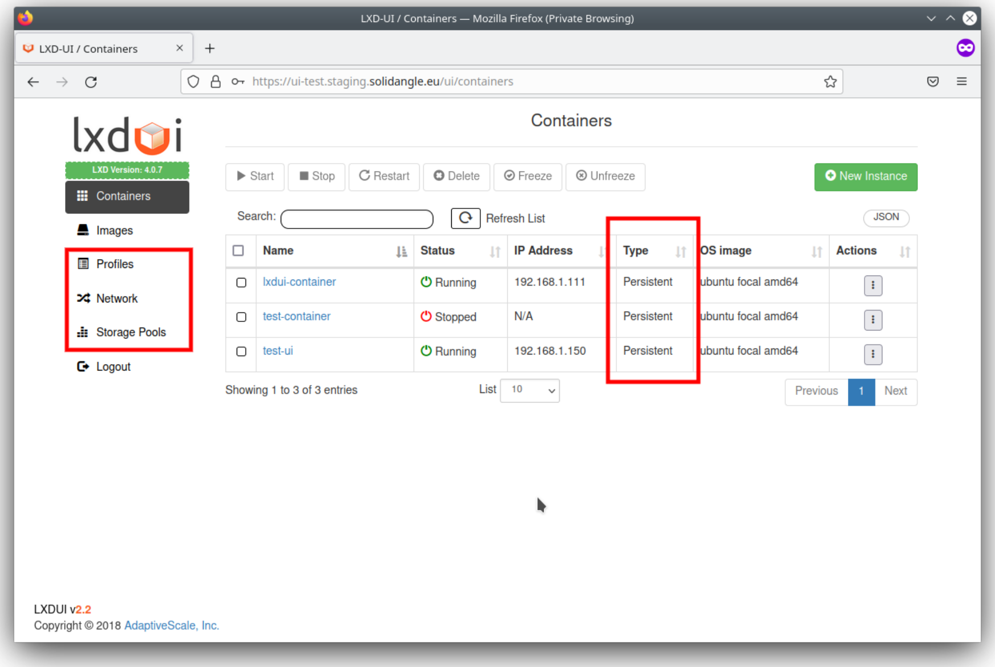
- Fixed console access to remote LXD server instances (instance console access was previously implemented on the UI, but was not fully compatible for instances that were remotely accessed)
- Updated the Python LXD API used in order to support VMs and added instance type column for both instances and images in order to differentiate between a container and a VM through the UI
- Added resource monitoring columns to LXDUI (allocated CPU, memory and storage space), simplifying LXD resource management through the UI
- Added authentication via Linux PAM instead of using LXDUI's password encryption method

[3]`https://github.com/AdaptiveScale/lxdui/pulls?q=is%3Apr+author%3Aye-yng` (Accessed on: January 28, 2021)

- Limited options in LXDUI to only allow access to operating system images and instances, as other options should not be manually accessible as they are automatically configured via Puppet

- Added instance resource usage information to simplify resource management (memory, CPU and disk allocation)

In figure 3.1 the original LXDUI interface is shown. We can see, highlighted on the left side, system configurations that are automatically through Puppet and should not be manually configurable by the end user. The *Type* column shown is also a technical instance aspect that is not informative for the UI user and can be replaced.



Containers

	Name	Status	IP Address	Type	OS image	Actions
☐	lxdui-container	Running	192.168.1.111	Persistent	ubuntu focal amd64	⋮
☐	test-container	Stopped	N/A	Persistent	ubuntu focal amd64	⋮
☐	test-ui	Running	192.168.1.150	Persistent	ubuntu focal amd64	⋮

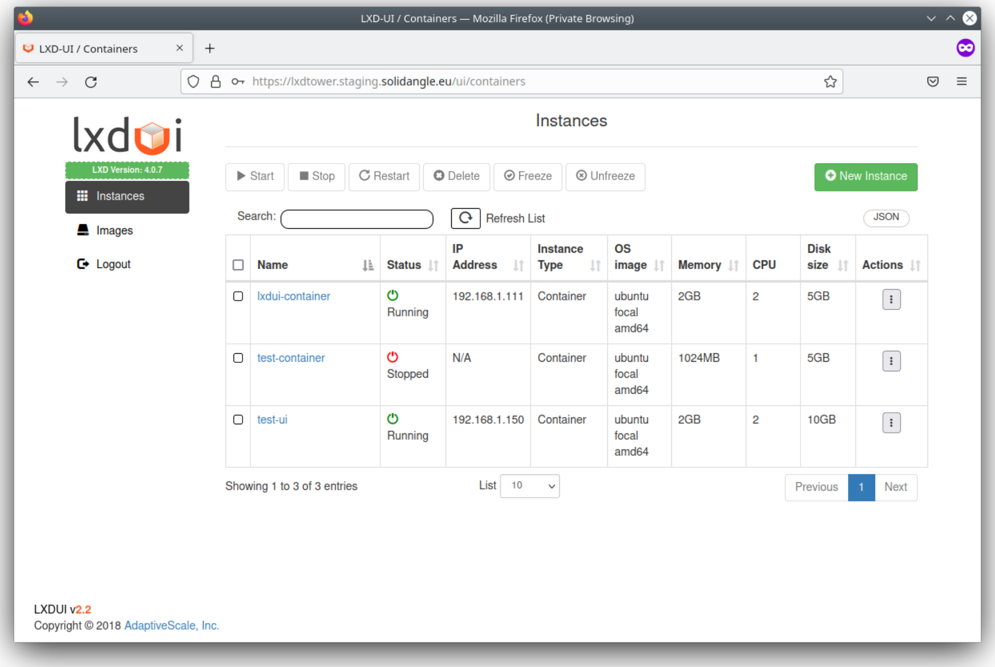
Showing 1 to 3 of 3 entries   List 10   Previous 1 Next

LXDUI v2.2
Copyright © 2018 AdaptiveScale, Inc.

Figure 3.1: Original LXDUI interface

In figure 3.2 the modified standalone version of LXDUI is presented. We can see that Puppet configured aspects of the system are no longer available through the UI. The instance table itself now shows more informative columns, namely resource allocation to each instance (CPU, Memory and disk allocation).

In figure 3.3 the clustered adaptation of LXDUI is shown. Apart from showing in which server an instance is currently located within the cluster, it is also possible to create and instance on a specific server or migrate the execution of an instance to another server.



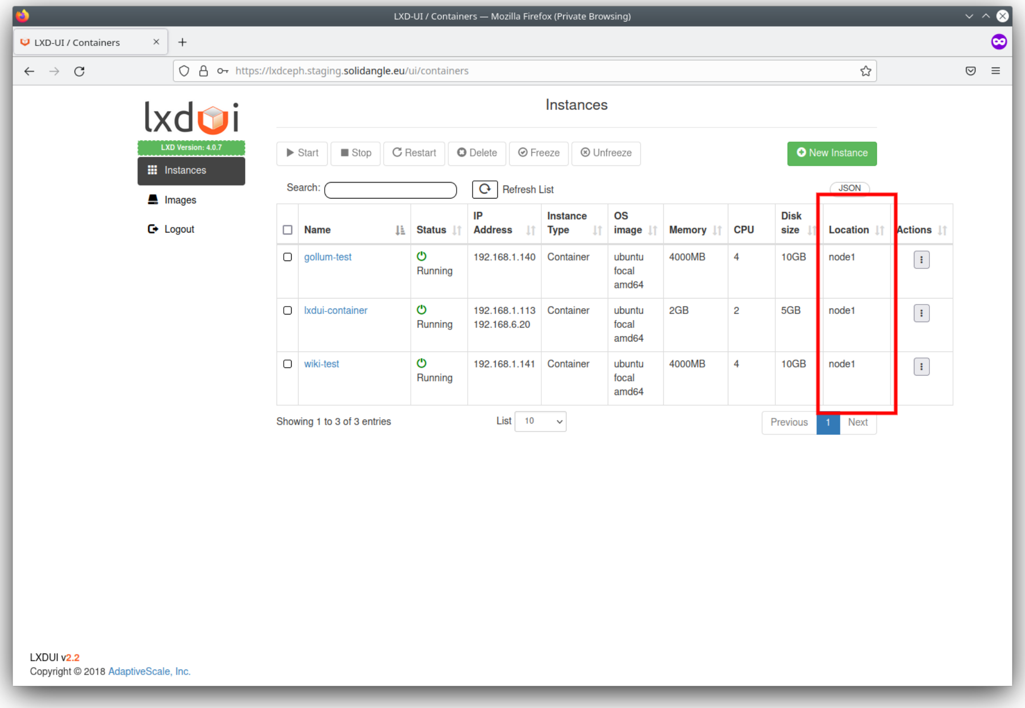
Instances

Name	Status	IP Address	Instance Type	OS image	Memory	CPU	Disk size	Actions
lxdui-container	Running	192.168.1.111	Container	ubuntu focal amd64	2GB	2	5GB	⋮
test-container	Stopped	N/A	Container	ubuntu focal amd64	1024MB	1	5GB	⋮
test-ui	Running	192.168.1.150	Container	ubuntu focal amd64	2GB	2	10GB	⋮

Showing 1 to 3 of 3 entries

LXDUI v2.2
Copyright © 2018 AdaptiveScale, Inc.

Figure 3.2: Original LXDUI interface



Instances

Name	Status	IP Address	Instance Type	OS image	Memory	CPU	Disk size	Location	Actions
gollum-test	Running	192.168.1.140	Container	ubuntu focal amd64	4000MB	4	10GB	node1	⋮
lxdui-container	Running	192.168.1.113 192.168.6.20	Container	ubuntu focal amd64	2GB	2	5GB	node1	⋮
wiki-test	Running	192.168.1.141	Container	ubuntu focal amd64	4000MB	4	10GB	node1	⋮

Showing 1 to 3 of 3 entries

LXDUI v2.2
Copyright © 2018 AdaptiveScale, Inc.

Figure 3.3: Original LXDUI interface

3.3 Windows Instances

LXD's 4.0 release introduced Virtual Machine creation and management, which was used for this project to create both Ubuntu 20.04 and Windows VMs [8]. While Ubuntu VM creation is straightforward as the images are hosted on LXD's image server, Windows images on the other hand require a more manual setup.

Since Windows is not natively supported by LXD, transferring and configuring the Windows image to a working state had to be done manually.

To configure a working Windows image, two files had to be transferred: the official Windows Server ISO image and the Windows *Virtio* drivers. Virtio allows for any operating system to be virtualized by using KVM as its virtual driver [18]. This allows us to create Windows VMs on our Ubuntu 20.04 physical host.

After configuring our first instance with a Windows image and the updated Virtio drivers, LXD is used to temporarily access the instance's graphical console. The following functions had to be configured through the graphical console so that the Windows image can be published:

- SSH access the Windows instance's CLI remotely
- RDP (Remote desktop protocol) to access the Windows instance's graphical console remotely by using the instance's IP address and use credentials

With the image configured and ready to be published, we then execute the *sysprep* tool that is installed by default on Windows Server images. Sysprep generalizes a Windows installation by removing any credentials, product keys or any unique identity data associated to the instance, while maintaining system configuration options [19]. Instances created with the sysprep treated Windows image will have SSH and RDP configured by default, but will lack any product keys or user information that was given during the initial setup.

After preparing the Windows image with the sysprep tool, it was then exported as a tar archive using LXD's publish command. This tar file containing the prepared Windows image can be imported to any LXD installation and be ready for use. For legal reasons, the image for this project was hosted on a password protected URL to prevent public distribution of Windows images.

3.4 Network Configuration

The network configuration in the system's environment will include both a LAN and a private network. In order for containers to access servers on the same network as the host machine and vice versa a bridge interface has to be configured. The private network will only be used for communication between the instances and the host machine itself and is inaccessible to any other machines.

This section details the configuration of both networks.

3.4.1 LAN configuration

In a production environment, LXD instances would be created on multiple physical servers. One of the main requirements to take into account is the access to all the instances created on the network by all the physical servers on the same network. In order to satisfy this requirement, a *network bridge* was created. A network bridge connects two or more networks, or segments of the same network [20]. In the use case in question, it will be used to connect all the LXD instances of a physical host to its network, assigning an IP address via DHCP (Dynamic Host Configuration Protocol), if available, to each instance by default. This allows for all other hosts on the same network to connect to or access any running instance on another host as if they were independent hosts.

In figure 3.4 an example of a typical network bridge interface is shown. To create the interface `br0` a *gateway* address must be provided, which will be used to perform communication between the local network and other external networks.

In this example, consider the presence of a DHCP server on the network. An instance created using the `br0` interface will send a DHCP discover message across the network through the physical host's NIC (Network Interface Controller). The DHCP server will then offer an unused IP address to the instance which will eventually be accepted, assigning the address to the instance [21].

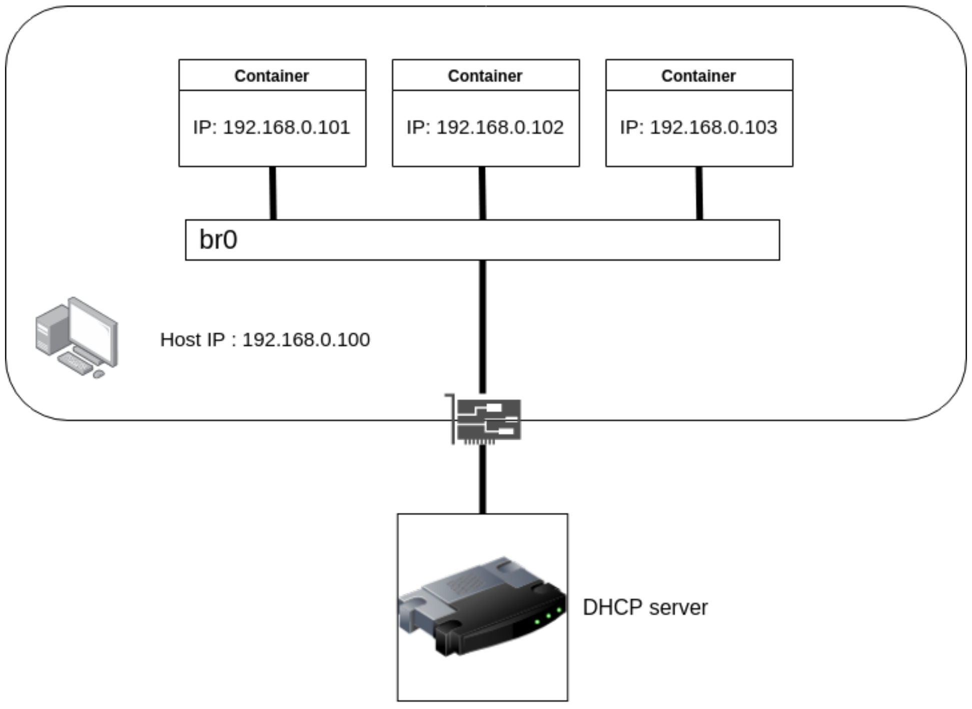


Figure 3.4: Network bridge example

As all physical servers will run Ubuntu 20.04 as the base operating system, *Netplan* was used to effortlessly create the bridge interface through a configurable YAML file. The bridge interface is automatically created with Puppet through a custom module developed for this project as will be later shown.

3.4.2 Private Network configuration

Another network requirement on a production server was the option to use a private network which would only be accessible to the host and its instances. This network would be used for secure communication between instances and the host, without forwarding traffic to the LAN.

LXD allows for the creation of private bridged networks, which contain their own virtualized DHCP servers [22]. These networks are managed by LXD, meaning that they can be configured through the command line to, for example, set the IPv4 address range for the bridge network or disable IPv6.

A private network can be created through the command line as follows:

```
lxc network create lxdbr0 ipv4.address=10.20.30.0/24
```

This network device can then be attached to any instance, assigning a private network IP address. An instance can have both a LAN and a private network IP address assigned at the same time.

3.4.3 Static IP

Apart from a randomly assigned IP address through a DHCP server, another requirement was to have static IP addresses assigned to some instances. This is crucial for services where the assigned address must be known beforehand so access to the service is deterministic. Example services include:

- File server
- Mail server
- Container hosting LXDUI

Since all Linux images used in this project are based on Ubuntu, Netplan was used to configure the network of all Linux instances.

Windows instances cannot be managed directly by LXD therefore it was not possible to configure a static IP address for these instances using the command line.

As previously mentioned Netplan is a network configuration tool which became Ubuntu's default network manager from Ubuntu 17.10 [6]. Netplan facilitates the configuration of a host's network through multiple configurable YAML files. This makes it useful when a host with multiple network interfaces, say both a LAN and a private network interface, has to be configured with a static IP address on either one or both of its interfaces. Since

Netplan accepts multiple YAML files, two separate configuration files can be created, one for each interface, simplifying network management.

Netplan configuration files are generated and applied through two *bash-scripts*, one for the LAN and another for the private network, which were specifically developed for the use case in question. These scripts are automatically executed if the static IP address option is chosen in the Puppet declaration. Below is an example of a Netplan network configuration file that sets a static LAN IP address:

```
network:
    version: 2
    ethernets:
        eth0:
            addresses:
            - 192.168.1.50/24
            nameservers:
                addresses:
                - 8.8.8.8
```

Listing 3.1: Netplan network configuration file for a static LAN IP address configuration

3.5 Instance configuration

LXD introduces the concept of *Profiles* which simplifies instance configuration, namely network interfaces and storage. A profile allows us to set a group of configuration options (for example: network, memory, storage pools, among others) which can then be applied to any instance. Instances using the same profile will have the same settings applied.

The network requirements for all instances are as follows:

- Access to LAN
- Access to private network
- Access to both LAN and private network

This project required profiles for both container and VM instances. The main difference between container and VM profiles is the disk initialization. VMs require their disks to be initialized by `cloud-init` so they are in working order.

A total of six profiles were created, three for each instance type, according to the above mentioned network options. All the instances using any of the profiles will be assigned, by default, an IP address through DHCP.

Below are two examples of both container and VM instance profiles that will assign a LAN IP address to either instance type:

```
config:
user.network-config: |
        #cloud-config
        version: 1
        config:
          - type: physical
            name: eth0
            subnets:
              - type: dhcp
                ipv4: true
description: Profile to be used for a container on the LAN.
devices:
  eth0:
    name: eth0
    nictype: bridged
    parent: br0
    type: nic
name: container-lan
used_by: []
```

Listing 3.2: Container profile example

```
config:
user.network-config: |
        #cloud-config
        version: 1
        config:
          - type: physical
            name: enp5s0
            subnets:
              - type: dhcp
                ipv4: true
description: Profile to be used for a VM on the LAN.
devices:
  config:
    source: cloud-init:config
    type: disk
  eth0:
    name: eth0
    nictype: bridged
    parent: br0
    type: nic
name: vm-lan
used_by: []
```

Listing 3.3: VM profile example

The profile of an instance can be changed according to network needs, switching between the three possible network configurations listed above.

All created profiles configure an instance's network interfaces with `cloud-init` as otherwise the network interfaces would not be detected correctly on instance creation or profile change.

While one could add a source LXD pool as a disk device on the profile, for the project this configuration was left out. The decision to add the storage pool manually to the instance came from the fact that LXD treats devices (network interfaces and storage disks) from profiles differently from manually added devices. For example, to change the disk size of a device added through a profile for the first time, we first have to *override* the device so that the contents of the device are copied to allow for changes in the configuration keys.

This additional overriding step adds an extra layer of complexity for day to day operations which can be eliminated if devices were to be added manually. The procedure to change the disk size of a profile added disk device is as follows:

1. If the disk's capacity is being changed for the first time, it must be overriden
2. After the first overriding step, the disk's capacity can be freely changed

A manually added disk device's capacity change procedure is as follows:

1. On instance creation, specify a storage pool to host the disk device (if none is provided, LXD will throw an exception as instances cannot be created without a disk)
2. The disk's capacity can be freely changed

By manually adding the disk device, the process to change the storage capacity will be identical for the first time it is done and any subsequent capacity changes, eliminating the initial *if* condition for a profile added disk device.

3.6 Filesystem choice

One of the main aspects that was taken into consideration when developing the system was choosing the right filesystem backend to be used by LXD instances.

The main filesystems recommended by LXD's authors are: *ZFS* [4] and *btrfs*[5] [23]. During filesystem performance testing, multiple issues were noticed, some directly related to performance and others due to the filesystem's default behavior.

These issues made the recommended filesystems inadequate for a production setup, thus another option for the filesystem was equired. Our final choice, *LVM*, was a filesystem manager rather than a filesystem itself. LXD can use LVM to create an *ext4* filesystem

[4]`https://openzfs.org/wiki/Main_Page` (Accessed on: January 13, 2021)

[5]`https://btrfs.wiki.kernel.org/index.php/Main_Page` (Accessed on: January 13, 2021)

partition to be used for instances. LVM, while listed as supported storage backend [23], is less recommended than both ZFS and btrfs since it does not provide optimized image and instance transfer. This is not an issue for a single server system since instances will rarely need to be migrated from the host.

In section 5.1 a more in-depth explanation of the tests performed and the respective results will be presented. This section will give an overview of the issues found during the testing phase.

3.6.1 First choice - ZFS

Initially, the system was tested using ZFS as the base filesystem for LXD instances as this was the main filesystem recommended by the authors. However, during the performance test phases, inconsistencies were noticed in the reported results and performance issues due to ZFS and its incompatibility with the server's hardware setup, namely the *RAID* 6 configuration of the storage disks, made it inadequate for a production setup.

RAID (Redundant Arrays of Inexpensive Disks) guarantees reliability or performance when using hardware disks to store data. Reliability and performance levels differ with the RAID level [24]. At Solid Angle, RAID 6 is used for production machines as it is an ideal trade-off between reliability and performance. A RAID 6 array allows for a maximum of 2 simultaneous drive failures until data is compromised. This is possible by calculating and stripping two sets of parity bits across all disks on the array which can be used to reconstruct lost data from both drives [25]. The main cost for RAID 6 is reduced disk capacity as the capacity of two disks will be used for storing the parity bits and reduced performance on write operations as parity bits must be calculated when writing data to the array to guarantee data safety.

The disk read and write performance results from the initial tests between container and VM instances were inconsistent and higher than the native performance. These results surpassed the physical capabilities of the underlying hardware disks and were inconsistent when running consecutively. This inconsistency was found to be caused by the native design of ZFS and its use of a modified version of *ARC* (Adaptive Replacement Cache).

ZFS is a filesystem the allows for dynamic disk space allocation and simplifies the creation of storage pools using one or multiple hardware storage devices [26]. This is one of the main recommended filesystems for LXD as it is highly versatile when it comes to storage pool creation and also allocates disk space dynamically. Dynamic disk space allocation implies that an instance created with a storage limit of 100GB will not have that amount of space allocated on creation time, rather it will only take up as much space as is currently being used until it reaches the 100GB limit. This significantly decreases instance creation time, as a large portion of time is taken up by disk space allocation.

As stated before, ZFS uses a modified version of ARC to improve its performance. ARC is based on the existence of two cache lists, one that captures recency, and another one that captures frequency, the total capacity of pages stored within both lists is the total ARC size [27]. The modified version of ARC developed by the *OpenZFS* community adds the following improvements to the original algorithm [28]:

- Added a locking mechanism for blocks in cache, making them un-evictable from cache so long as there is an active external reference to the blocks meaning the data may still be used in the near future
- Dynamic cache size that grows or shrinks according to the current need
- Dynamic page size so that space in cache can be freed more efficiently whenever there is a cache miss, resembling the space used by the new block as close as possible

To prevent the cache memory from affecting performance results, it was cleared before every file I/O benchmark run by executing the following commands:

```
        sync; echo 3 > /proc/sys/vm/drop_caches
```

Clearing the cached content stabilized performance results, however another issue was found with the reported results. Both native and LXD instances ZFS performance were significantly slower than that of the native ext4. This sharp decrease in performance made ZFS inadequate for a production environment. The official documentation showed that ZFS does not work properly when coupled with hardware RAID controllers, which is the setup for production servers [29].

The main issue was found to be the ZFS write performance on a RAID 6 setup, as it is affected by *partial stripe writes* [30, p. 14]. This phenomenon occurs as the hardware disk's sector size is not reported correctly to ZFS when using hardware RAID 6. When writing new data to the disk, if the data does not fill a whole stripe, the old data must be read with the corresponding parity block to compute a new parity block. The new data and the computed parity block are then written to the disk creating multiple disk accesses on partial stripe writes.

Due to the previously mentioned issues, ultimately ZFS had to be replaced with another filesystem that supports a production server's hardware configuration.

3.6.2 Second choice - btrfs

The main issue found with btrfs is the lack of a mount point for every sub-volume [31]. This implies that container instances created with a btrfs backed storage pool show the host pool's total capacity rather than the quota assigned to it. This is an issue as no isolated virtualized instance should have information regarding its hosts resources. Wrongfully reported disk capacity values within a container may also lead to errors in software when the real capacity of allocated space must be known, as the reported values would be much larger than the real quota set to the container. VMs are not affected by this issue as their disks are fully virtualized and initialized with `cloud-init`, displaying the correct quota values.

3.6.3 Final choice - LVM

The final storage backend to be tested was LVM (Logical Volume Manager). LVM simplifies the creation of logical partitions on hardware disks by managing the following 3 concepts [32]:

- `Volume groups` - a named collection of *physical volumes* and *logical volumes*
- `Physical Volumes` - hardware disks or disk partitions that store logical volumes
- `Logical volumes` - similar to a disk partition, logical volumes hold a filesystem, however they can also span across multiple physical volumes unlike traditional disk partitions

On a production server, a logical volume will be created on the array of physical disks which will use the remaining space after the base Ubuntu installation. This logical volume will be used as the base storage pool for all LXD instances.

While LVM is not a filesystem, LXD can use it to manage a device partition and create an ext4 filesystem on it.

LVM is less recommended than both ZFS and btrfs as it does not support optimized image and instance transfer [6]. However this issue will not affect single virtualization servers since instances will not be regularly transferred from host to host.

As expected, the LXD created LVM partition's performance is nearly identical to that of the native ext4 performance since LXD uses LVM to manage the logical volume creating an ext4 filesystem on it, identical to the native filesystem. The performance overhead for both containers and VMs was within an acceptable margin when compared to ZFS' performance.

As such, LVM solves the issues found on the previous tested filesystems:

- It has acceptable performance for a production environment, significantly better compared to ZFS with the server's RAID 6 disk setup
- It allows for separate mountpoints for every sub-volume created inside the main storage volume. This makes it possible to view only the allocated disk space to that instance whether it be a container or a VM, which fixes the issue regarding btrfs sub-volume management

[6]`https://lxd.readthedocs.io/en/latest/storage/#storage-backends-and-supported-functions` (Accessed on: January 29, 2021)

3.7 Puppet configuration

With the system's base requirements chosen, the automation of the installation procedure was followed. This was required as many resources (namely LXD storage pools, profiles and network configuration), if done manually, not only consume a large portion of a system administrator's time, but are also prone to human error. If any resource is setup with the wrong values, the final system may be incoherent and unusable.

By using Puppet as a system configurator, one can minimize human interaction during system setup and thus decrease the chances of an incorrect setup due to human error. For the base LXD installation and setup, the `is_lxd` module was created containing the following classes:

- `lxd_base`
- `lxd_network`
- `lxd_server_base`
- `lxd_server_ui`
- `lxd_standard_images`
- `lxd_windows_images`

The `lxd_base` class is used for installing or updating the LXD version through the *Snapcraft* package manager. This class uses a modified version of the `ethanhs-snapd` [33] Puppet module which not only allows for a fresh LXD installation, but also a version upgrade of an existing LXD installation.

The `lxd_network` class configures the previously mentioned bridge network on the host machine so that container and VM instances can be assigned LAN IP addresses. It is also responsible for creating LXD's private network.

For the host's base LXD configuration, the `lxd_server_base` class is used. This is the main class that uses both the `lxd_base` and `lxd_network` classes to install LXD and setup the host's network, and creates all the resources needed so that the host is in a functioning state after the Puppet run. These resources include: the main storage pool to be used by all instances, instance profiles, package dependencies and both utility and service scripts.

Since LXD's graphical interface (LXDUI) is to be installed in a container, it was also necessary to automate its installation, which is done through the `lxd_server_ui` class. This class will download the required dependencies for LXDUI, including the source code, compile the UI and connect to the host as a remote LXD server, which makes it possbile to access the host's instances through the container.

The `lxd_standard_images` and `lxd_windows_images` classes are responsible for transferring both Linux and Windows based images. These are separated since Windows images are optional in a production environment.

Apart from the newly implemented Puppet module, the following modules were also modified from their original implementation:

- ethanhs-snapd [33]
- lxd-puppet-module (by OVH) [34]

For the `ethanhs-snapd` module, a version selection option was implemented so that it was possible to upgrade LXD's version on a physical host through Puppet. The original module only allowed for version selection on install and did not support version upgrades for already present snap packages.

While OVH's LXD Puppet module facilitated the development of the project, it lacked some core features that were required for the project. One of the main features was VM support. The original module was using outdated LXD API calls which did not allow for the creation of LXD VM instances. Another required feature was the download of LXD images from password protected URLs. This was needed as the Server image was hosted on a password protected URL so as to not publicly publish a prepared Windows image, preventing any legal issues.

The instructions for setting up both the host server and the UI container through Puppet can be found on the project repository's wiki page [2]. While some manual setup is needed, mainly configuring the Puppet node declaration with the correct variable values, the effort needed to configure a host from root is significantly lower when compared to the manual setup.

Below is an example Puppet declaration of an LXD instance:

```
lxd::container { 'ubuntu-container':
        state           => 'started',
        type            => 'container',
        profiles        => ['container-lan'],
        config          => {
                'limits.cpu' => '2',
                'limits.memory' => '2GB', },
        devices         => {
                'root' => {
                        'path' => '/',
                        'pool' => 'default',
                        'size' => '10GB',
                        'type' => 'disk',
                },
        },
        image           => 'ubuntu2004',
        set_ip          => true,
        set_private_ip => false,
        ip_address      => "192.168.1.120",
        netmask         => "255.255.255.0",
        gateway         => "192.168.1.1",
        dns             => ["8.8.8.8","1.1.1.1"],
}
```

Listing 3.4: Puppet instance declaration example

The instance created with this declaration will use Ubuntu 20.04 as its base image, be assigned 2 CPU cores and 2GB of memory, get 10GB of disk space allocation and a static LAN IP address of 192.168.1.120. These values can be changed as desired and the instance will be updated whenever the Puppet agent is executed on the host.

Chapter 4

Clustered virtualization server development

Having finished, tested and released a stable implementation of a single server virtualization system, the next step was to investigate the settings aiming at a clustered LXD environment. The main objective for the research on clustering was to take advantage of the extra layer of redundancy and instance recovery that was provided when coupling LXD clustering with the *Ceph* object storage system.

4.1 LXD Clustering

LXD clustering allows multiple physical hosts to effortlessly communicate with each other through the network, sharing resources such as storage pools, profiles and instances [15].

LXD provides a unified command set to manage instances in both a single server and clustered setup. The only difference when managing instances on both scenarios is the addition of a `-target` flag that allows the user to choose on which node an instance is to be hosted on. The command line similarity between both setups simplifies the development and usage.

With the addition of clustering, the LXD instance creation procedure does not change as expected, since resource creation uniformity should be guaranteed in a clustered environment. Otherwise, the added layer of complexity when managing resources in a cluster may deem it inadequate to be used in a production environment. LXD implements an optional configuration option that allows the user to choose on which node the instance is to be hosted. If none is specified, LXD automatically creates the instance on the node with the least number of hosted instances.

Since it is possible to choose a specific node in a cluster where an instance will be created, we can manage computational resources for a much larger number of LXD instances, as the resources from multiple hosts in a production environment will surely be greater than a single host. When extra computational resources are needed, we can add a new node to the cluster to host the instances, a process that is simplified by LXD.

Clustering in LXD functions similarly to the remote concept discussed in section 3.2, by exposing the port 8443 to the network and allowing new hosts to join the cluster so long as they provide a valid password, which is defined during cluster creation, and the cluster certificate.

4.2 Network requirements

4.2.1 LXD instance network

In a clustered environment, the network configuration for LXD instances remains unchanged and the previously developed instance profiles can be reused for this development phase.

In order for network profiles to function properly in a clustered environment, bridge network interface names would have to be identical across all nodes in the cluster. This is needed since profiles attach network interfaces to instances by using the node's bridge interface as a parent. For example, the profiles assigning a LAN interface to an instance would use the node's `br0` interface as the parent. This interface name would have to be identical on all nodes so instance creation does not fail due to using a non existent interface name.

Since the private network is managed by LXD, we simply need to create an identical private bridge interface (mainly configuring an identical IPv4 subnet) for every node in the cluster. The main caveat when it comes to private networks in a clustered network is the fact that two instances running on a private network on two different hosts cannot communicate with each other even though their IPs are shown in the same subnet. This is due to the fact that private networks are hosted locally on each host, meaning that only the host and instances created on the same private network can communicate with each other. Instances can access the Internet and the LAN as LXD creates a NAT (Network Address Translator) by default when creating a private network [22], yet other nodes on the LAN are unable to connect to the node's instances on the private network.

4.2.2 Node network

In the scenario presented, the Ceph and LXD cluster would be configured in a data center environment. The hosts on the datacenter would be assigned leased public IP addresses where traffic to and from the addresses requires monetary payment. The public network may also be limited to less than 1 Gigabit per second ethernet connection which is less than the minimal recommended settings for a Ceph cluster [35]. To reduce costs and guarantee cluster performance a secondary private network is needed for both Ceph and

LXD communication.

Ceph is highly network dependent as the data replicated to disks on other nodes in the cluster is done through the network. Having a separate private network dedicated to data replication will greatly reduce the traffic on the public network, thus reducing costs. It will also allow us to control the network speed by replacing network hardware as best fits the current needs, which is not possible on the public network as the network speed would be dependent on the service provider.

LXD in comparison to Ceph is not as network heavy. However, since it is not essential for the LXD daemons in different nodes to communicate on the public network, it can be setup so that LXD communicates on the private network instead, further decreasing the traffic on the public network.

In figure 4.1 an example of the previously described network configuration is shown. Users will access the nodes or instances through the public network while the private network will be exclusively used for Ceph and LXD traffic.

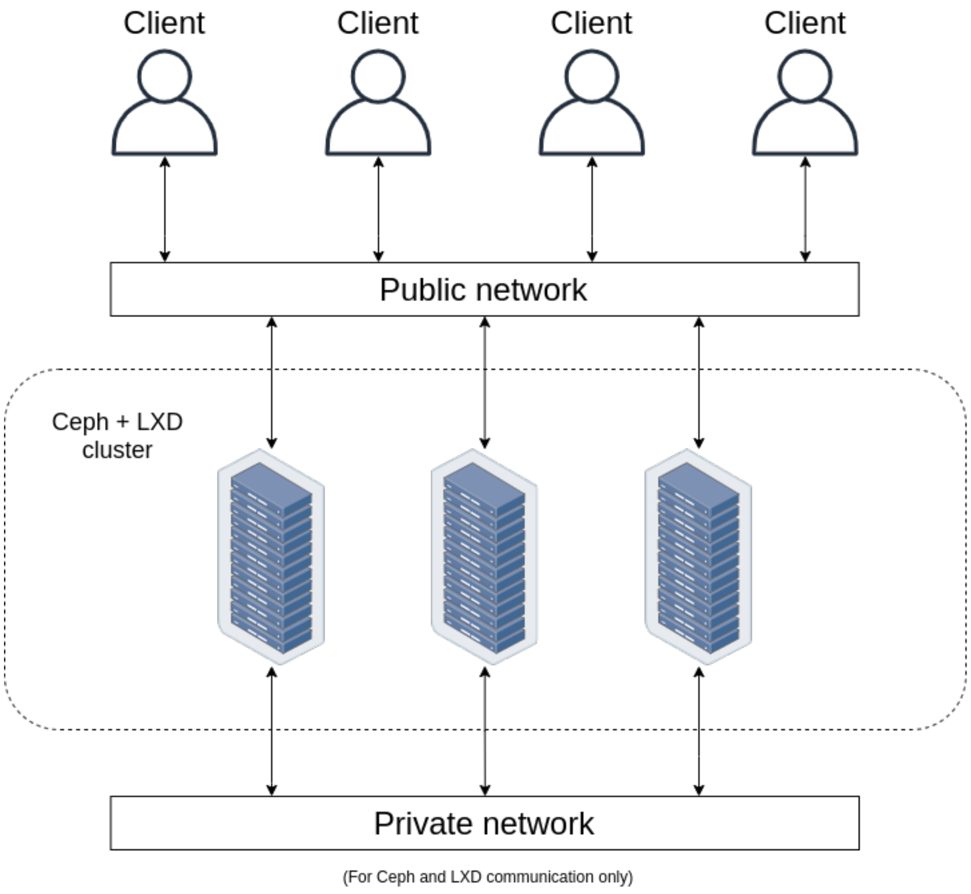


Figure 4.1: Ceph and LXD cluster network example

4.3 Ceph storage

While one could use local storage backends, namely ZFS, btrfs and an LVM managed ext4 partition, using them would limit the capabilities of LXD clustering as the storage pool would be tied to a specific node. This would mean that if a node in the cluster hosting instances loses connection to the cluster, whether due to maintenance purposes, network errors or hardware errors, instances hosted on that node will also become inaccessible until the node rejoins the cluster.

This scenario would be identical to a single server setup in which, if a host fails, all hosted instances fail and are unrecoverable. To guarantee redundancy among nodes we must then choose a storage backend that replicates data across the nodes in the cluster, rather than just expose the data available in the local storage pool. To achieve this, the *Ceph* storage backend was used.

Ceph storage clustering guarantees data availability and redundancy by replicating data written to disk across the network to other participant nodes in the cluster. This is done through multiple Ceph services that run in the background and communicate with each other through the network.

4.3.1 Terminology

Below are Ceph specific terms that will be further used to describe the storage backend [36]:

- MON (Monitor) service - this daemon is crucial for the Ceph cluster to function correctly. It maintains maps of all Ceph services to ensure they are all up in working order and coordinates host authentication to cluster. A cluster should have at least three Ceph monitor daemons running to guarantee redundancy.
- MGR (Manager) service - the manager service offers monitoring data regarding storage usage and the general state of the cluster. It also offers a graphical dashboard to monitor the state of the cluster.
- OSD (Object Storage Daemon) service - this is the main storage service that manages reading and writing data to disk as well as data replication. It can be seen as Ceph's storage devices.

4.4 Ceph cluster setup

Prior to configuring the Ceph storage cluster, an installation tool had to be chosen for the process as multiple forms of cluster configuration are made available [37].

While a Puppet module for Ceph has been developed [1], it uses an outdated Ceph

[1] `https://github.com/openstack/puppet-ceph` (Accessed on: 26 April, 2021)

deployment method where each service has to be explicitly defined and manually configured as separate services. While a Ceph cluster can be configured in the described way, it complicates cluster management as each individual service would have to be individually managed through Puppet instead of it being automatically configured.

The chosen tool to setup a storage cluster was *Cephadm*. Cephadm accesses remote cluster nodes through SSH to install and configure all of the Ceph related services. Cephadm also automatically detects the number of cluster members currently present and deploys the necessary services for the given cluster size, mainly MON and MGR services to guarantee cluster availability and redundancy [38] and is one of the main recommended tools to setup a Ceph cluster [37].

Setting up a Ceph cluster consists on the following steps [39]:

1. Running the Cephadm *bootstrap* on the first cluster member. This will initialize the cluster, Ceph services, create the configuration files and the Ceph public key to SSH into future nodes.
2. After bootstrapping the cluster, new hosts can be added to the cluster by copying the generated SSH public key to said hosts, allowing SSH access to Cephadm as the root user which configures all the necessary Ceph services
3. Finally we can initiate the desired disk partitions or storage devices on all hosts as OSDs, which allows Ceph to use device partitions or whole storage devices to read and store data as well as replicate the data in the storage pool.

Ceph OSDs can be seen as the storage devices and the storage pools can be seen as device partitions in which data will be stored. These pools are created by default with a replication factor of 3, meaning that the data written to OSDs is replicated twice across all available OSDs on the network [40]. Having 2 replicas of any data stored on OSDs also means that the total storage capacity is divided by 3 to accommodate the replicated data.

During the testing phase of Cephadm, multiple issues were found in the documentation and added to the project's repository wiki [2]. Below are some of the main issues found during the manual setup following the official Cephadm documentation:

- While there is mention that Cephadm will access the hosts through SSH as the root user to install and configure Ceph services, this is not directly documented on the default Ceph cluster bootstrap instructions but rather under the *Further information about Cephadm bootstrap* section [3]. This information is easily missed when consulting the documentation which could lead to a failed cluster creation process and must be explicitly stated.

[2]`https://bitbucket.org/asolidodev/ironman/wiki/LXD%20&%20Ceph%20installation%20and%20setup` (Accessed on: 26 April, 2021)

[3]`https://docs.ceph.com/en/latest/cephadm/install/#further-information-about-cephadm-bootstrap` (Accessed on: 8 May, 2021)

- While not directly related to the Ceph documentation, LXD uses a default built in version of Ceph that is used whenever a Ceph storage pool is created. This leads to failures in creating the storage pool as the cluster information is not present in the built in Ceph installation's directory. In order to use the manually installed version of Ceph using Cephadm, the LXD snap configuration option `ceph.external` must be set to true [41].

Other minor details regarding a Ceph and LXD cluster setup from root were also documented on the wiki which was used as a point of reference for the development of the Puppet module designed to automate the setup process, similarly to what was done when developing a single virtualization server.

4.5 LXDUI adaptation

Since LXDUI was not developed for a clustered LXD environment, some adaptations had to be made to the previously modified version of the interface. These adaptations included:

- Instance location column that shows in which node an instance is hosted on
- Instance migration option to migrate the instance between hosts in the cluster

Since these adaptations are not used for a single server setup, the installation of the UI had to be intercompatible between a clustered and a non clustered setup. To satisfy this requirement, the Puppet class used to install and configure LXDUI performs an environment check, replacing and editing the necessary files to expose the newly implemented features on a clustered environment or installing the default interface for a non clustered environment.

The network interfaces for LXDUI also had to be changed for a clustered environment. A public network interface is needed so the UI is accessible to users and a private network interface so we can add a cluster node as a remote LXD server for the LXDUI container. A specific profile and static IP script was created for the LXDUI container in a clustered environment to configure the necessary network changes.

4.6 Puppet development

Prior to starting the Puppet module development to automatically configure a Ceph and LXD cluster, cluster dependencies, both in the form of software packages and configuration files, had to be ordered and mapped. This step is crucial as any out of order configuration may lead to errors in the bootstrapping phase of the cluster leaving it in an incoherent state.

The cluster nodes were separated into two different categories for the Puppet module development: *bootstrap* and *regular* nodes. In a cluster setup there must only be a single bootstrap node that will initiate both the Ceph and the LXD cluster and allow access to a pre determined list of regular nodes to each cluster.

Below are the two main dependencies to be taken into account:

- Cephadm requires passwordless root user SSH connection to the target hosts in order to install and initiate all Ceph related services
- regular nodes can only join the LXD cluster after they have been added as part of the Ceph cluster. This requirement comes from the fact that to join an LXD cluster we must specify the source of the remote storage pool, which in this case would be a Ceph OSD pool requiring previous configuration.

These dependencies stand out from the rest as they break the flow of a Puppet execution, meaning that the cluster cannot be configured with a single Puppet agent run on each node. A bootstrap node configures both Ceph and LXD clusters, but cannot add hosts (regular nodes) to said clusters since passwordless root SSH access has not yet been granted. The regular nodes cannot join the LXD cluster since they have not yet joined the Ceph cluster during its first Puppet run. In figure 4.2 a graphical representation of the dependency map is shown. In comparison to a single virtualization server Puppet setup, a clustered setup is less streamlined since one cannot configure the cluster by executing the Puppet agent only once on each node.

Below are the main classes developed for the clustered LXD environment:

- `bootstrap_node`
- `dependencies`
- `host`
- `regular_node`

The `bootstrap_node` class is used to configure the bootstrap node, by installing Ceph, LXD and their dependencies, setting up the required network configurations (namely the bridge network to be used by LXD instances) as well as initializing both clusters if they are not yet initialized.

The `dependencies` class installs all package dependencies for both Ceph and LXD and is used by both the bootstrap and regular nodes.

The `host` class is used on the bootstrap node's Puppet declaration to add regular nodes as Ceph cluster members and to copy the LXD cluster join bash script to the same nodes. The bootstrap node will only attempt to add the node as a host or copy the script if SSH access has already been granted.

The `regular_node` class is used by all regular node Puppet declarations to install the package dependencies and add the SSH key generated by the bootstrap node to the

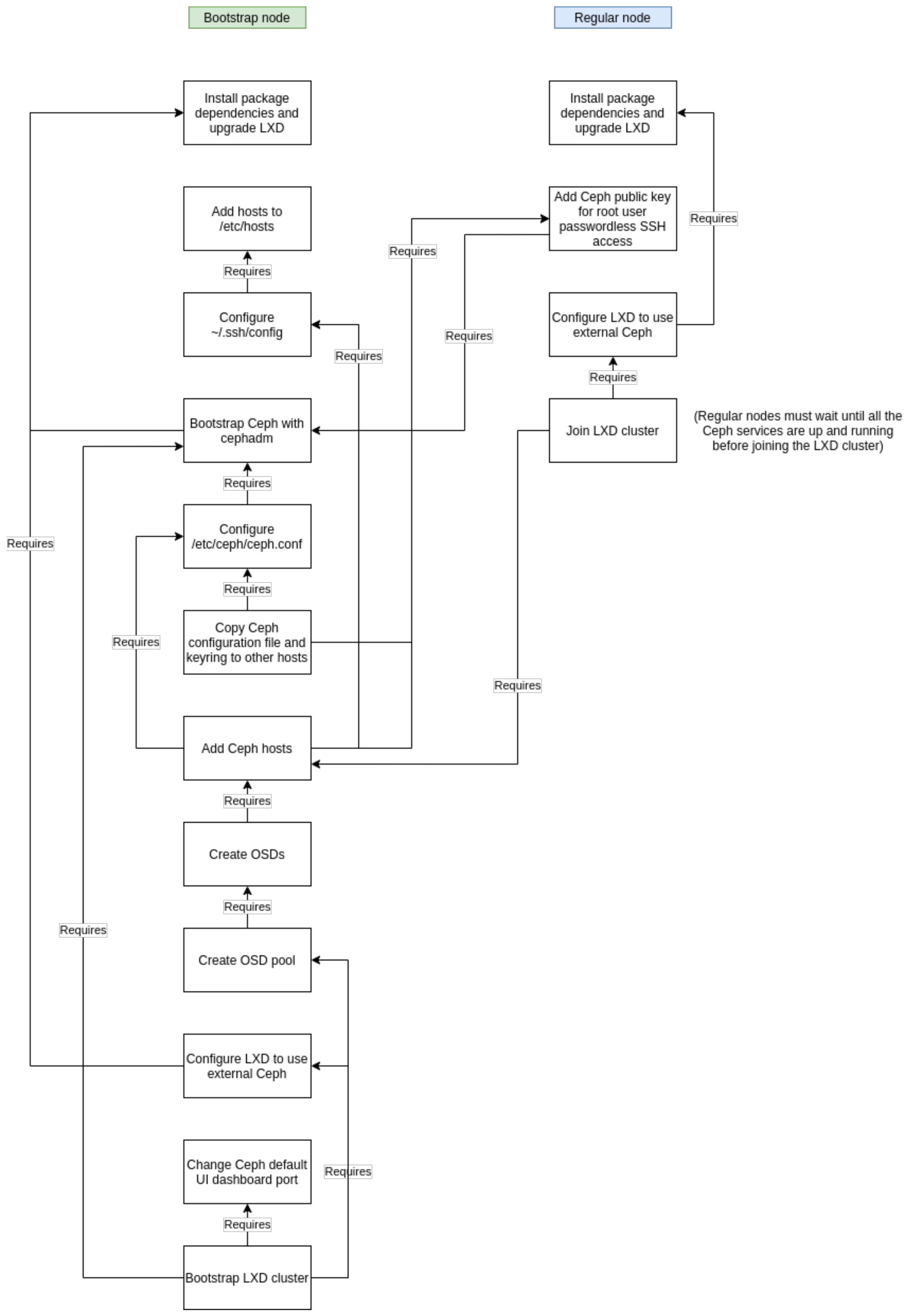

Bootstrap node
Regular node
Install package dependencies and upgrade LXD
Add hosts to /etc/hosts
Configure ~/.ssh/config
Bootstrap Ceph with cephadm
Configure /etc/ceph/ceph.conf
Copy Ceph configuration file and keyring to other hosts
Add Ceph hosts
Create OSDs
Create OSD pool
Configure LXD to use external Ceph
Change Ceph default UI dashboard port
Bootstrap LXD cluster
Install package dependencies and upgrade LXD
Add Ceph public key for root user passwordless SSH access
Configure LXD to use external Ceph
Join LXD cluster
(Regular nodes must wait until all the Ceph services are up and running before joining the LXD cluster)
Requires

Figure 4.2: Ceph configuration dependency map

`authorized_keys` file so root SSH access is granted. It will attempt to join the LXD cluster by executing the previously mentioned script if present.

The Ceph and LXD cluster Puppet module uses the previously developed LXD module to configure any LXD related settings so as to not repeat Puppet code for this development phase.

4.6.1 Puppet execution

Due to the two way dependency between bootstrap and regular nodes, the Puppet agent runs on each node will have to be executed multiple times in phases as opposed to a single Puppet agent run in the case of a single virtualization server setup. Each separate execution will configure the node until it requires dependencies of another node.

Below is the list of the Puppet agent execution steps and the changes made on each run:

- **Bootstrap node first run** - On the first bootstrap node Puppet run, LXD and its dependencies will be installed and configured, the Ceph cluster will be initiated and the Cephadm SSH key pair generated. This key pair will later be used by Cephadm to establish an SSH connection to regular nodes as the root user.
- **Regular nodes first run** - Before executing Puppet on regular nodes, we must first manually extract the generated public key on the bootstrap node. This key will be placed in a global variable on the Puppet node declaration file and will be added to the `authorized_keys` file of every regular node, granting passwordless root SSH access to Cephadm. A regular node's first run will handle SSH access dependencies so the bootstrap can proceed to the next step, by adding the nodes to the Ceph cluster. It will also install and configure LXD and its dependencies.
- **Bootstrap node second run** - On the bootstrap node's second run, each node will be added as a Ceph cluster member and their storage disks (if declared in the Puppet node declaration file) will be initialized as OSDs. Alongside adding the hosts as a Ceph cluster member, the bootstrap node will also place a bash script in each regular node that will be used to join the LXD cluster and other files required for Ceph cluster access on regular nodes.
- **Regular nodes second run** - The second and last regular node Puppet run will execute the previously mentioned bash script on the regular nodes, adding them to the LXD cluster.
- **Bootstrap node third run** - The third bootstrap node agent run will download the base Linux images, initiate the LXDUI container and optionally download the Windows Server image.

After running the Puppet agent in the steps described above on all nodes, the Ceph and LXD cluster is fully configured and ready to be used.

4.7 Operational scenarios

To test the capabilities of a fault tolerant LXD cluster, operational scenarios were set to be tested in order to guarantee the reliability a Ceph and LXD cluster brings.

Below is a list of the tested operational scenarios:

- Instance migration between working nodes
- Instance recovery from faulty node
- Recovery of faulty node with a new server with fresh OS installation
- Node network loss and cluster rejoin
- Node maintenance and cluster rejoin simulation
- Bootstrap node root OS reinstall and cluster rejoin

Instructions to reproduce these operational scenarios were documented in the project repository's official Wiki page [2].

To migrate instances between nodes in an LXD cluster, we must specify the target node with the `-target` option. Instances must be stopped before migrating across nodes in a cluster. The commands used to migrate an instance are as follows:

```
# To stop the instance from executing
lxc stop instance-name

# Migrate the stopped instance to another host
lxc move instance-name --target node2
```

Recovering an instance from a faulty node that has lost connection to the cluster requires changing the instance's host from the faulty node to another working node. The procedure is identical to migrating instances between working nodes, the only difference being that the `-target` node must be a working node and the instance does not need to be stopped as it will have already stopped running since the host is offline. One should note that this is only possible by using Ceph as the storage backend so that the database is replicated across all nodes, meaning that no single node is the sole owner of a given instance's data. Were the LXD cluster to be using any other storage backend, all of the instance's data would be stored in the faulty node, remaining unrecoverable until the node reconnects to the cluster.

When a node fails and requires a fresh OS installation, it will have to be added as a new Ceph cluster member. While Cephadm handles new members by configuring and executing the required services for the host, it does not handle node removal from the cluster. Therefore, the process of removing a node from a Ceph cluster is fully manual. This process involves removing any references to services that were previously executing on the faulty node. These services can be: MON daemons, MGR daemons and OSDs.

To simulate node network loss, the ethernet cable was pulled from the physical host temporarily, until both Ceph and LXD clusters detect that the node is unavailable. Recovering instances from nodes that are temporarily off the network is identical to recovering instances from faulty nodes as they base off the same principle of accessing the instance's data from the distributed Ceph database. After the ethernet cable is reconnected and the node regains network connection, both Ceph and LXD clusters return to normal functioning.

When a node is shutdown, say for maintenance purposes, the scenario is identical to when it looses network connection. Since both Ceph and LXD do not distinguish the cause of a node becoming unavailable, instance recovery steps are identical to the ones previously mentioned.

To guarantee that the cluster does not rely on a sole node for it to function, the bootstrap node was removed, reinstalled and readded to the cluster. During the bootstrap node removal, the cluster remained functional and the node was detected to be offline by the cluster. This however did not hinder the normal functioning of the cluster as it was possible to transfer any instances from the bootstrap node to any other node still in the cluster since the database is replicated. The process of removing the Ceph related services on the bootstrap node was identical to other nodes tested on previous operational scenarios. In order for the replacement node to be added back to the cluster, a new bootstrap node had to be manually elected. This node would not actually bootstrap the cluster, but would be used to access other nodes through SSH as the root user to setup the hosts as Ceph nodes using Cephadm. The replacement node is then added back to the cluster as a regular node.

Chapter 5

Performance benchmarks

After finalizing the development of a single server and clustered virtualization system, multiple performance benchmarks were run to guarantee that the developed system is in working order. The goal is to learn the extent to which the use of LXD impacts the processing power, memory and disk IO operations in comparison with bare metal and virtualization alternatives.

5.1 Single server performance benchmarks

To benchmark both host and instance CPU and memory performance, `sysbench`[1] was used.

For IO benchmarks, `dd`[2] was used.

The host server used for benchmarking has the following hardware components:

- **CPU**: AMD Opteron(tm) Processor 6348
- **Memory**: 4x 8GB DDR3 1600Mhz, Hynix
- **Storage**: 4x Toshiba MG04SCA40EE (in a RAID 6 configuration)

Both container and VM instances used for benchmarking have the following specifications:

- **CPUs**: 4
- **Memory**: 8GB
- **Storage**: 50GB

5.1.1 CPU benchmarks

The CPU benchmark tests executed by sysbench are based on prime number calculation. A maximum number limit is set so when it reaches the set limit, it restarts the calculations

[1]`https://github.com/akopytov/sysbench` (Accessed on: March 1, 2021)

[2]`https://man7.org/linux/man-pages/man1/dd.1.html` (Accessed on: March 1, 2021)

from number 1. The CPU tests were limited to 4 threads, 2 minutes of total runtime and a prime number limit of 100,000. The performance is measured by the number of events (number of calculated prime numbers) executed per second. The tests were run 10 times on both types of instances and the host, with the average of 10 tests being presented in figure 5.1.

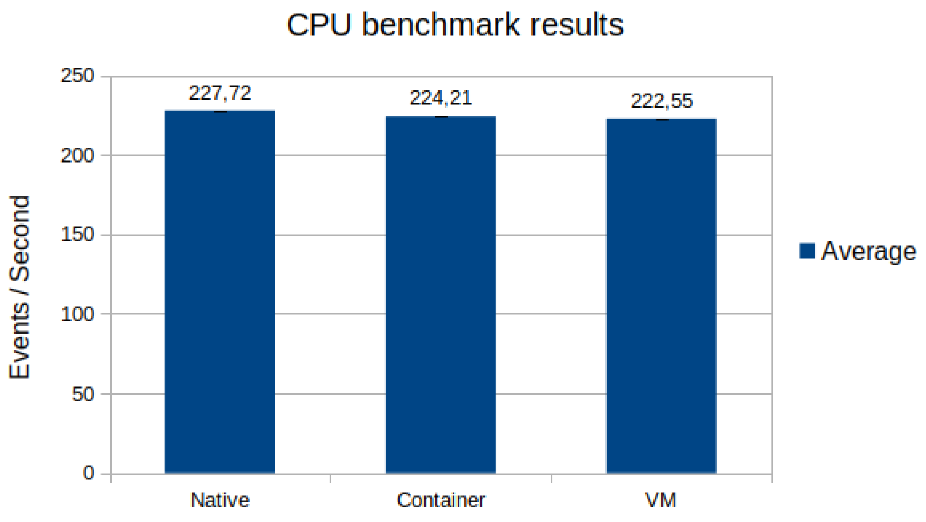


Figure 5.1: CPU benchmark results

Native CPU tests show a performance advantage by 1.54% and 2.3% when compared to container and VM tests respectively. These performance values are expected as LXC has less of an overhead than VM instances since they directly share the kernel with the host, leading to a decreased performance overhead.

5.1.2 Memory benchmarks

Memory benchmarks were performed by reading and writing 300GB worth of data from and to memory. Similar to CPU benchmarks, all 4 CPUs allocated to both container and VM instances were used to perform these tests. The host's performance test was also limited to 4 CPUs. The benchmark was executed 10 times for each one of the three test scenarios.

Test results for memory read benchmarks are shown in figure 5.2. Containers show an advantage over native performance by 0.4% while VM's have a performance decrease of 8.2%.

The marginal performance advantage of containers is mostly due to the performed test's limited sample size which has an affect on the average performance calculation due to performance fluctuation.

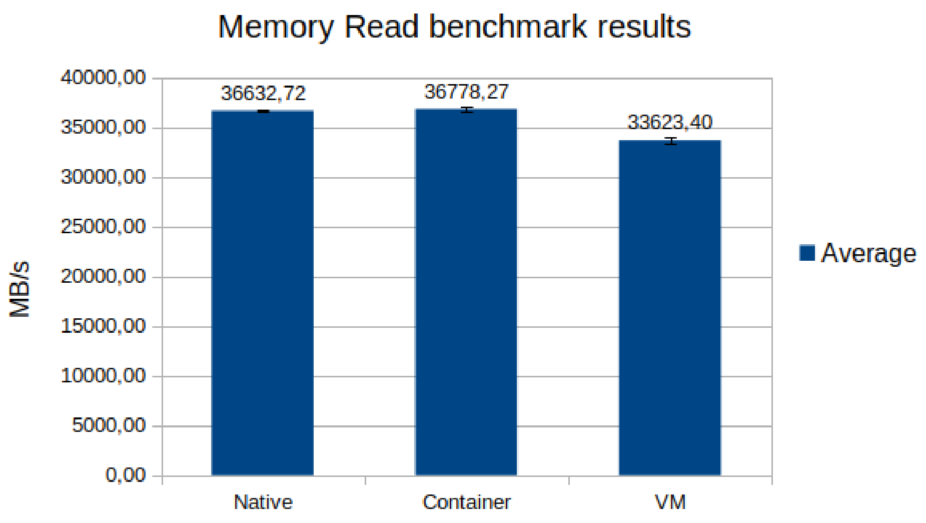


Figure 5.2: Memory Read benchmark results

In figure 5.3 the memory write benchmark results are shown. Both instance types have a comparable memory write performance as opposed to read performance in which the containers outperformed the VM instances. The native test shows a performance advantage over both instance types by 5.9% and 5.1% for the container and VM test cases respectively. In this test, the VM instances show a 0.9% performance increase over the container instance, which can also be attributed to the test's sample size. While containers show a lower average, the performance results have a larger fluctuation when compared to VM instances.

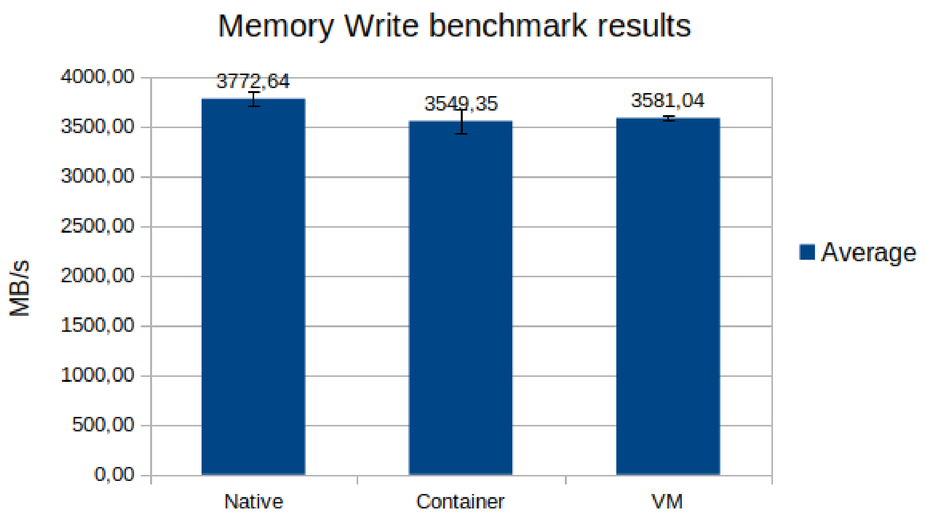


Figure 5.3: Memory Read benchmark results

5.1.3 IO benchmarks

To test disk IO performance, dd was used to create a random file, which is read then written to another directory, measuring mainly the disk's write performance. The copied file is then written to the `/dev/null` file, also known as the *null device*. Data written to this device is accepted, but immediately discarded, never written to the hardware disk. This makes write operations nearly instantaneous, meaning that disk read performance would be measured in this scenario.

Each read and write test is executed 10 times and the average values are presented in the graphs throughout this section.

In figure 5.4 a comparison is made between each filesystem's read performance on the host machine. While LVM is not a filesystem, the volume managed by it is formatted in ext4. We can see that both the host's root ext4 filesystem disk performance and LXD's LVM managed ext4 partition is nearly identical. This is expected as the root disk is also managed by LVM on OS installation. With this comparison, it becomes clear that the most significant performance difference is ZFS' compared to all other filesystems, where it achieves less than half of native ext4 performance. This is due to the host's hardware incompatibility with the ZFS filesystem which was mentioned in subsection 3.6.1 and made it inadequate for a production setup.

While a performance decrease of 12.8% can be observed when comparing the LVM partition's performance with btrfs' performance, the main issue of btrfs is the lack of individual mountpoints. This implies that the host's entire storage pool capacity is shown inside a container instead of the quota assigned. btrfs is then considered inadequate for a production environment, as reporting the correct disk quota values is essential for proper functioning services, applications or general container use.

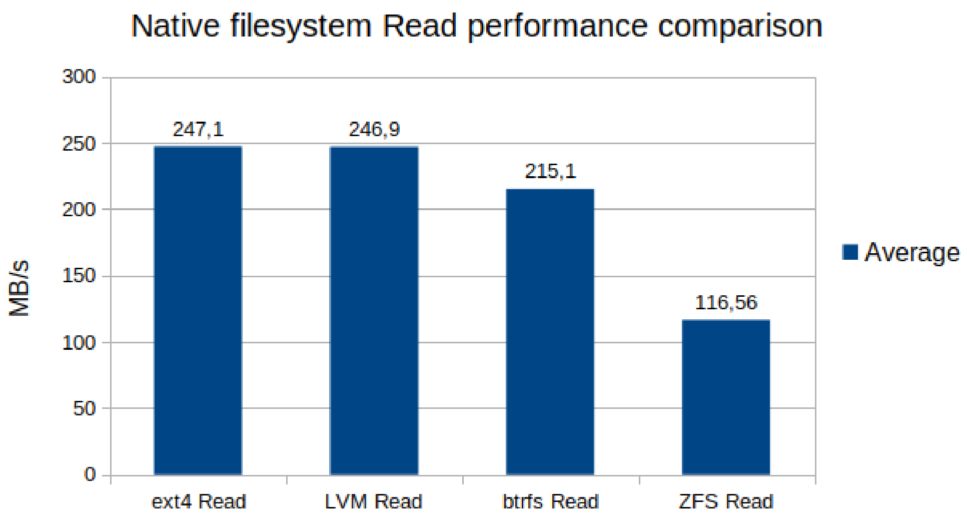


Figure 5.4: Native disk filesystem read performance comparison

In figure 5.5 a comparison is made between the filesystems' write performance on the host machine. The write performance variations are similar to that of the read performance, with ZFS having an even larger decrease in performance due to the partial stripe write issue mentioned in subsection 3.6.1. Also similar to the read performance results, we can see that both the LXD managed LVM partition and the root ext4 partition have near identical performance since the root partition is also managed by LVM. While the LVM managed partition used by LXD has a 0.31% performance advantage over the native ext4 partition, this difference is minimal and mostly due to performance value fluctuations in the limited sample size gathered for the test. Since the host's native filesystem is also managed by LVM, performance metrics should in theory be identical to that of an LVM managed partition used by LXD as the storage backend used is also identical.

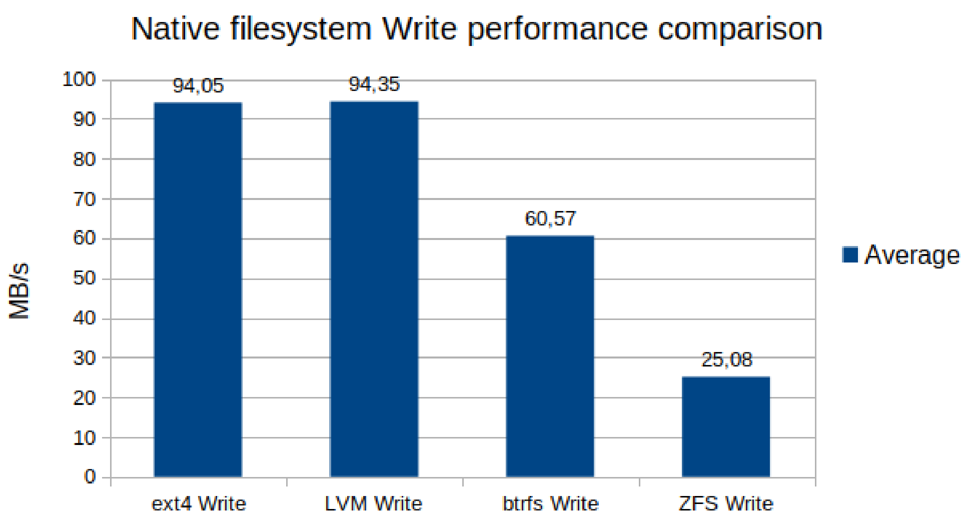


Figure 5.5: Native disk filesystem write performance comparison

In figures 5.6 and 5.7 the read and write performance tests for both container and VM instance types is shown. A significant performance decrease can be observed in both instances types compared to the native LVM performance. Containers show a 5.5% performance advantage over VM instances in read operations and 10% in write operations. The lesser performance overhead from container instances is expected due to not having a fully virtualized kernel as opposed to VMs.

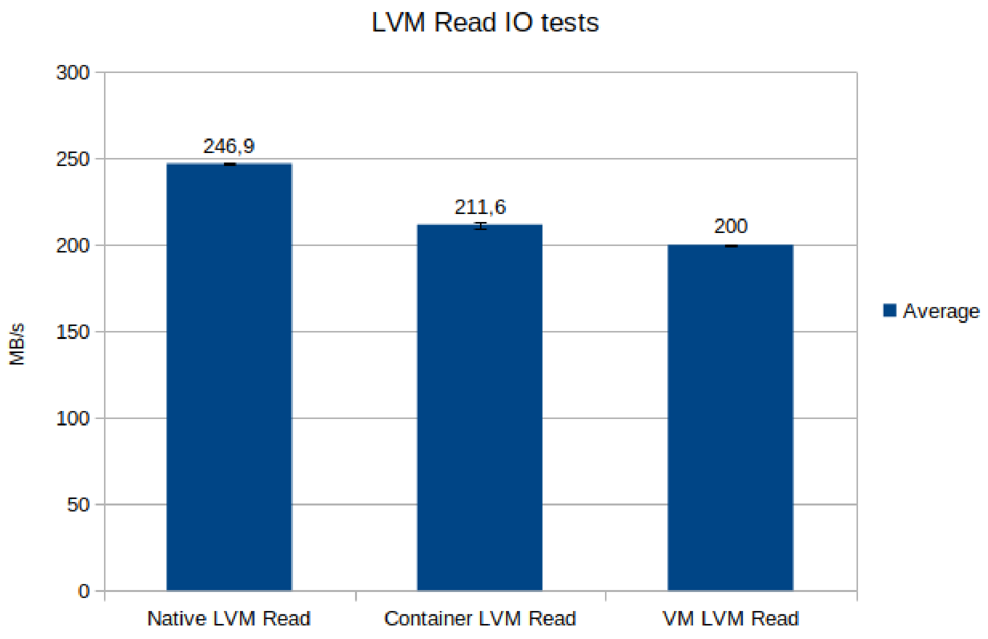


Figure 5.6: LVM Read instance IO tests

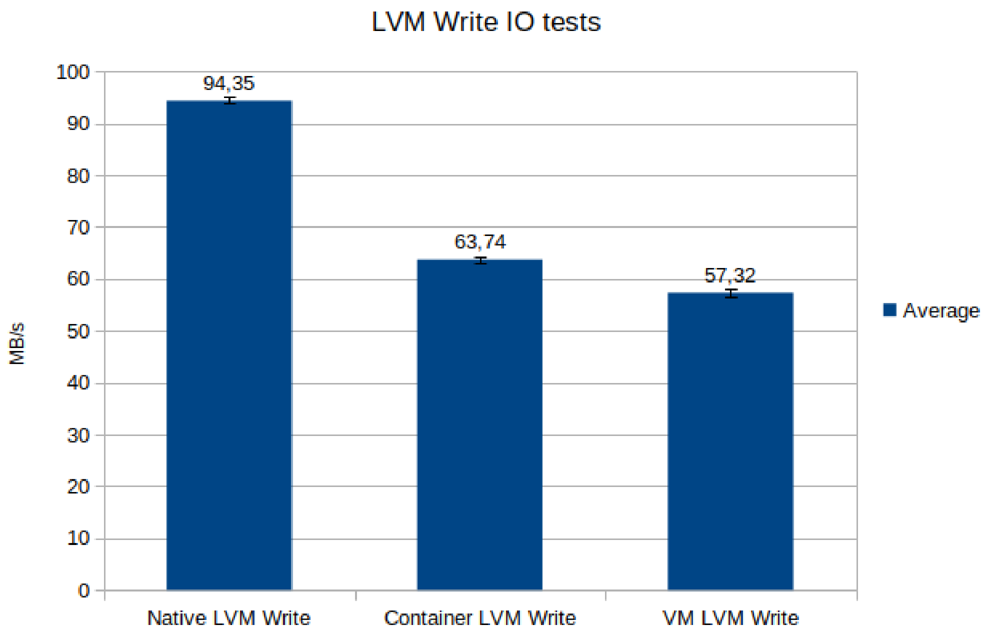


Figure 5.7: LVM Write instance IO tests

These performance values solidify LVM's position as the best filesystem manager for a singular LXD virtualization server's main storage pool. As the default filesystem for the created logical volume is ext4, the performance is significantly greater when compared to other alternatives. While the main drawback of an LVM backed storage pool is instance transfer speed, this does not negatively impact a single server production use case. Instances created on a server will, in most cases, stay within the specific server, without the need to transfer the instance across the local network. For a multiple server clustered environment, *Ceph* would be the ideal filesystem solution.

5.1.4 LXD benchmarks

For LXD instance life cycle benchmarks, the following test environment was used:

- LXD version: 4.9
- Number of container instances: 40
- Number of VM instances: 20

The number of VM instances is lower than containers for testing as their life cycle time is significantly longer. Benchmark results are calculated through the average time each life cycle step takes in 40 container instances and 20 VM instances.

Below are the descriptions of each instance life cycle step:

- Creation: instance creation without initialization (stopped state of execution)
- First boot: first instance boot, needed since container instances take significantly longer on first boot to setup their disks for usage
- Boot: subsequent boot after the first boot
- Reboot: instance reboot
- Shutdown: instance shutdown
- Delete: instance deletion

In figure 5.8 average creation, first boot, boot and reboot time is shown for both instance types. Containers take significantly less time to create, reboot and perform subsequent boots. However the first boot time is significantly longer as LXD needs to remap the filesystem *User Identifier* (UID) and *Group Identifier* (GID) for unprivileged containers [42].

Containers show a 69.9%, 37,4% and 168,3% time advantage over VM instances for creation, boot and reboot instance life cycle steps and a disadvantage of 17,35% on first boot for reasons previously stated.

In figure 5.9 a comparison between the shutdown and deletion time for both instance types is presented. Similar to previous results, containers maintain a time advantage over VM life cycle actions being significantly faster on both shutdown and deletion.

Containers outperform VMs in both shutdown and deletion life cycle steps by 53.5% and 75.4% respectively.

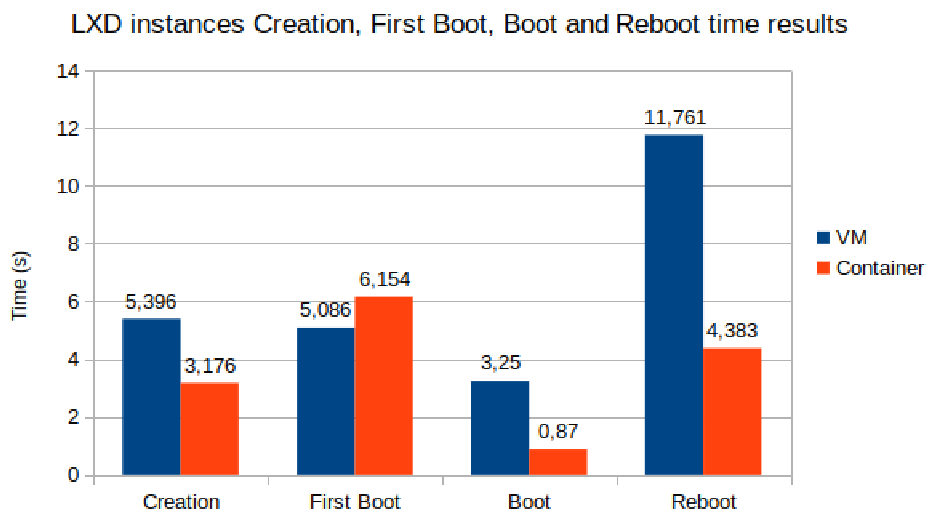


Figure 5.8: LXD instances Creation, First Boot, Boot and Reboot time results

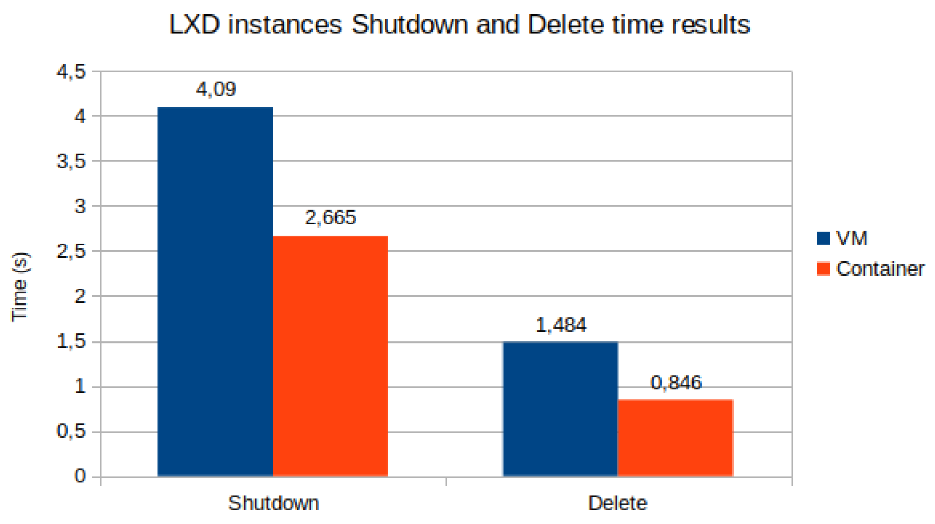


Figure 5.9: LXD instances Shutdown and Delete time results

With the presented results, one may conclude that life cycle event execution time has a significant difference between container and VM instances. When choosing an instance type for any operational scenario these performance values should be taken into account.

5.2 Clustered server performance benchmarks

To benchmark disk IO operations on a clustered setup `dd` was used, following an identical procedure to the tests run in section 5.1.

5.2.1 Cluster nodes hardware specification

The cluster used for testing was comprised of three physical nodes. Below are the specifications of each node:

Node 1 specifications:

- **CPU**: AMD Opteron(tm) Processor 6348
- **Memory**: 4x 8GB DDR3 1600Mhz, Hynix
- **Storage**: 4x Toshiba MG04SCA40EE 4TB

Node 2 specifications:

- **CPU**: AMD Opteron(tm) Processor 6328
- **Memory**: 16x 16GB DDR3, Kingston
- **Storage**: 4x Toshiba MG03SCA200 2TB

Node 3 specifications:

- **CPU**: Intel(R) Xeon(R) CPU D-1520 @ 2.20GHz
- **Memory**: 1x 32GB DDR4, Hynix Semiconductor
- **Storage**: 1x ATA KINGSTON SA400S3 480GB

In a production environment, all nodes would have identical storage devices to ensure that the data is evenly replicated across all nodes. Since this cluster setup is for testing purposes only, the uneven storage device capacity is not an issue since the main objective of the test cluster was to have a working prototype to guarantee that the system is configured correctly.

When a client writes data to a Ceph OSD pool, the data is first written to the primary OSD which then issues the write operations to the secondary and tertiary OSD. After both the secondary and tertiary OSDs acknowledge the write operation, the primary will then forward the write acknowledgement to the client, signaling that the data has been succesfully written to the cluster. This means that the IO performance of a cluster is largely affected by the slowest storage device in the cluster [43, p. 75].

5.2.2 Benchmarks

The cluster's performance was tested between a RAID 6 and non RAID setup on the servers supporting it. As previously stated, Ceph uses the host's disks to intiate OSDs where data will be stored. These disks can either be RAID 6 or non RAID disks. Since Ceph handles data replication and distribution, the added layer of redundancy in a RAID 6 configuration was initially labelled as not necessary.

In figure 5.10 a comparison between disk read performance on a RAID 6 and non RAID setup is made. Containers and VMs show similar IO performance with containers' having a marginal performance advantage. There is no significant read performance difference between both setups as the main performance hit when configuring a RAID 6 array originates from the dual parity bit calculation on writing data to disk. Fluctuations between the performance values are mainly due to the benchmark's limited sample size. Performance values vary from 4.22%, 4.56% and 4.37% between a RAID 6 and non RAID setup for native, container and VM test cases respectively and are therefore considered comparable.

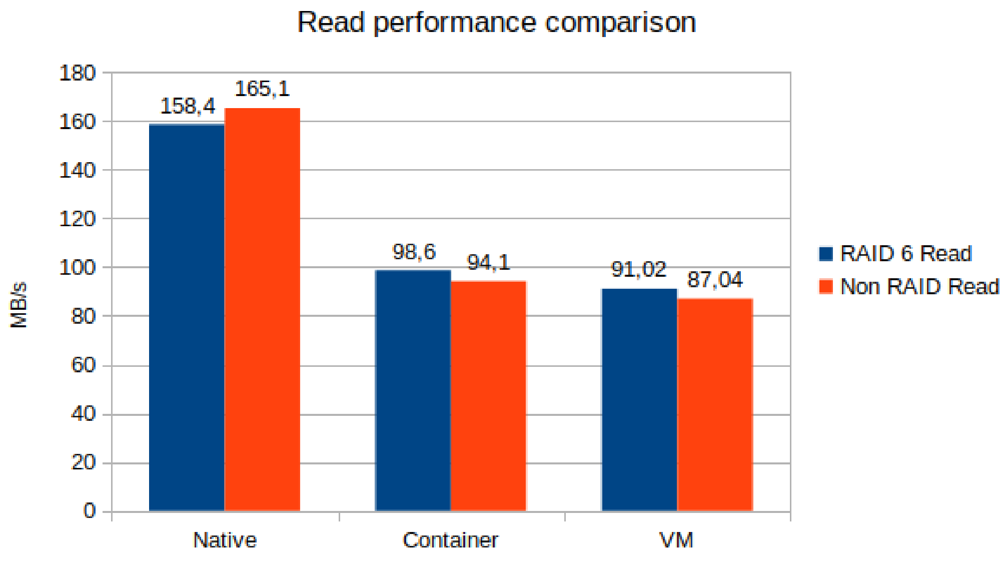


Figure 5.10: Ceph disk read benchmark results

In figure 5.11 a comparison between disk write performance on a RAID 6, non RAID and RAID 6 with write cache enabled is shown. A significant increase in write performance is observed when RAID 6 is not configured, as expected, since parity bits are not calculated.

While there is a significant performance increase in write operations with a non RAID setup, the ease of disk replacement came into question when a cluster is installed in a production environment. In a non RAID setup if a storage device fails, the Ceph OSD's configuration data would have to be manually removed from the Ceph cluster configuration and later readded when a new storage drive is installed on the node. In case of a RAID 6 setup, if one or two devices fail (the maximum number of storage device failures supported by RAID 6) the disks will enter the *degraded mode*, meaning that optimal performance and data redundancy is no longer guaranteed [44], however the RAID 6 disk array will remain functional, allowing data read and write operations to disks.

Another aspect to take into account in a production scenario is the fact that servers will always be backed by an Uninterruptible Power Supply (UPS) which prevents the server from shutting down in case of power loss. This allows us to safely turn on *Write cache* on the hardware RAID controller. Write caching allows for any data from write operations to be stored in the RAID controller first, which leads to a faster response time, increasing performance. The controller is in charge of forwarding the cached data to the physical storage devices. If a UPS was not available and a power failure occurred during a cached write operation to the array of disks, the data not yet written to disk could be corrupted and there would also be a chance of corrupting existing data on the same hardware disks. Since all servers are backed by a UPS and historically no catastrophic issues were registered at Solid Angle, write cache was deemed safe to enable in a production environment.

Performance benefits can be seen in figure 5.11 where a Ceph OSD with a RAID 6 disk with write cache enabled shows a significant performance increase when compared to an OSD with a RAID 6 disk with write cache disabled. The cached setup's performance comes close to that of a non RAID setup with write cache disabled.

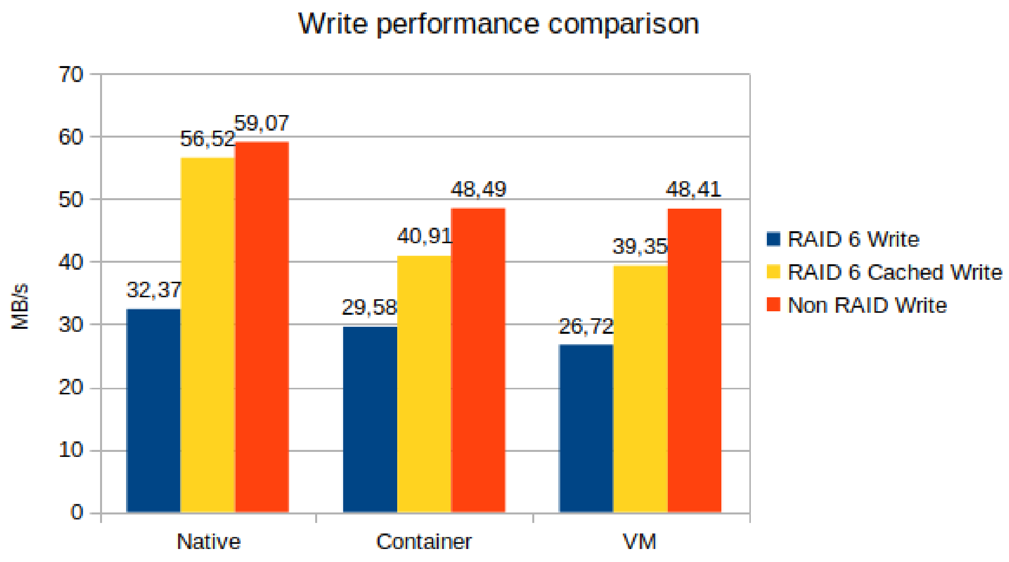


Figure 5.11: Ceph disk write benchmark results

Instance migration time was also benchmarked and the results are shown in figure 5.12. The migration benchmark was run in loops of 1000 iterations. In each iteration the specified instance is moved from one node to another. To ensure that instance size does not affect migration time, tests were run on both large and small container and VM instances. Large instances contained a 400GB file with random content, while small instances have no additional content other than the initial files already present on instance creation.

With these benchmark results, one may conclude that the instance's size does not influence migration time. Since Ceph replicates instance data across all cluster nodes, data does not need to be migrated from one cluster node to another. Instance migration only transfers the execution of the instance from one node to another. Instance type also has no effect on migration time as can be seen.

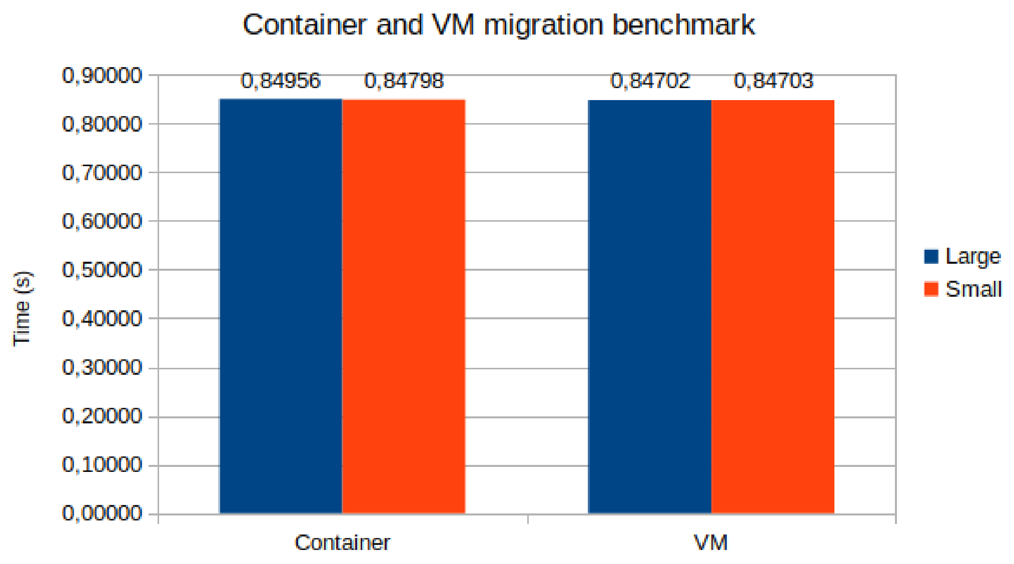


Figure 5.12: LXD instance migration benchmark

5.3 Clustered vs Single server IO performance

While Ceph brings data replication to the table, it comes at a cost in the form of decreased IO performance. In figures 5.13 and 5.14, performance comparisons are shown between Ceph and the single server system with an LVM managed ext4 partition that was previously presented. A clear performance decrease can be observed for both read and write operations in all test cases.

Read performance tests have a performance decrease of 35,8%, 53,4% and 54,49% from a single server LVM setup to a Ceph clustered setup for the native, containers and VMs test cases respectively. Since the IO performance decrease is significant, more than halving in a clustered setup for container and VM instances, it should be one of the main aspects to take into account when opting for a clustered setup in a production scenario as services that require high data throughput will be significantly affected.

For write performance tests, the performance decrease from a single server to a clustered setup is 40,1%, 35,8% and 31,4% for native, container and VM test cases respectively. While the performance decrease for write operation tests is inferior when compared to read operation tests, it still maintains as a significant IO performance decrease.

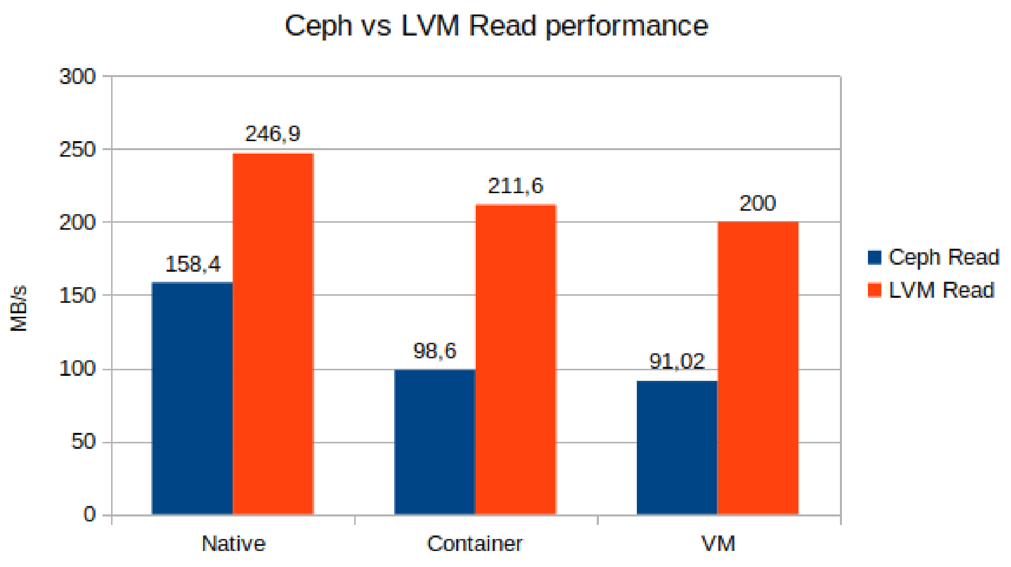


Figure 5.13: Ceph vs LVM read performance comparison

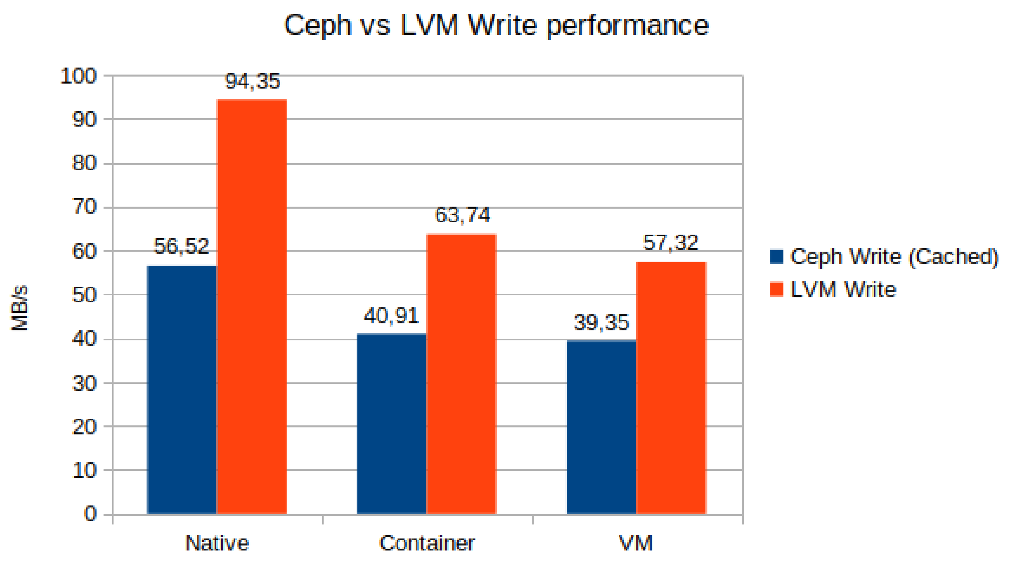


Figure 5.14: Ceph vs LVM write performance comparison

Chapter 6

Discussion

With the results presented above and the documentation of both a single server and a clustered server virtualization system, one may conclude that LXD is a suitable tool when virtualizating underlying physical hardware into multiple virtual instances. However, LXD is not a be-all and end-all virtualization solution, and this chapter addresses some of its limitations.

6.1 Single virtualization server system

When choosing a virtualization tool we must take into consideration the main applications for the virtualized instances. If the goal is to host simple stateless applications, the configuration of LXD along with required software and network dependencies may be unnecessarily complex. Docker containers may be a better fit in this particular scenario as they excel in hosting simple stateless application with little to no setup other than configuring the application to be built and executed in a Docker container.

While an LXD client is made available for other operating systems, mainly Windows and macOS [45], the LXD daemon is only available for the Ubuntu Linux distribution. LXD clients for other operating system are only used for sending and receiving remote requests to and from an LXD server which will be hosted on a Ubuntu based server. Instance hosting is therefore limited to Ubuntu based servers, while instance configuration is accessible through any of the previously mentioned operating systems. This may become an issue if, for example, hosts in a production setup do not use Ubuntu as their operating system. Configuring the developed system in this scenario would imply either installing Ubuntu on an existing host or adding a new Ubuntu server machine to the production setup to host LXD instances. Both options increase time and monetary costs.

6.2 Clustered virtualization server system

Clustering LXD servers brings many benefits, mainly regarding reliability and scalability. When coupled with the Ceph object storage system the reliability factor increases further as data is not stored on a single node. However, clustering may not be an ideal solution for every scenario with some of the main issues arising from hardware and maintenance costs.

An ideal setup for a stable working Ceph and LXD cluster needs three physical hosts with the same type and amount of storage disks. This guarantees that the data is evenly spread throughout all of the node's storage disks, avoiding Ceph data balancing issues. This is more costly than a single server virtualization system as we would require three times the amount of physical hosts for a minimal setup. Given the instance's use cases or data reliability needs, a clustered server setup may be excessively expensive when there is no need of a high level of data redundancy.

Other than hardware costs, there are also associated maintenance costs. Ceph is an object storage tool independent of LXD. To manage a Ceph cluster, a user would have to learn the main concepts and the CLI instructions. This would be needed to manage any erroneous Ceph services or to add new Ceph hosts to the cluster. Learning a new tool requires the system administrator's time and increases monitoring costs.

As shown in section 5.3, a clustered LXD environment increases latency of disk IO operations. This is expected as it depends on network performance, the nodes' storage disk hardware and data replication procedures. The non negligible decrease in disk IO performance may be unacceptable for setups where a high data throughput is essential and must be taken into consideration when opting for a clustered setup.

While costs increase in a clustered environment, a Ceph and LXD cluster still has its own benefits which are far from negligible. The operational scenarios tested in this project show that the cluster has high fault resilience, being able to function normally when supported cluster nodes fail. When opting for a clustered setup in a production enviroment, the additional cost to increase instance availability and data replication must be very carefully weighed to guarantee that the trade-offs ultimately bring a net positive result.

Chapter 7

Conclusion

In this project LXD was explored and tested as an open source virtualization solution for both VM and container instances. The project showed that with the correct guidelines, LXD can be setup for a production environment, allowing easily configurable instance settings (mainly network settings, CPU, memory and storage space allocation) either directly through the CLI or for this project's particular scenario, through resources defined in Puppet.

Simple and efficient instance management was one of the main goals for this project. Since users with all backgrounds must be able to use the system, documentation prepared in the scope of this project was limited, on a technical perspective, to facilitate day to day operations. To further simplify the system's utilization, a graphical interface, LXDUI, was introduced and modified for this project's specific use case. This enabled instance management through a simple user interface that can be used without requiring a deep insight on the inner workings of LXD and the CLI commands it provides.

The developed project makes the following improvements, to any LXD implementation project:

- A well documented CLI and GUI to manage LXD instances (both container and VM)
- A Puppet module to automate the installation of all LXD dependencies and configure the host's network accordingly
- An updated LXD Puppet module to manage both container and VM instances through Puppet
- A modified version of LXDUI featuring only the necessary functionalities for managing LXD instances

For high availability scenarios, the project showed that LXD can also be deployed in a clustered environment. Together with the aid of the Ceph storage backend, LXD provides instance recovery from dead cluster nodes. This is limited only to Ceph since it replicates data across all nodes, meaning that every node will have access to all instances' data.

The clustered LXD development introduces the following components:

- Documentation to manage the cluster both manually through the CLI and automatically through Puppet
- Modified version of LXDUI to support clustering
- Automatic network configuration through Puppet to use a backside private network for LXD and Ceph communication only
- Documentation for cluster disaster recovery

One may conclude that LXD is a viable open source virtualization tool, with an acceptable instance performance. The simple command set coupled with a an intuitive GUI makes the developed system versatile as it can then be used by users with different backgrounds. If clustering is desired, the decrease in IO performance and the increase in hardware cost and maintenance must be thoroughly analyzed to ensure that the benefits (instance recovery from dead nodes, data replication, among others) outweigh the drawbacks.

Acronyms

API	Application Programming Interface.
CLI	Command Line Interface.
CPU	Central Processing Unit.
GID	Group Identifier.
GUI	Graphical User Interface.
KVM	Kernel-based Virtual Machine.
LXC	Linux Container.
RAID	Redundant Array of Inexpensive Disks.
UID	User Identifier.
UPS	Uninterruptible Power Supply.
VM	Virtual Machine.

Bibliography

[1] Red Hat Inc. What is the Linux kernel? https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel. Accessed on: October 20, 2020.

[2] Yang Ye (2020). Ironman LXD. https://bitbucket.org/asolidodev/ironman/wiki/Home. Accessed on: October 26, 2020.

[3] Opensource.com. What is open source. https://opensource.com/resources/what-open-source. Accessed on: October 15 , 2020.

[4] Canonical Ltd. For CTO's: the no-nonsense way to accelerate your business with containers. https://ubuntu.com/engage/whitepaper-containers. Accessed on: March 1, 2021.

[5] William Stallings. *Operating Systems, Internals and Design Principles*. Pearson Education, Inc., publishing as Prentice Hall, 2012.

[6] Langasek Steve and Harper Ryan. Virtual Machines. https://wiki.ubuntu.com/MigratingToNetplan. Accessed on: October 20, 2020.

[7] Canonical Ltd. What's LXD? https://linuxcontainers.org/lxd/introduction/. Accessed on: January 25, 2021.

[8] stgraber (April 22, 2020). Running virtual machines with LXD 4.0 [Discussion post]. https://discuss.linuxcontainers.org/t/running-virtual-machines-with-lxd-4-0/7519. Accessed on: November 3, 2020.

[9] Docker. What is Docker? https://docs.docker.com/get-started/overview/. Accessed on: October 18, 2020.

[10] Docker. Docker frequently asked questions (FAQ). https://docs.docker.com/engine/faq/. Accessed on: October 18, 2020.

[11] kubernetes. What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Accessed on: October 25, 2020.

[12] kubernetes. Nodes. https://kubernetes.io/docs/concepts/architecture/nodes/. Accessed on: October 25, 2020.

[13] kubernetes. Pods. https://kubernetes.io/docs/concepts/workloads/pods/. Accessed on: October 25, 2020.

[14] Khodja Layachi. Deploying Kubernetes on Bare Metal. https://www.inap.com/blog/deploying-kubernetes-on-bare-metal/. Accessed on: October 25, 2020.

[15] Graber Stéphane. Clustering. https://lxd.readthedocs.io/en/latest/clustering/. Accessed on: October 25, 2020.

[16] Canonical Ltd. Setup your LXD server as remote server. https://linuxcontainers.org/lxd/advanced-guide/#setup-your-lxd-server-as-remote-server. Accessed on: April 12, 2021.

[17] Yang Ye (2020). Ironman [Source Code]. https://bitbucket.org/asolidodev/ironman/src/master/. Accessed on: October 26, 2020.

[18] Virtio. Virtio. https://wiki.libvirt.org/page/Virtio. Accesed on: April 18, 2021.

[19] Microsoft 2021. Sysprep (Generalize) a Windows installation. https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/sysprep--generalize--a-windows-installation. Accessed on: April 18, 2021.

[20] Cisco Systems. Traffic regulators: Network interfaces, hubs, switches, bridges, routers, and firewalls. September 1999.

[21] Oracle Corporation and/or its affiliates. How DHCP Works. https://docs.oracle.com/cd/E19253-01/816-4554/dhcp-overview-3/index.html, 2010. Accessed on: October 10, 2020.

[22] Graber Stéphane. Network Configuration. https://lxd.readthedocs.io/en/latest/networks/#network-bridge. Accessed on: February 24, 2021.

[23] Graber Stéphane. Storage configuration. `https://lxd.readthedocs.io/en/latest/storage/#storage-backends-and-supported-functions`, 2020. Accessed on: October 14, 2020.

[24] David A Patterson, Garth Gibson, and Randy H Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). Berkeley, CA 94720, USA, 1988.

[25] Sun Microsystems. RAID 6 Arrays. `https://docs.oracle.com/cd/E19494-01/820-1260-15/appendixf.html#50548797_51002`. Accessed on: May 8, 2021.

[26] Bonwick Jeff, Ahrens Matt, Henson Val, Maybee Mark, and Shellenbaum Mark. The Zettabyte File System.

[27] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. San Francisco, CA, USA, 2003.

[28] OpenZFS. OpenZFS ZFS Source code (Version 2.0.3), [Source code]. `https://github.com/openzfs/zfs/blob/master/module/zfs/arc.c`. Accessed on: February 14, 2021.

[29] OpenZFS. Hardware RAID controllers. `https://openzfs.github.io/openzfs-docs/Performance%20and%20Tuning/Hardware.html#hardware-raid-controllers`. Accessed on: February 19, 2021.

[30] John Henry Hartman. The Zebra Striped Network File System. Master's thesis, University of California at Berkeley, 1994.

[31] stgraber (December 22, 2016). lxd btrfs disk quota [Discussion post]. LXD git repository issues. `https://github.com/lxc/lxd/issues/2756#issuecomment-268854269`. Accessed on: February 12, 2021.

[32] Ubuntu. Lvm. `https://wiki.ubuntu.com/Lvm`. Accessed on: February 10, 2021.

[33] Smith Ethan (2020). Puppet snapd [Source Code]. `https://github.com/ethanhs/puppet-snapd`. Accessed on: April 2, 2021.

[34] OVH (2020). Puppet LXD mangament module [Source Code]. `https://github.com/ovh/lxd-puppet-module`. Accessed on: April 2, 2021.

[35] Red Hat Inc. Ceph - Chapter 3. Networking Recommendations. `https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/1.3/html/hardware_guide/networking_recommendations`. Accessed on: June 8, 2021.

[36] Ceph authors and contributors. Intro to Ceph. https://docs.ceph.com/en/latest/start/intro/. Accessed on: April 24, 2021.

[37] Ceph authors and contributors. Installing Ceph. https://docs.ceph.com/en/latest/install/index.html. Accessed on: April 26, 2021.

[38] Ceph authors and contributors. Cephadm. https://docs.ceph.com/en/latest/cephadm/. Accessed on: April 26, 2021.

[39] Ceph authors and contributors. Deploying a new Ceph cluster. https://docs.ceph.com/en/latest/cephadm/install/. Accessed on: April 26, 2021.

[40] Ceph authors and contributors. Ceph Pools - Get the number of replcias. https://docs.ceph.com/en/latest/rados/operations/pools/?#get-the-number-of-object-replicas. Accessed on: April 27, 2021.

[41] Canonical Ltd. Integrations - LXD. https://ubuntu.com/ceph/docs/integration-lxd. Accessed on: April 26, 2021.

[42] tomp (Thomas Parrott). Container slower boot times when compared to VMs [Discussion post]. https://discuss.linuxcontainers.org/t/container-slower-first-boot-time-compared-to-vms/11401/2. Accessed on: June21, 2021.

[43] Karan Singh. *Learning Ceph*. Packt Publishing Ltd., 2015.

[44] Derek Vadala. *Managing RAID on Linux*. O' Reilly & Associates, Inc., 2003.

[45] Graber Stéphan. LXD client on Windows and macOS. https://ubuntu.com/blog/lxd-client-on-windows-and-macos, 2017. Accessed on: May 15, 2021.

[46] Skysilk Inc. LXC vs KVM | What is the difference? https://www.skysilk.com/blog/2019/lxc-vs-kvm/. Accessed on: October 16, 2020.

[47] MOREnet. How to Locate IP, Gateway, Subnet and DNS Information. https://web.mit.edu/rama/www/IP_tools.htm, January 2003. Accessed on: October 10, 2020.

[48] Graber Stéphane. instances, containers and virtual-machines. https://lxd.readthedocs.io/en/latest/rest-api/#instances-containers-and-virtual-machines, 2020. Accessed on: October 14, 2020.

[49] Olivo Kyle. How Linux containers work. https://kyleolivo.com/dev/2016/08/15/containers-how-do-they-work/. Accessed on: October 12, 2020.

[50] Lane Katie. Kubernetes vs. Docker: What Does it Really Mean? https://www.sumologic.com/blog/kubernetes-vs-docker/. Accessed on: October 25, 2020.

[51] Canonical Ltd. cloud-init Documentation. https://cloudinit.readthedocs.io/en/latest/#. Accessed on: October 30, 2020.

[52] btrfs. RAID56. https://btrfs.wiki.kernel.org/index.php/RAID56. Accessed on: February 10, 2021.

[53] Debian. LVM. https://wiki.debian.org/LVM. Accessed on: January 13, 2021.

[54] VMware Inc. What is a hypervisor? https://www.vmware.com/topics/glossary/content/hypervisor. Accessed on: June 16, 2021.